man pages section 9: DDI and DKI
Kernel Functions
Contents

Preface 23

Introduction 29
Intro(9F) 30

Kernel Functions for Drivers 49
adjmsg(9F) 50
alloca(9F) 51
anocancel(9F) 54
aphysio(9F) 55
ASSERT(9F) 57
assert(9F) 58
backq(9F) 59
bcanput(9F) 60
bcanputnext(9F) 61
bcmp(9F) 62
bcopy(9F) 63
biodone(9F) 68
bioerror(9F) 70
biofini(9F) 71
bioinit(9F) 72
biomodified(9F) 73
bioriset(9F) 74
biosize(9F) 75
biowait(9F) 76
bp_mapin(9F) 77
bp_mapout(9F) 78
btop(9F) 79
btopr(9F) 80
bufcall(9F) 81
bzero(9F) 84
canput(9F) 85
canputnext(9F) 86
crbuf(9F) 87
cmn_err(9F) 88
condvar(9F) 93
copyb(9F) 96
copyin(9F) 98
copymsg(9F) 100
copyout(9F) 102
csx_AccessConfigurationRegister(9F) 104
csx_ConvertSize(9F) 106
csx_ConvertSpeed(9F) 107
csx_CS_DDI_Info(9F) 108
csx_DeregisterClient(9F) 110
csx_DupHandle(9F) 111
csx_Error2Text(9F) 113
csx_Event2Text(9F) 114
csx_FreeHandle(9F) 115
csx_Get16(9F) 116
csx_Get32(9F) 117
csx_Get64(9F) 118
csx_Get8(9F) 119
csx_GetEventMask(9F) 120
csx_GetFirstClient(9F) 122
csx_GetFirstTuple(9F) 124
csx_GetHandleOffset(9F) 126
csx_GetMappedAddr(9F) 127
csx_GetNextClient(9F) 128
csx_GetNextTuple(9F) 130
csx_GetStatus(9F) 132
csx_GetTupleData(9F) 136
csx_MakeDeviceNode(9F) 138
csx_MapLogSocket(9F) 140
csx_MapMemPage(9F) 141
csx_ModifyConfiguration(9F) 142
csx_ModifyWindow(9F) 144
csx_Parse_CISTPL_BATTERY(9F) 146
csx_Parse_CISTPL_BYTEORDER(9F) 147
csx_Parse_CISTPL_CFTABLE_ENTRY(9F) 149
csx_Parse_CISTPL_CONFIG(9F) 155
csx_Parse_CISTPL_DATE(9F) 158
csx_Parse_CISTPL_DEVICE(9F) 159
csx_Parse_CISTPL_DEVICE_A(9F) 162
csx_Parse_CISTPL_DEVICE_GEO(9F) 165
csx_Parse_CISTPL_DEVICE_GEO_A(9F) 167
csx_Parse_CISTPL_DEVICE_OA(9F) 169
csx_Parse_CISTPL_DEVICE_OC(9F) 172
csx_Parse_CISTPL_FORMAT(9F) 175
csx_Parse_CISTPL_FUNCE(9F) 177
csx_Parse_CISTPL_FUNCID(9F) 185
csx_Parse_CISTPL_GEOMETRY(9F) 187
csx_Parse_CISTPL_JEDEC_A(9F) 188
csx_Parse_CISTPL_JEDEC_C(9F) 190
csx_Parse_CISTPL_LINKTARGET(9F) 192
csx_Parse_CISTPL_LONGLINK_A(9F) 193
csx_Parse_CISTPL_LONGLINK_C(9F) 195
csx_Parse_CISTPL_LONGLINK_MFC(9F) 197
csx_Parse_CISTPL_MANFID(9F) 199
csx_Parse_CISTPL_ORG(9F) 200
csx_Parse_CISTPL_SPCL(9F) 201
csx_Parse_CISTPL_SWIL(9F) 203
csx_Parse_CISTPL_VERS_1(9F) 204
csx_Parse_CISTPL_VERS_2(9F) 205
csx_ParseTuple(9F) 206
csx_Put16(9F) 208
csx_Put32(9F) 209
csx_Put64(9F) 210
csx_Put8(9F) 211
csx_RegisterClient(9F) 212
csx_ReleaseConfiguration(9F) 215
csx_ReleaseIO(9F) 217
csx_ReleaseIRQ(9F) 222
csx_ReleaseSocketMask(9F) 225
csx_ReleaseWindow(9F) 227
csx_RemoveDeviceNode(9F) 232
csx_RepGet16(9F) 234
csx_RepGet32(9F) 236
csx_RepGet64(9F) 238
csx_RepGet8(9F) 240
csx_RepPut16(9F) 242
csx_RepPut32(9F) 244
csx_RepPut64(9F) 246
csx_RepPut8(9F) 248
csx_RequestConfiguration(9F) 250
csx_RequestIO(9F) 254
csx_RequestIRQ(9F) 259
csx_RequestSocketMask(9F) 262
csx_RequestWindow(9F) 264
csx_ResetFunction(9F) 269
csx_SetEventMask(9F) 270
csx_SetHandleOffset(9F) 272
csx_ValidateCIS(9F) 273
cv_broadcast(9F) 274
cv_destroy(9F) 277
cv_init(9F) 280
cv_signal(9F) 283
cv_timedwait(9F) 286
cv_timedwait_sig(9F) 289
cv_wait(9F) 292
cv_wait.sig(9F) 295
datamsq(9F) 298
ddi_add_intr(9F) 299
ddi_add_softintr(9F) 302
ddi_binding_name(9F) 307
ddi_btop(9F) 310
ddi_btopr(9F) 311
ddi_check_acc_handle(9F) 312
ddi_check_dma_handle(9F) 314
ddi_copyin(9F) 316
ddi_copyout(9F) 319
ddi_create_minor_node(9F) 322
ddi_device_copy(9F) 325
ddi_device_zero(9F) 327
ddi_devid_compare(9F) 328
ddi_devid_free(9F) 332
ddi_devid_init(9F) 336
ddi_devid_register(9F) 340
ddi_devid_sizeof(9F) 344
ddi_devid_str_decode(9F) 348
ddi_devid_str_encode(9F) 352
ddi_devid_str_free(9F) 356
ddi_devid_unregister(9F) 360
ddi_devid_valid(9F) 364
ddi_dev_is_needed(9F) 368
ddi_dev_is_sid(9F) 370
ddi_devmap_segmap(9F) 371
ddi_dev_nintrs(9F) 373
ddi_dev_nregs(9F) 374
ddi_dev_regsize(9F) 375
ddi_dev_report_fault(9F) 376
ddi_dma_addr_bind_handle(9F) 379
ddi_dma_addr_setup(9F) 383
ddi_dma_alloc_handle(9F) 385
ddi_dma_buf_bind_handle(9F) 387
ddi_dma_buf_setup(9F) 391
ddi_dma_burstsizes(9F) 393
ddi_dma_coff(9F) 394
ddi_dma_curwin(9F) 395
ddi_dma_devalign(9F) 396
ddi_dmae_1stparty(9F) 397
ddi_dmae(9F) 401
ddi_dmae_alloc(9F) 405
ddi_dmae_disable(9F) 409
ddi_dmae_enable(9F) 413
ddi_dmae_getattr(9F) 417
ddi_dmae_getcnt(9F)  421
ddi_dmae_getlim(9F)  425
ddi_dmae_prog(9F)  429
ddi_dmae_release(9F)  433
ddi_dmae_stop(9F)  437
ddi_dma_free(9F)  441
ddi_dma_free_handle(9F)  442
ddi_dma_get_attr(9F)  443
ddi_dma_getwin(9F)  444
ddi_dma_htoc(9F)  446
ddi_dma_mem_alloc(9F)  447
ddi_dma_mem_free(9F)  450
ddi_dma_movwin(9F)  451
ddi_dma_nextcookie(9F)  453
ddi_dma_nextseg(9F)  455
ddi_dma_nextwin(9F)  457
ddi_dma_numwin(9F)  459
ddi_dma_segtocookie(9F)  460
ddi_dma_set_sbus64(9F)  462
ddi_dma_setup(9F)  463
ddi_dma_sync(9F)  465
ddi_dma_unbind_handle(9F)  467
ddi_driver_major(9F)  468
ddi_driver_name(9F)  469
ddi_enter_critical(9F)  470
ddi_exit_critical(9F)  471
ddi_ffs(9F)  472
ddi_fls(9F)  473
ddi_get16(9F)  474
ddi_get32(9F)  476
ddi_get64(9F)  478
ddi_get8(9F)  480
ddi_getb(9F)  482
ddi_get_cred(9F)  484
ddi_get_devstate(9F)  485
ddi_get_driver_private(9F)  486
ddi_get_iblock_cookie(9F)  487
ddi_getimminor(9F)  490
ddi_get_instance(9F) 491
ddi_get_kt_did(9F) 492
ddi_getl(9F) 493
ddi_get_lbolt(9F) 495
ddi_getll(9F) 496
ddi_getlongprop(9F) 498
ddi_getlongprop_buf(9F) 502
ddi_get_name(9F) 506
ddi_get_parent(9F) 507
ddi_get_pid(9F) 508
ddi_getprop(9F) 509
ddi_getproplen(9F) 513
ddi_get_soft_iblock_cookie(9F) 517
ddi_get_soft_state(9F) 524
ddi_get_time(9F) 529
ddi_getw(9F) 530
ddi_in_panic(9F) 532
ddi_intr_hilevel(9F) 533
ddi_io_get16(9F) 534
ddi_io_get32(9F) 536
ddi_io_get8(9F) 538
ddi_io_getb(9F) 540
ddi_io_getl(9F) 542
ddi_io_getw(9F) 544
ddi_iomin(9F) 546
ddi_iopb_alloc(9F) 547
ddi_iopb_free(9F) 549
ddi_io_put16(9F) 551
ddi_io_put32(9F) 553
ddi_io_put8(9F) 555
ddi_io_putb(9F) 557
ddi_io_putl(9F) 559
ddi_io_putw(9F) 561
ddi_io_rep_get16(9F) 563
ddi_io_rep_get32(9F) 565
ddi_io_rep_get8(9F) 567
ddi_io_rep_getb(9F) 569
ddi_io_rep_getl(9F) 571
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>insq(9F)</td>
<td>1003</td>
</tr>
<tr>
<td>inw(9F)</td>
<td>1005</td>
</tr>
<tr>
<td>IOC_CONVERT_FROM(9F)</td>
<td>1007</td>
</tr>
<tr>
<td>kmem_alloc(9F)</td>
<td>1008</td>
</tr>
<tr>
<td>kmem_cache_alloc(9F)</td>
<td>1010</td>
</tr>
<tr>
<td>kmem_cache_create(9F)</td>
<td>1015</td>
</tr>
<tr>
<td>kmem_cache_destroy(9F)</td>
<td>1020</td>
</tr>
<tr>
<td>kmem_cache_free(9F)</td>
<td>1025</td>
</tr>
<tr>
<td>kmem_free(9F)</td>
<td>1030</td>
</tr>
<tr>
<td>kmem_zalloc(9F)</td>
<td>1032</td>
</tr>
<tr>
<td>kstat_create(9F)</td>
<td>1034</td>
</tr>
<tr>
<td>kstat_delete(9F)</td>
<td>1036</td>
</tr>
<tr>
<td>kstat_install(9F)</td>
<td>1037</td>
</tr>
<tr>
<td>kstat_named_init(9F)</td>
<td>1038</td>
</tr>
<tr>
<td>kstat_named_setstr(9F)</td>
<td>1039</td>
</tr>
<tr>
<td>kstat_queue(9F)</td>
<td>1040</td>
</tr>
<tr>
<td>kstat_runq_back_to_waitq(9F)</td>
<td>1042</td>
</tr>
<tr>
<td>kstat_runq_enter(9F)</td>
<td>1044</td>
</tr>
<tr>
<td>kstat_runq_exit(9F)</td>
<td>1046</td>
</tr>
<tr>
<td>kstat_waitq_enter(9F)</td>
<td>1048</td>
</tr>
<tr>
<td>kstat_waitq_exit(9F)</td>
<td>1050</td>
</tr>
<tr>
<td>kstat_waitq_to_runq(9F)</td>
<td>1052</td>
</tr>
<tr>
<td>linkb(9F)</td>
<td>1054</td>
</tr>
<tr>
<td>makecom(9F)</td>
<td>1055</td>
</tr>
<tr>
<td>makecom_g0(9F)</td>
<td>1057</td>
</tr>
<tr>
<td>makecom_g0_s(9F)</td>
<td>1059</td>
</tr>
<tr>
<td>makecom_g1(9F)</td>
<td>1061</td>
</tr>
<tr>
<td>makecom_g5(9F)</td>
<td>1063</td>
</tr>
<tr>
<td>makedevice(9F)</td>
<td>1065</td>
</tr>
<tr>
<td>max(9F)</td>
<td>1066</td>
</tr>
<tr>
<td>min(9F)</td>
<td>1067</td>
</tr>
<tr>
<td>minphys(9F)</td>
<td>1068</td>
</tr>
<tr>
<td>mkiocb(9F)</td>
<td>1070</td>
</tr>
<tr>
<td>mod_info(9F)</td>
<td>1073</td>
</tr>
<tr>
<td>mod_install(9F)</td>
<td>1074</td>
</tr>
<tr>
<td>mod_remove(9F)</td>
<td>1075</td>
</tr>
<tr>
<td>msgdsz(9F)</td>
<td>1076</td>
</tr>
<tr>
<td>msgpullup(9F)</td>
<td>1077</td>
</tr>
</tbody>
</table>
nvlist_lookup_int32(9F) 1156
nvlist_lookup_int32_array(9F) 1158
nvlist_lookup_int64(9F) 1160
nvlist_lookup_int64_array(9F) 1162
nvlist_lookup_string(9F) 1164
nvlist_lookup_string_array(9F) 1166
nvlist_lookup_uint16(9F) 1168
nvlist_lookup_uint16_array(9F) 1170
nvlist_lookup_uint32(9F) 1172
nvlist_lookup_uint32_array(9F) 1174
nvlist_lookup_uint64(9F) 1176
nvlist_lookup_uint64_array(9F) 1178
nvlist_next_nvpair(9F) 1180
nvlist_pack(9F) 1182
nvlist_remove(9F) 1184
nvlist_remove_all(9F) 1185
nvlist_size(9F) 1186
nvlist_unpack(9F) 1188
nvpair_name(9F) 1190
nvpair_type(9F) 1192
nvpair_value_byte(9F) 1194
nvpair_value_byte_array(9F) 1196
nvpair_value_int16(9F) 1198
nvpair_value_int16_array(9F) 1200
nvpair_value_int32(9F) 1202
nvpair_value_int32_array(9F) 1204
nvpair_value_int64(9F) 1206
nvpair_value_int64_array(9F) 1208
nvpair_value_string(9F) 1210
nvpair_value_string_array(9F) 1212
nvpair_value_uint16(9F) 1214
nvpair_value_uint16_array(9F) 1216
nvpair_value_uint32(9F) 1218
nvpair_value_uint32_array(9F) 1220
nvpair_value_uint64(9F) 1222
nvpair_value_uint64_array(9F) 1224
OTHERQ(9F) 1226
otherq(9F) 1227
Preface

Both novice users and those familiar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).

Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.

Section 9F describes the kernel functions available for use by device drivers.

Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and man(1) for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "filename . . .".

| Separator. Only one of the arguments separated by this character can be specified at a time.

{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.
PROTOCOL
This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION
This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.

IOCTL
This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the ioctl(2) system call is called ioctl and generates its own heading. ioctl calls for a specific device are listed alphabetically (on the man page for that specific device). ioctl calls are used for a particular class of devices all of which have an io ending, such as mtio(7I).

OPTIONS
This section lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.

OPERANDS
This section lists the command operands and describes how they affect the actions of the command.

OUTPUT
This section describes the output – standard output, standard error, or output files – generated by the command.

RETURN VALUES
If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.

ERRORS
On failure, most functions place an error code in the global variable errno indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than
one condition can cause the same error, each condition is described in a separate paragraph under the error code.

**USAGE**
This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality:

- Commands
- Modifiers
- Variables
- Expressions
- Input Grammar

**EXAMPLES**
This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.

**ENVIRONMENT VARIABLES**
This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

**EXIT STATUS**
This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.

**FILES**
This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

**ATTRIBUTES**
This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.

**SEE ALSO**
This section lists references to other man pages, in-house documentation, and outside publications.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAGNOSTICS</td>
<td>This section lists diagnostic messages with a brief explanation of the condition causing the error.</td>
</tr>
<tr>
<td>WARNINGS</td>
<td>This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.</td>
</tr>
<tr>
<td>NOTES</td>
<td>This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.</td>
</tr>
<tr>
<td>BUGS</td>
<td>This section describes known bugs and, wherever possible, suggests workarounds.</td>
</tr>
</tbody>
</table>
Introduction
Intro(9F)

NAME
Intro – introduction to DDI/DKI functions

DESCRIPTION
Section 9F describes the kernel functions available for use by device drivers. See Intro(9E) for an overview of device driver interfaces.

In this section, the information for each driver function is organized under the following headings:

- **NAME** summarizes the function’s purpose.
- **SYNOPSIS** shows the syntax of the function’s entry point in the source code. #include directives are shown for required headers.
- **INTERFACE LEVEL** describes any architecture dependencies.
- **ARGUMENTS** describes any arguments required to invoke the function.
- **DESCRIPTION** describes general information about the function.
- **RETURN VALUES** describes the return values and messages that can result from invoking the function.
- **CONTEXT** indicates from which driver context (user, kernel, interrupt, or high-level interrupt) the function can be called.

A driver function has **user context** if it was directly invoked because of a user thread. The read(9E) entry point of the driver, invoked by a read(2) system call, has user context.

A driver function has **kernel context** if it was invoked by some other part of the kernel. In a block device driver, the strategy(9E) entry point may be called by the page daemon to write pages to the device. The page daemon has no relation to the current user thread, so in this case strategy(9E) has kernel context.

**Interrupt context** is kernel context, but also has an interrupt level associated with it. Driver interrupt routines have interrupt context.

**High-level interrupt context** is a more restricted form of interrupt context. If ddi_intr_hilevel(9F) indicates that an interrupt is high-level, driver interrupt routines added for that interrupt with ddi_add_intr(9F) run in high-level interrupt context. These interrupt routines are only allowed to call ddi_trigger_softintr(9F) mutex_enter(9F) and mutex_exit(9F). Furthermore, mutex_enter(9F) and mutex_exit(9F) may only be called on mutexes initialized with the ddi_iblock_cookie returned by ddi_get_iblock_cookie(9F).

**SEE ALSO** indicates functions that are related by usage and sources, and which can be referred to for further information.

**EXAMPLES** shows how the function can be used in driver code.

Every driver MUST include <sys/ddi.h> and <sys/sunddi.h>, in that order, and as the last files the driver includes.

<table>
<thead>
<tr>
<th>STREAMS Kernel Function Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following table summarizes the STREAMS functions described in this section.</td>
</tr>
<tr>
<td>Routine</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>adjmsg</td>
</tr>
<tr>
<td>allocb</td>
</tr>
<tr>
<td>backq</td>
</tr>
<tr>
<td>bcanput</td>
</tr>
<tr>
<td>bcanputnext</td>
</tr>
<tr>
<td>bufcall</td>
</tr>
<tr>
<td>canput</td>
</tr>
<tr>
<td>canputnext</td>
</tr>
<tr>
<td>clrbuf</td>
</tr>
<tr>
<td>copyb</td>
</tr>
<tr>
<td>copymsg</td>
</tr>
<tr>
<td>datamsg</td>
</tr>
<tr>
<td>dupb</td>
</tr>
<tr>
<td>dupmsg</td>
</tr>
<tr>
<td>enableok</td>
</tr>
<tr>
<td>esballocc</td>
</tr>
<tr>
<td>esbbcall</td>
</tr>
<tr>
<td>flushband</td>
</tr>
<tr>
<td>flushq</td>
</tr>
<tr>
<td>freeb</td>
</tr>
<tr>
<td>freemsg</td>
</tr>
<tr>
<td>freezestr</td>
</tr>
<tr>
<td>getq</td>
</tr>
<tr>
<td>insq</td>
</tr>
<tr>
<td>linkb</td>
</tr>
<tr>
<td>msgdsize</td>
</tr>
<tr>
<td>msgpullup</td>
</tr>
<tr>
<td>mt-streams</td>
</tr>
<tr>
<td>noenable</td>
</tr>
<tr>
<td>Routine</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>OTHERQ</td>
</tr>
<tr>
<td>pullupmsg</td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td>putbq</td>
</tr>
<tr>
<td>putctl</td>
</tr>
<tr>
<td>putctl1</td>
</tr>
<tr>
<td>putnext</td>
</tr>
<tr>
<td>putnextctl</td>
</tr>
<tr>
<td>putq</td>
</tr>
<tr>
<td>qbufcall</td>
</tr>
<tr>
<td>qenable</td>
</tr>
<tr>
<td>qprocson</td>
</tr>
<tr>
<td>qprocsoff</td>
</tr>
<tr>
<td>qreply</td>
</tr>
<tr>
<td>qsize</td>
</tr>
<tr>
<td>qtimeout</td>
</tr>
<tr>
<td>qunbufcall</td>
</tr>
<tr>
<td>quntimeout</td>
</tr>
<tr>
<td>qwait</td>
</tr>
<tr>
<td>qwait_sig</td>
</tr>
<tr>
<td>qwriter</td>
</tr>
<tr>
<td>RD</td>
</tr>
<tr>
<td>rmvb</td>
</tr>
<tr>
<td>rmvq</td>
</tr>
<tr>
<td>SAMESTR</td>
</tr>
<tr>
<td>strlog</td>
</tr>
<tr>
<td>strqget</td>
</tr>
<tr>
<td>strqset</td>
</tr>
<tr>
<td>testb</td>
</tr>
</tbody>
</table>
The following table summarizes the functions not specific to STREAMS.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>unbufcall</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>unfreezestr</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>unlinkb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>WR</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>ASSERT</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>anocancel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>aphysio</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bcmp</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bcopy</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>biodone</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>biocln</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biofini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioinit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biomodified</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biosize</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioerror</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioreset</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biowait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bp_mapin</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bp_mapout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>btop</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>btopr</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bzero</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>cmn_err</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>copyin</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>copyout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>cv_broadcast</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_signal</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_timedwait</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_wait</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_wait_sig</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_add_intr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_add_softintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_btop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_btopr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_copyin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_copyout</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_create_minor_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_is_sid</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_nintrs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_nregs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_regsize</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_device_copy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_device_zero</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_devmap_segmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_addr_bind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_addr_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_alloc_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_buf_bind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_buf_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_burstsizes</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_coff</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_curwin</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ddi_dma_devalign</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_free_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_getwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_htoc</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_mem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_mem_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_movwin</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_nextcookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_nextseg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_nextwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_numwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_segtocookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_set_sbus64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_sync</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_unbind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dmae</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_1stparty</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_alloc</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_disable</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_enable</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_getattr</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_getcnt</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_getlim</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_prog</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_release</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_dmae_stop</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>ddi_enter_critical</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>ddi_exit_critical</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_ffs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_fls</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_cred</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_driver_private</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_iblock_cookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_instance</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_name</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_parent</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_soft_iblock_cookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_soft_state</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getlongprop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getlongprop_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getprop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getproplen</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_intr_hilevel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_getl</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

Intro(9F)
<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iomin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iopb_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iopb_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_map_regs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_intercept</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_nointercept</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_set_device_acc_attr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_free</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>
## Routine Types

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mmap_get_model</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_model_convert_from</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_node_name</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peekc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokec</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokes</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_create</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_exists</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_get_int</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_byte_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_int_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>ddi_prop_lookup_string</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_string_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_modify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_op</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_remove</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_remove_all</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_undefine</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_byte_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_int</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_int_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_string</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_string_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_ptob</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put11</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_regs_map_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_regs_map_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_intr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_minor_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_softintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>ddi_rep_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get1</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get1l</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put1</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put1l</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_report_dev</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_root_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_segmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_segmap_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_set_driver_private</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_slaveonly</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_fini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_zalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_trigger_softintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_umem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_umem_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_unmap_regs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>delay</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>devmap_default_access</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_devmem_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_do_ctxmgt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_load</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_set_ctx_timeout</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_unmem_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_unload</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>disksort</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>drv_getparm</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_hztousec</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_priv</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_usectohz</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_usecwait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>free_pktiopb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>freerbuf</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>get_pktiopb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>geterror</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getmajor</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getminor</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getrbuf</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>hat_getkpfnum</td>
<td>DKI only</td>
</tr>
<tr>
<td>inb</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>inl</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>inw</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>kmem_alloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>kmem_free</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>kmem_zalloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>kstat_create</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_delete</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_install</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_named_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_queue</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_back_to_waitq</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_exit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_waitq_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_waitq_exit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_waitq_to_runq</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>makecom_g0</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>makecom_g0_s</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>makecom_g1</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>makecom_g5</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>makesdevice</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>max</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>min</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>minphys</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mod_info</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mod_install</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mod_remove</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_exit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_owned</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mutex_tryenter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>nochpoll</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>
## Routine

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodev</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>nulldev</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>numtos</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>outb</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>outl</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>outw</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>pci_config_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pci_config_teardown</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>physio</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>pollwakeup</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>proc_ref</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>proc_signal</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>proc_unref</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ptcb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>repinsb</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>repinsd</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>repinsw</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>repoutsb</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>repoutsd</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>repoutsw</td>
<td>Solaris IA DDI</td>
</tr>
<tr>
<td>rmalloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmalloc_wait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmallocmap</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmallocmap_wait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmfree</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmfreemap</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rw_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_downgrade</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_exit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_read_locked</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_tryenter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_tryupgrade</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_abort</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi Alloc_consistent_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_cname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsiDestroy_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dmafree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dmaget</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_errmsg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_free_consistent_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_attach</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>
## Routine Type

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_hba_attach_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_detach</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_fini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_lookup_capstr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_pkt_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_pkt_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_probe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_tran_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_tran_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_ifgetcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_ifsetcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_init_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_log</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_mname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_pktalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_pktfree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_poll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_probe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_resalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_reset</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_reset_notify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_resfree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_rname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_slave</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_sname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_sync_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_transport</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_unprobe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>scsi_unslave</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_p</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_p_sig</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_tryp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_v</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sprintf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>stoi</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strchr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strcmp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strcpy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strlen</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strncmp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strncpy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>swab</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>timeout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>uiomove</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>untimeout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>ureadc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>uwritec</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>va_arg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>va_end</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>va_start</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>vcmn_err</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>vsprintf</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

**SEE ALSO** Intro(9E)
Kernel Functions for Drivers
adjmsg(9F)

NAME
adjmsg – trim bytes from a message

SYNOPSIS
#include <sys/stream.h>

int adjmsg(mblk_t *mp, ssize_t len);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
mp Pointer to the message to be trimmed.
len The number of bytes to be removed.

DESCRIPTION
The adjmsg() function removes bytes from a message. |len| (the absolute value of len) specifies the number of bytes to be removed. The adjmsg() function only trims bytes across message blocks of the same type.

The adjmsg() function finds the maximal leading sequence of message blocks of the same type as that of mp and starts removing bytes either from the head of that sequence or from the tail of that sequence. If len is greater than 0, adjmsg() removes bytes from the start of the first message block in that sequence. If len is less than 0, it removes bytes from the end of the last message block in that sequence.

The adjmsg() function fails if |len| is greater than the number of bytes in the maximal leading sequence it finds.

The adjmsg() function may remove any except the first zero-length message block created during adjusting. It may also remove any zero-length message blocks that occur within the scope of |len|.

RETURN VALUES
The adjmsg() function returns:
1 Successful completion.
0 An error occurred.

CONTEXT
The adjmsg() function can be called from user or interrupt context.

SEE ALSO
STREAMS Programming Guide
allocate a message block

```
#include <sys/stream.h>

mblk_t *allocb(size_t size, uint_t pri);
```

Architecture independent level 1 (DDI/DKI).

`allocb()` tries to allocate a STREAMS message block. Buffer allocation fails only when the system is out of memory. If no buffer is available, the `bufcall(9F)` function can help a module recover from an allocation failure.

A STREAMS message block is composed of three structures. The first structure is a message block (`mblk_t`). See `msgb(9S)`. The `mblk_t` structure points to a data block structure (`dblk_t`). See `datab(9S)`. Together these two structures describe the message type (if applicable) and the size and location of the third structure, the data buffer. The data buffer contains the data for this message block. The allocated data buffer is at least double-word aligned, so it can hold any C data structure.

The fields in the `mblk_t` structure are initialized as follows:

- `b_cont` set to `NULL`
- `b_rptr` points to the beginning of the data buffer
- `b_wptr` points to the beginning of the data buffer
- `b_datap` points to the `dblk_t` structure

The fields in the `dblk_t` structure are initialized as follows:

- `db_base` points to the first byte of the data buffer
- `db_lim` points to the last byte +1 of the buffer
- `db_type` set to `M_DATA`

The following figure identifies the data structure members that are affected when a message block is allocated.
allocb(9F)

<table>
<thead>
<tr>
<th>PARAMETERS</th>
<th>size</th>
<th>The number of bytes in the message block.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pri</td>
<td></td>
<td>Priority of the request (no longer used).</td>
</tr>
</tbody>
</table>

| RETURN VALUES | Upon success, allocb() returns a pointer to the allocated message block of type M_DATA. On failure, allocb() returns a NULL pointer. |

| CONTEXT | allocb() can be called from user or interrupt context. |

<table>
<thead>
<tr>
<th>EXAMPLES</th>
<th><strong>EXAMPLE 1 allocb() Code Sample</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Given a pointer to a queue (q) and an error number (err), the send_error() routine sends an M_ERROR type message to the stream head.</td>
</tr>
<tr>
<td></td>
<td>If a message cannot be allocated, NULL is returned, indicating an allocation failure (line 8). Otherwise, the message type is set to M_ERROR (line 10). Line 11 increments the write pointer (bp-&gt;b_wptr) by the size (one byte) of the data in the message.</td>
</tr>
<tr>
<td></td>
<td>A message must be sent up the read side of the stream to arrive at the stream head. To determine whether q points to a read queue or to a write queue, the q-&gt;q_flag member is tested to see if QREADR is set (line 13). If it is not set, q points to a write queue, and in line 14 the RD(9F) function is used to find the corresponding read queue. In line 15, the putnext(9F) function is used to send the message upstream, returning 1 if successful.</td>
</tr>
<tr>
<td></td>
<td>1 send_error(q,err)</td>
</tr>
<tr>
<td></td>
<td>2 queue_t *q;</td>
</tr>
<tr>
<td></td>
<td>3 unsigned char err;</td>
</tr>
<tr>
<td></td>
<td>4 {</td>
</tr>
<tr>
<td></td>
<td>5 mblk_t *bp;</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7 if (!bp = allocb(1, BPRI_HI)) == NULL) /* allocate msg. block */</td>
</tr>
<tr>
<td></td>
<td>8 return(0);</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10 bp-&gt;b_datap-&gt;db_type = M_ERROR; /* set msg type to M_ERROR */</td>
</tr>
<tr>
<td></td>
<td>11 <em>bp-&gt;b_wptr++ = err; /</em> increment write pointer */</td>
</tr>
</tbody>
</table>
EXAMPLE 1 allocb() Code Sample  (Continued)

```c
12 13 if (!(q->q_flag & QREAD)) /* if not read queue */
14    q = RD(q); /* get read queue */
15    putnext(q,bp); /* send message upstream */
16    return(1);
17 }
```

SEE ALSO  RD(9F), bufcall(9F), esballo(9F), esbbcall(9F), putnext(9F), testb(9F),
          datab(9S), msgb(9S)

Writing Device Drivers

STREAMS Programming Guide

NOTES  The pri argument is no longer used, but is retained for compatibility with existing
        drivers.
anocancel

NAME
anocancel – prevent cancellation of asynchronous I/O request

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int anocancel();

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION
anocancel() should be used by drivers that do not support canceling asynchronous
I/O requests. anocancel() is passed as the driver cancel routine parameter to
aphysio(9F).

RETURN VALUES
anocancel() returns ENXIO.

SEE ALSO
aread(9E), awrite(9E), aphysio(9F)

Writing Device Drivers
aphysio(9F)

NAME
aphysio – perform asynchronous physical I/O

SYNOPSIS
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>
#include <sys/aio_req.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int aphysio(int *strat, struct buf *, int *cancel, struct buf *, dev_t dev, int rw, void *mincnt, struct buf *, struct aio_req *aio_req);

PARAMETERS
strat Pointer to device strategy routine.
cancel Pointer to driver cancel routine. Used to cancel a submitted request. The driver must pass the address of the function anocancel(9F) because cancellation is not supported.
dev The device number.
rw Read/write flag. This is either B_READ when reading from the device or B_WRITE when writing to the device.
mincnt Routine which bounds the maximum transfer unit size.
aio_req Pointer to the aio_req(9S) structure which describes the user I/O request.

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION
aphysio() performs asynchronous I/O operations between the device and the address space described by aio_req->aio_uio.

Prior to the start of the transfer, aphysio() verifies the requested operation is valid. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, strat, is then called one or more times to perform the physical I/O operations. aphysio() does not wait for each transfer to complete, but returns as soon as the necessary requests have been made.

aphysio() calls mincnt to bound the maximum transfer unit size to a sensible default for the device and the system. Drivers which do not provide their own local mincnt routine should call aphysio() with minphys(9F). minphys(9F) is the system mincnt routine. minphys(9F) ensures the transfer size does not exceed any system limits.

If a driver supplies a local mincnt routine, this routine should perform the following actions:

- If bp->b_bcount exceeds a device limit, set bp->b_bcount to a value supported by the device.
- Call minphys(9F) to ensure that the driver does not circumvent additional system limits.

RETURN VALUES
aphysio() returns:
aphysio(9F)

<table>
<thead>
<tr>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Upon success.</td>
</tr>
<tr>
<td>non-zero</td>
<td>Upon failure.</td>
</tr>
</tbody>
</table>

**CONTEXT**
aphysio() can be called from user context only.

**SEE ALSO**
aread(9E), awrite(9E), strategy(9E), anocancel(9F), biodone(9F), biowait(9F), minphys(9F), physio(9F), aio_req(9S), buf(9S), uio(9S)

**Writing Device Drivers**

**WARNINGS**
It is the driver’s responsibility to call biodone(9F) when the transfer is complete.

**BUGS**
Cancellation is not supported in this release. The address of the function anocancel(9F) must be used as the cancel argument.
<table>
<thead>
<tr>
<th>NAME</th>
<th>ASSERT, assert – expression verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/debug.h&gt;</td>
</tr>
<tr>
<td></td>
<td>void ASSERT(EX);</td>
</tr>
<tr>
<td>INTERFACE LEVEL PARAMETERS</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>EX          boolean expression.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>ASSERT() is a macro which checks to see if the expression EX is true. If it is not, then ASSERT() causes an error message to be logged to the console and the system to panic. ASSERT() works only if the preprocessor symbol DEBUG is defined.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>ASSERT() can be used from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td>NAME</td>
<td>ASSERT, assert – expression verification</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/debug.h&gt;</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>void <strong>ASSERT</strong>(EX);</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td></td>
<td><strong>EX</strong>         boolean expression.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td><strong>ASSERT()</strong> is a macro which checks to see if the expression <strong>EX</strong> is true. If it is not, then <strong>ASSERT()</strong> causes an error message to be logged to the console and the system to panic. <strong>ASSERT()</strong> works only if the preprocessor symbol <strong>DEBUG</strong> is defined.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><strong>ASSERT()</strong> can be used from user or interrupt context.</td>
</tr>
</tbody>
</table>

*Writing Device Drivers*
NAME
backq – get pointer to the queue behind the current queue

SYNOPSIS
#include <sys/stream.h>

queue_t *backq(queue_t *cq);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
cq The pointer to the current queue. queue_t is an alias for the queue(9S) structure.

DESCRIPTION
backq() returns a pointer to the queue preceding cq (the current queue). If cq is a read queue, backq() returns a pointer to the queue downstream from cq, unless it is the stream end. If cq is a write queue, backq() returns a pointer to the next queue upstream from cq, unless it is the stream head.

RETURN VALUES
If successful, backq() returns a pointer to the queue preceding the current queue. Otherwise, it returns NULL.

CONTEXT
backq() can be called from user or interrupt context.

SEE ALSO
queue(9S)

Writing Device Drivers
STREAMS Programming Guide
bcanput(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>bcanput – test for flow control in specified priority band</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td>INTERFACE LEVEL PARAMETERS</td>
<td>q Pointer to the message queue.</td>
</tr>
<tr>
<td></td>
<td>pri Message priority.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>bcanput() searches through the stream (starting at q) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message of priority pri in the queue.</td>
</tr>
<tr>
<td></td>
<td>If pri is 0, bcanput() is equivalent to a call with canput(9F).</td>
</tr>
<tr>
<td></td>
<td>canputnext(q) and bcanputnext(q, pri) should always be used in preference to canput(q→q_next) and bcanput(q→q_next, pri) respectively.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>1 If a message of priority pri can be placed on the queue.</td>
</tr>
<tr>
<td></td>
<td>0 If the priority band is full.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>bcanput() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>bcanputnext(9F), canput(9F), canputnext(9F), putbq(9F), putnext(9F)</td>
</tr>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td></td>
<td>STREAMS Programming Guide</td>
</tr>
<tr>
<td>WARNINGS</td>
<td>Drivers are responsible for both testing a queue with bcanput() and refraining from placing a message on the queue if bcanput() fails.</td>
</tr>
</tbody>
</table>
NAME | canputnext, bcanputnext – test for room in next module’s message queue

SYNOPSIS | 
#include <sys/stream.h>

int canputnext(queue_t *q);
int bcanputnext(queue_t *q, unsigned char pri);

INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI).
PARAMETERS | q Pointer to a message queue belonging to the invoking module.
pri Minimum priority level.

DESCRIPTION | The invocation canputnext(q); is an atomic equivalent of the canput(q→q_next); routine. That is, the STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing q through its q_next field and then invoking canput(9F) proceeds without interference from other threads.

bcanputnext(q, pri); is the equivalent of the bcanput(q→q_next, pri); routine.

canputnext(q); and bcanputnext(q, pri); should always be used in preference to canput(q→q_next); and bcanput(q→q_next, pri); respectively.

See canput(9F) and bcanput(9F) for further details.

RETURN VALUES | 1 If the message queue is not full.
0 If the queue is full.

CONTEXT | canputnext() and bcanputnext() can be called from user or interrupt context.

WARNINGS | Drivers are responsible for both testing a queue with canputnext() or bcanputnext() and refraining from placing a message on the queue if the queue is full.

SEE ALSO | bcanput(9F), canput(9F)

Writing Device Drivers

STREAMS Programming Guide
NAME | bcmp – compare two byte arrays  
SYNOPSIS | 
#include <sys/types.h>  
#include <sys/ddi.h>  

    int bcmp(const void *s1, const void *s2, size_t len);  

INTERFACE LEVEL PARAMETERS |  
Architecture independent level 1 (DDI/DKI).  
s1 | Pointer to the first character string.  
s2 | Pointer to the second character string.  
len | Number of bytes to be compared.  

DESCRIPTION | bcmp() compares two byte arrays of length len.  

RETURN VALUES | bcmp() returns 0 if the arrays are identical, or 1 if they are not.  

CONTEXT | bcmp() can be called from user or interrupt context.  
SEE ALSO | strncmp(9F)  

Writing Device Drivers  

NOTES | Unlike strncmp(9F), bcmp() does not terminate when it encounters a null byte.
bcopy – copy data between address locations in the kernel

**SYNOPSIS**

```c
#include <sys/types.h>

void bcopy(const void *from, void *to, size_t bcount);
```

**LEVEL**

Architecture independent level 1 (DDI/DKI).

**PARAMETERS**

- `from` Source address from which the copy is made.
- `to` Destination address to which copy is made.
- `bcount` The number of bytes moved.

**DESCRIPTION**

`bcopy()` copies `bcount` bytes from one kernel address to another. If the input and output addresses overlap, the command executes, but the results may not be as expected.

Note that `bcopy()` should never be used to move data in or out of a user buffer, because it has no provision for handling page faults. The user address space can be swapped out at any time, and `bcopy()` always assumes that there will be no paging faults. If `bcopy()` attempts to access the user buffer when it is swapped out, the system will panic. It is safe to use `bcopy()` to move data within kernel space, since kernel space is never swapped out.

**EXAMPLES**

**EXAMPLE 1** Copying data between address locations in the kernel:

An I/O request is made for data stored in a RAM disk. If the I/O operation is a read request, the data is copied from the RAM disk to a buffer (line 8). If it is a write request, the data is copied from a buffer to the RAM disk (line 15). `bcopy()` is used since both the RAM disk and the buffer are part of the kernel address space.

```c
1 #define RAMDNBLK 1000 /* blocks in the RAM disk */
2 #define RAMDBSIZ 512 /* bytes per block */
3 char ramdblks[RAMDNBLK][RAMDBSIZ]; /* blocks forming RAM */

... 

4
5 if (bp->b_flags & B_READ) /* if read request, copy data */
6    /* from RAM disk data block */
7    /* to system buffer */
8    bcopy(&ramdblks[bp->b_blkno][0], bp->b_un.b_addr,
9       bp->b_count);
10
11 else /* else write request, */
12    /* copy data from a */
13    /* system buffer to RAM disk */
14    /* data block */
15    bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0],
16       bp->b_count);
```

**SEE ALSO**

`copyin(9F)`, `copyout(9F)`
Writing Device Drivers

WARNINGS
The from and to addresses must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.
bioclone – clone another buffer

#include <sys/ddi.h> #include <sys/sunddi.h>

struct buf *bioclone (struct buf *bp, off_t off, size_t len, dev_t dev,
               daddr_t blkno, int (*iodone) (struct buf *), struct buf *bp_mem,
               int sleepflag);

Solaris DDI specific (Solaris DDI).

bp      Pointer to the buf(9S) structure describing the original I/O request.
off     Offset within original I/O request where new I/O request should start.
len     Length of the I/O request.
dev     Device number.
blkno   Block number on device.
iodone  Specific biodone(9F) routine.
bp_mem  Pointer to a buffer structure to be filled in or NULL.
sleepflag Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

bioclone() returns an initialized buffer to perform I/O to a portion of another buffer. The new buffer will be set up to perform I/O to the range within the original I/O request specified by the parameters off and len. An offset 0 starts the new I/O request at the same address as the original request. off + len must not exceed b_bcount, the length of the original request. The device number dev specifies the device to which the buffer is to perform I/O. blkno is the block number on device. It will be assigned to the b_blkno field of the cloned buffer structure. iodone lets the driver identify a specific biodone(9F) routine to be called by the driver when the I/O is complete. bp_mem determines from where the space for the buffer should be allocated. If bp_mem is NULL, bioclone() will allocate a new buffer using getrbuf(9F). If sleepflag is set to KM_SLEEP, the driver may sleep until space is freed up. If sleepflag is set to KM_NOSLEEP, the driver will not sleep. In either case, a pointer to the allocated space is returned or NULL to indicate that no space was available. After the transfer is completed, the buffer has to be freed using freerbuf(9F). If bp_mem is not NULL, it will be used as the space for the buffer structure. The driver has to ensure that bp_mem is initialized properly either using getrbuf(9F) or bioinit(9F).

If the original buffer is mapped into the kernel virtual address space using bp_mapin(9F) before calling bioclone(), a clone buffer will share the kernel mapping of the original buffer. An additional bp_mapin() to get a kernel mapping for the clone buffer is not necessary.
bioclone(9F)

The driver has to ensure that the original buffer is not freed while any of the clone buffers is still performing I/O. The biodone() function has to be called on all clone buffers before it is called on the original buffer.

RETURN VALUES

The bioclone() function returns a pointer to the initialized buffer header, or NULL if no space is available.

CONTEXT

bioclone() can be called from user or interrupt context. Drivers must not allow bioclone() to sleep if called from an interrupt routine.

EXAMPLES

EXAMPLE 1 Using bioclone() for Disk Striping

A device driver can use bioclone() for disk striping. For each disk in the stripe, a clone buffer is created which performs I/O to a portion of the original buffer.

```c
static int
stripe_strategy(struct buf *bp)
{
    ...
    bp_orig = bp;
    bp_1 = bioclone(bp_orig, 0, size_1, dev_1, blkno_1,
        stripe_done, NULL, KM_SLEEP);
    fragment++;
    ...
    bp_n = bioclone(bp_orig, offset_n, size_n, dev_n,
        blkno_n, stripe_done, NULL, KM_SLEEP);
    fragment++;
    /* submit bp_1 ... bp_n to device */
    xxstrategy(bp_n);
    return (0);
}
```

```c
static uint_t
xxintr(caddr_t arg)
{
    ...
    /*
     * get bp of completed subrequest. biodone(9F) will
     * call stripe_done()
     */
    biodone(bp);
    return (0);
}
```

```c
static int
stripe_done(struct buf *bp)
{
    ...
    freerbuf(bp);
    fragment--;
    if (fragment == 0) {
        /* get bp_orig */
        biodone(bp_orig);
}
```
EXAMPLE 1 Using bioclone() for Disk Striping  (Continued)

        return (0);
    
SEE ALSO  biodone(9F), bp_mapin(9F), freerbuf(9F), getrbuf(9F), buf(9S)

Writing Device Drivers
### NAME
biodone – release buffer after buffer I/O transfer and notify blocked threads

### SYNOPSIS
```
#include <sys/types.h>
#include <sys/buf.h>

void biodone(struct buf *bp);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- **bp** Pointer to a buf(9S) structure.

### DESCRIPTION
biodone() notifies blocked processes waiting for the I/O to complete, sets the B_DONE flag in the b_flags field of the buf(9S) structure, and releases the buffer if the I/O is asynchronous. biodone() is called by either the driver interrupt or strategy(9E) routines when a buffer I/O request is complete.

biodone() provides the capability to call a completion routine if bp describes a kernel buffer. The address of the routine is specified in the b_iDONE field of the buf(9S) structure. If such a routine is specified, biodone() calls it and returns without performing any other actions. Otherwise, it performs the steps above.

### CONTEXT
biodone() can be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1**

Generally, the first validation test performed by any block device strategy(9E) routine is a check for an end-of-file (EOF) condition. The strategy(9E) routine is responsible for determining an EOF condition when the device is accessed directly. If a read(2) request is made for one block beyond the limits of the device (line 10), it will report an EOF condition. Otherwise, if the request is outside the limits of the device, the routine will report an error condition. In either case, report the I/O operation as complete (line 27).

```c
1 #define RAMDNBLK 1000 /* Number of blocks in RAM disk */
2 #define RAMDSIZ 512 /* Number of bytes per block */
3 char ramdblks[RAMDNBLK][RAMDSIZ]; /* Array containing RAM disk */
4
5 static int
6 ramdstrategy(struct buf *bp)
7 {
8     daddr_t blkno = bp->b_blkno; /* get block number */
9
10     if ((blkno < 0) || (blkno >= RAMDNBLK)) {
11         /*
12         * If requested block is outside RAM disk
13         * limits, test for EOF which could result
14         * from a direct (physio) request.
15         */
16         if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
17             /*
18             * If read is for block beyond RAM disk
19             * limits, mark EOF condition.
20             */
21             bp->b_resid = bp->b_bcount; /* compute return value */
```
EXAMPLE 1 (Continued)

```c
} else { /* I/O attempt is beyond */
    bp->b_error = ENXIO; /* limits of RAM disk */
    bp->b_flags |= B_ERROR; /* return error */
}
```

```c
biodone(bp); /* mark I/O complete (B_DONE) */
/*
 * Wake any processes awaiting this I/O
 * or release buffer for asynchronous
 * (B_ASYNC) request.
 */
return (0);
```

SEE ALSO `biodone(9F)`

WARNINGS After calling `biodone()`, `bp` is no longer available to be referred to by the driver. If the driver makes any reference to `bp` after calling `biodone()`, a panic may result.

NOTES Drivers that use the `b_iiodone` field of the `buf(9S)` structure to specify a substitute completion routine should save the value of `b_iiodone` before changing it, and then restore the old value before calling `biodone()` to release the buffer.
bioerror(9F)

NAME    bioerror – indicate error in buffer header

SYNOPSIS
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>

void bioerror(struct buf *bp, int error);

INTERFACE LEVEL    Solaris DDI specific (Solaris DDI)
PARAMETERS
bp Pointer to the buf(9S) structure describing the transfer.
error Error number to be set, or zero to clear an error indication.

DESCRIPTION
If error is non-zero, bioerror() indicates an error has occurred in the buf(9S) structure. A subsequent call to geterror(9F) will return error.

If error is 0, the error indication is cleared and a subsequent call to geterror(9F) will return 0.

CONTEXT    bioerror() can be called from any context.

SEE ALSO    strategy(9E), geterror(9F), getrbuf(9F), buf(9S)
NAME

biofini – uninitialize a buffer structure

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void biofini(struct buf *bp);
```

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

bp Pointer to the buffer header structure.

DESCRIPTION

The biofini() function uninitializes a buf(9S) structure. If a buffer structure has been allocated and initialized using kmem_alloc(9F) and bioinit(9F) it needs to be uninitialized using biofini() before calling kmem_free(9F). It is not necessary to call biofini() before freeing a buffer structure using freerbuf(9F) because freerbuf() will call biofini() directly.

CONTEXT

The biofini() function can be called from any context.

EXAMPLES

EXAMPLE 1 Using biofini()

```c
struct buf *bp = kmem_alloc(biosize(), KM_SLEEP);
bioinit(bp);
/* use buffer */
biofini(bp);
kmem_free(bp, biosize());
```

SEE ALSO

bioinit(9F), bioreset(9F), biosize(9F), freerbuf(9F), kmem_alloc(9F), kmem_free(9F), buf(9S)

Writing Device Drivers
**NAME**
bioinit – initialize a buffer structure

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void bioinit(struct buf *bp);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- `bp` Pointer to the buffer header structure.

**DESCRIPTION**
The `bioinit()` function initializes a `buf(9S)` structure. A buffer structure contains state information which has to be initialized if the memory for the buffer was allocated using `kmem_alloc(9F)`. This is not necessary for a buffer allocated using `getrbuf(9F)` because `getrbuf()` will call `bioinit()` directly.

**CONTEXT**
The `bioinit()` function can be called from any context.

**EXAMPLES**
**EXAMPLE 1 Using bioinit()**

```c
struct buf *bp = kmem_alloc(biosize(), KM_SLEEP);
bioinit(bp);
/* use buffer */
```

**SEE ALSO**
`biofini(9F)`, `bioreset(9F)`, `biosize(9F)`, `getrbuf(9F)`, `kmem_alloc(9F)`, `buf(9S)`

*Writing Device Drivers*
NAME

biomodified – check if a buffer is modified

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int biomodified(struct buf *bp);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

bp Pointer to the buffer header structure.

DESCRIPTION

The `biomodified()` function returns status to indicate if the buffer is modified. The `biomodified()` function is only supported for paged I/O request, that is the `B_PAGEIO` flag must be set in the `b_flags` field of the `buf(9S)` structure. The `biomodified()` function will check the memory pages associated with this buffer whether the Virtual Memory system’s modification bit is set. If at least one of these pages is modified, the buffer is indicated as modified. A filesystem will mark the pages unmodified when it writes the pages to the backing store. The `biomodified()` function can be used to detect any modifications to the memory pages while I/O is in progress.

A device driver can use `biomodified()` for disk mirroring. An application is allowed to `mmap` a file which can reside on a disk which is mirrored by multiple submirrors. If the file system writes the file to the backing store, it is written to all submirrors in parallel. It must be ensured that the copies on all submirrors are identical. The `biomodified()` function can be used in the device driver to detect any modifications to the buffer by the user program during the time the buffer is written to multiple submirrors.

RETURN VALUES

The `biomodified()` function returns the following values:

1 Buffer is modified.

0 Buffer is not modified.

-1 Buffer is not used for paged I/O request.

CONTEXT

`biomodified()` can be called from any context.

SEE ALSO

`bp_mapin(9F), buf(9S)`

*Writing Device Drivers*
**NAME**
bioreset – reuse a private buffer header after I/O is complete

**SYNOPSIS**
```c
#include <sys/buf.h>
#include <sys/ddi.h>

void bioreset (struct buf *bp);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI)

**PARAMETERS**
bp Pointer to the buf(9S) structure.

**DESCRIPTION**
bioreset() is used by drivers that allocate private buffers with getrbuf(9F) or kmem_alloc(9F) and want to reuse them in multiple transfers before freeing them with freerbuf(9F) or kmem_free(9F). bioreset() resets the buffer header to the state it had when initially allocated by getrbuf() or initialized by bioinit(9F).

**CONTEXT**
bioreset() can be called from any context.

**SEE ALSO**
strategy(9E), bioinit(9F), biofini(9F), freerbuf(9F), getrbuf(9F), kmem_alloc(9F), kmem_free(9F), buf(9S)

**NOTES**
bp must not describe a transfer in progress.
The `biosize()` function returns the size in bytes of the `buf(9S)` structure. The `biosize()` function is used by drivers in combination with `kmem_alloc(9F)` and `bioinit(9F)` to allocate buffer structures embedded in other data structures.

The `biosize()` function can be called from any context.

**SEE ALSO**

`biofini(9F), bioinit(9F), getrbuf(9F), kmem_alloc(9F), buf(9S)`

*Writing Device Drivers*
biowait(9F)

NAME
biowait – suspend processes pending completion of block I/O

SYNOPSIS
#include <sys/types.h>
#include <sys/buf.h>

int biowait(struct buf *bp);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
bp Pointer to the buf structure describing the transfer.

DESCRIPTION
Drivers allocating their own buf structures with getrbuf(9F) can use the
biowait() function to suspend the current thread and wait for completion of the
transfer.

Drivers must call biodone(9F) when the transfer is complete to notify the thread
blocked by biowait(). biodone() is usually called in the interrupt routine.

RETURN VALUES
0 Upon success
non-zero Upon I/O failure. biowait() calls geterror(9F) to retrieve the
error number which it returns.

CONTEXT
biowait() can be called from user context only.

SEE ALSO
biodone(9F), geterror(9F), getrbuf(9F), buf(9S)

Writing Device Drivers
### bp_mapin(9F)

#### NAME
bp_mapin – allocate virtual address space

#### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/buf.h>

void bp_mapin(struct buf *bp);
```

#### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

#### PARAMETERS
- **bp**: Pointer to the buffer header structure.

#### DESCRIPTION
`bp_mapin()` is used to map virtual address space to a page list maintained by the buffer header during a paged-I/O request. `bp_mapin()` allocates system virtual address space, maps that space to the page list, and returns the starting address of the space in the `bp->b_un.b_addr` field of the `buf(9S)` structure. Virtual address space is then deallocated using the `bp_mapout(9F)` function.

If a null page list is encountered, `bp_mapin()` returns without allocating space and no mapping is performed.

#### CONTEXT
`bp_mapin()` can be called from user and kernel contexts.

#### SEE ALSO
- `bp_mapout(9F)`, `buf(9S)`

*Writing Device Drivers*
<table>
<thead>
<tr>
<th>NAME</th>
<th>bp_mapout – deallocate virtual address space</th>
</tr>
</thead>
</table>
| SYNOPSIS   | `#include <sys/types.h>`
|            | `#include <sys/buf.h>`
|            | `void bp_mapout(struct buf *bp);` |
| INTERFACE  | Architecture independent level 1 (DDI/DKI). |
| LEVEL      | |
| PARAMETERS | bp Pointer to the buffer header structure. |
| DESCRIPTION| `bp_mapout()` deallocates system virtual address space allocated by a previous call to `bp_mapin(9F)`. `bp_mapout()` should only be called on buffers which have been allocated and are owned by the device driver. It must not be called on buffers passed to the driver through the `strategy(9E)` entry point (for example a filesystem). Because `bp_mapin(9F)` does not keep a reference count, `bp_mapout()` will wipe out any kernel mapping that a layer above the device driver might rely on. |
| CONTEXT    | `bp_mapout()` can be called from user context only. |
| SEE ALSO   | `strategy(9E), bp_mapin(9F), buf(9S)` |

*Writing Device Drivers*
### NAME
btop – convert size in bytes to size in pages (round down)

### SYNOPSIS
```
#include <sys/ddi.h>

unsigned long btop(unsigned long numbytes);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numbytes</td>
<td>Number of bytes.</td>
</tr>
</tbody>
</table>

### DESCRIPTION
`btop()` returns the number of memory pages that are contained in the specified number of bytes, with downward rounding in the case that the byte count is not a page multiple. For example, if the page size is 2048, then `btop(4096)` returns 2, and `btop(4097)` returns 2 as well. `btop(0)` returns 0.

### RETURN VALUES
The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

### CONTEXT
`btop()` can be called from user or interrupt context.

### SEE ALSO
`btopr(9F), ddi_btop(9F), ptob(9F)`

*Writing Device Drivers*
btopr(9F)

NAME  btopr – convert size in bytes to size in pages (round up)

SYNOPSIS  
#include <sys/ddi.h>

unsigned long btopr(unsigned long numbytes);

INTERFACE LEVEL
PARAMETERS  
numbytes Number of bytes.

DESCRIPTION  btopr() returns the number of memory pages contained in the specified number of bytes memory, rounded up to the next whole page. For example, if the page size is 2048, then btopr(4096) returns 2, and btopr(4097) returns 3.

RETURN VALUES  The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

CONTEXT  btopr() can be called from user or interrupt context.

SEE ALSO  btop(9F), ddi_btopr(9F), ptob(9F)

Writing Device Drivers
bufcall(9F)

bufcall – call a function when a buffer becomes available

**SYNOPSIS**

```
#include <sys/types.h>
#include <sys/stream.h>

bufcall_id_t bufcall(size_t size, uint_t pri, void *func, void *arg);
```

**INTERFACE LEVEL**

Architecture independent level 1 (DDI/DKI).

**PARAMETERS**

- `size` Number of bytes required for the buffer.
- `pri` Priority of the `allocb(9F)` allocation request (not used).
- `func` Function or driver routine to be called when a buffer becomes available.
- `arg` Argument to the function to be called when a buffer becomes available.

**DESCRIPTION**

`bufcall()` serves as a `timeout(9F)` call of indeterminate length. When a buffer allocation request fails, `bufcall()` can be used to schedule the routine `func`, to be called with the argument `arg` when a buffer becomes available. `func` may call `allocb()` or it may do something else.

**RETURN VALUES**

If successful, `bufcall()` returns a `bufcall` ID that can be used in a call to `unbufcall()` to cancel the request. If the `bufcall()` scheduling fails, `func` is never called and 0 is returned.

**CONTEXT**

`bufcall()` can be called from user or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Calling a function when a buffer becomes available:

The purpose of this `srv(9E)` service routine is to add a header to all `M_DATA` messages. Service routines must process all messages on their queues before returning, or arrange to be rescheduled.

While there are messages to be processed (line 13), check to see if it is a high priority message or a normal priority message that can be sent on (line 14). Normal priority message that cannot be sent are put back on the message queue (line 34). If the message was a high priority one, or if it was normal priority and `canputnext(9F)` succeeded, then send all but `M_DATA` messages to the next module with `putnext(9F)` (line 16).

For `M_DATA` messages, try to allocate a buffer large enough to hold the header (line 18). If no such buffer is available, the service routine must be rescheduled for a time when a buffer is available. The original message is put back on the queue (line 20) and `bufcall` (line 21) is used to attempt the rescheduling. It will succeed if the rescheduling succeeds, indicating that `qenable` will be called subsequently with the argument `q` once a buffer of the specified size (`sizeof(struct hdr)`) becomes available. If it does, `qenable(9F)` will put `q` on the list of queues to have their service routines called. If `bufcall()` fails, `timeout(9F)` (line 22) is used to try again in about a half second.
EXAMPLE 1 Calling a function when a buffer becomes available: (Continued)

If the buffer allocation was successful, initialize the header (lines 25–28), make the message type M_PROTO (line 29), link the M_DATA message to it (line 30), and pass it on (line 31).

Note that this example ignores the bookkeeping needed to handle bufcall() and timeout(9F) cancellation for ones that are still outstanding at close time.

```c
1 struct hdr {
2     unsigned int h_size;
3     int h_version;
4 }
5
6 void xxxsrv(q) 
7     queue_t *q;
8 {
9     mblk_t *bp;
10     mblk_t *mp;
11     struct hdr *hp;
12
13     while ((mp = getq(q)) != NULL) { /* get next message */
14         if (mp->b_datap->db_type >= QPCTL || /* if high priority */
15             canputnext(q)) { /* normal & can be passed */
16             if (mp->b_datap->db_type != M_DATA) /* send all but M_DATA */
17                 putnext(q, mp);
18             else {
19                 bp = allocb(sizeof(struct hdr), BPRI_LO);
20                 if (bp == NULL) { /* if unsuccessful */
21                     putbq(q, mp); /* put it back */
22                     if (!bufcall(sizeof(struct hdr), BPRI_LO,
23                                     qenable, q, drv_usectohz(500000)))
24                         return (0);
25                 } else { /* normal priority, canputnext failed */
26                     putbq(q, mp); /* put back on the message queue */
27                     return (0);
28                 }
29             }
30         } else { /* normal priority, canputnext failed */
31             putbq(q, mp); /* put back on the message queue */
32             return (0);
33         }
34     }
35 }
```

SEE ALSO srv(9E), allocb(9F), canputnext(9F), esballoc(9F), esbbcall(9F), putnext(9F), qenable(9F), testb(9F), timeout(9F), unbufcall(9F)
Even when `func` is called by `bufcall()`, `allocb(9F)` can fail if another module or driver had allocated the memory before `func` was able to call `allocb(9F).`
**NAME**  
bzero – clear memory for a given number of bytes

**SYNOPSIS**  
```c
#include <sys/types.h>
#include <sys/ddi.h>

void bzero(void *addr, size_t bytes);
```

**INTERFACE LEVEL**  
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `addr` Starting virtual address of memory to be cleared.
- `bytes` The number of bytes to clear starting at `addr`.

**DESCRIPTION**  
bzero() clears a contiguous portion of memory by filling it with zeros.

**CONTEXT**
bzero() can be called from user or interrupt context.

**SEE ALSO**
`bcopy(9F), clrbuf(9F), kmem_zalloc(9F)`

**Writing Device Drivers**

**WARNINGS**
The address range specified must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.
canput - test for room in a message queue

#include <sys/stream.h>

int canput(queue_t *q);

Architecture independent level 1 (DDI/DKI).

q Pointer to the message queue.

canput() searches through the stream (starting at q) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message in the queue.

canputnext(q) and bcanputnext(q, pri) should always be used in preference to canput(q→q_next) and bcanput(q→q_next, pri) respectively.

1 If the message queue is not full.

0 If the queue is full.

canput() can be called from user or interrupt context.

bcanput, bcanputnext, canputnext, putbq, putnext

Writing Device Drivers

STREAMS Programming Guide

Drivers are responsible for both testing a queue with canput() and refraining from placing a message on the queue if canput() fails.
canputnext(9F)

NAME

canputnext, bcanputnext – test for room in next module’s message queue

SYNOPSIS

#include <sys/stream.h>

int canputnext(queue_t *q);
int bcanputnext(queue_t *q, unsigned char pri);

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

PARAMETERS

q Pointer to a message queue belonging to the invoking module.

pri Minimum priority level.

DESCRIPTION

The invocation canputnext(q) is an atomic equivalent of the canput(q->q_next) routine. That is, the STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing q through its q_next field and then invoking canput(9F) proceeds without interference from other threads.

bcanputnext(q, pri) is the equivalent of the bcanput(q->q_next, pri) routine.

canputnext(q) and bcanputnext(q, pri) should always be used in preference to canput(q->q_next); and bcanput(q->q_next, pri); respectively.

See canput(9F) and bcanput(9F) for further details.

RETURN VALUES

1 If the message queue is not full.
0 If the queue is full.

CONTEXT
canputnext() and bcanputnext() can be called from user or interrupt context.

WARNINGS

Drivers are responsible for both testing a queue with canputnext() or bcanputnext() and refraining from placing a message on the queue if the queue is full.

SEE ALSO

bcanput(9F), canput(9F)

Writing Device Drivers

STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>clrbuf – erase the contents of a buffer</th>
</tr>
</thead>
</table>
| SYNOPSIS | ```
#include <sys/types.h>
#include <sys/buf.h>

void clrbuf(struct buf *bp);
``` |
| INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI). |
| PARAMETERS | `bp` Pointer to the `buf(9S)` structure. |
| DESCRIPTION | `clrbuf()` zeros a buffer and sets the `b_resid` member of the `buf(9S)` structure to 0. Zeros are placed in the buffer starting at `bp->b_un.b_addr` for a length of `bp->b_bcount` bytes. `b_un.b_addr` and `b_bcount` are members of the `buf(9S)` data structure. |
| CONTEXT | `clrbuf()` can be called from user or interrupt context. |
| SEE ALSO | `getrbuf(9F), buf(9S)` |

*Writing Device Drivers*
NAME | cmn_err, vcmn_err – display an error message or panic the system

SYNOPSIS

```c
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

void cmn_err(int level, char *format...);

#include <sys/varargs.h>

void vcmn_err(int level, char *format, va_list ap);
```

INTERFACE LEVEL

cmn_err()

Architecture independent level 1 (DDI/DKI).

- `level` A constant indicating the severity of the error condition.
- `format` The message to be displayed.

vcmn_err()

- `vcmn_err()` takes `level` and `format` as described for `cmn_err()`, but its third argument is different:
  - `ap` The variable argument list passed to the function.

cmn_err()

- `cmn_err()` displays a specified message on the console. `cmn_err()` can also panic the system. When the system panics, it attempts to save recent changes to data, display a “panic message” on the console, attempt to write a core file, and halt system processing. See the CE_PANIC level below.

- `level` is a constant indicating the severity of the error condition. The four severity levels are:

  - **CE_CONT** Used to continue another message or to display an informative message not associated with an error. Note that multiple CE_CONT messages without a newline may or may not appear on the system console or in the system log as a single line message. A single line message may be produced by constructing the message with `sprintf(9F)` or `vsprintf(9F)` before calling `cmn_err()`.

  - **CE_NOTE** Used to display a message preceded with NOTICE. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a message saying that a sector on a disk needs to be accessed repeatedly before it can be accessed correctly might be noteworthy.

  - **CE_WARN** Used to display a message preceded with WARNING. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.

  - **CE_PANIC** Used to display a message preceded with “panic”, and to panic the system. Drivers should specify this level only under the most severe conditions or when debugging a driver. A valid use of this
level is when the system cannot continue to function. If the error is recoverable, or not essential to continued system operation, do not panic the system.

format is the message to be displayed. It is a character string which may contain plain characters and conversion specifications. By default, the message is sent both to the system console and to the system log.

Each conversion specification in format is introduced by the % character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (ll) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

d,D,o,O,x,X,u  The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X), or unsigned decimal (u), respectively, and displayed. The letters abcdef are used for x and X conversion.

c  The character value of the argument is displayed.

b  The %b conversion specification allows bit values to be displayed meaningfully. Each %b takes an integer value and a format string from the argument list. The first character of the format string should be the output base encoded as a control character. This base is used to display the integer argument. The remaining groups of characters in the format string consist of a bit number (between 1 and 32, also encoded as a control character) and the next characters (up to the next control character or '\0') give the name of the bit field. The string corresponding to the bit fields set in the integer argument is displayed after the numerical value. See EXAMPLE section.

p  The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void * when using the %p format specifier.

s  The argument is taken to be a string (character pointer), and characters from the string are displayed until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.

%  Copy a %; no argument is converted.
The first character in format affects where the message will be written:

! The message goes only to the system log.

^ The message goes only to the console.

? If level is also CE_CONT, the message is always sent to the system log, but is only written to the console when the system has been booted in verbose mode. See kernel(1M). If neither condition is met, the ‘?’ character has no effect and is simply ignored.

Refer to syslogd(1M) to determine where the system log is written.

cmn_err() appends a \n to each format, except when level is CE_CONT.

vcmn_err() is identical to cmn_err() except that its last argument, ap, is a pointer to a variable list of arguments. ap contains the list of arguments used by the conversion specifications in format. ap must be initialized by calling va_start(9F). va_end(9F) is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by va_start(9F) and va_end(9F), are possible.

RETURN VALUES

None. However, if an unknown level is passed to cmn_err(), the following panic error message is displayed:

panic: unknown level in cmn_err (level=level, msg=format)

CONTEXT

cmn_err() can be called from user, kernel, interrupt, or high-level interrupt context.

EXAMPLES

EXAMPLE 1 Using cmn_err()

This first example shows how cmn_err() can record tracing and debugging information only in the system log (lines 17); display problems with a device only on the system console (line 23); or display problems with the device on both the system console and in the system log (line 28).

```c
struct reg {
    uchar_t data;
    uchar_t csr;
};

struct xxstate {
    ...
    dev_info_t *dip;
    struct reg *regp;
    ...
};

dev_t dev;
struct xxstate *xsp;

#ifdef DEBUG /* in debugging mode, log function call */
cmn_err(CE_CONT, "%!%s%d: xxopen function called.",
      ddi_binding_name(xsp->dip), getminor(dev));
```

va_start(9F) is used to clean up and must be called after each traversal of the list.

Multiple traversals of the argument list, each bracketed by va_start(9F) and va_end(9F), are possible.

RETURN VALUES

None. However, if an unknown level is passed to cmn_err(), the following panic error message is displayed:

panic: unknown level in cmn_err (level=level, msg=format)

CONTEXT

cmn_err() can be called from user, kernel, interrupt, or high-level interrupt context.

EXAMPLES

EXAMPLE 1 Using cmn_err()

This first example shows how cmn_err() can record tracing and debugging information only in the system log (lines 17); display problems with a device only on the system console (line 23); or display problems with the device on both the system console and in the system log (line 28).

```c
struct reg {
    uchar_t data;
    uchar_t csr;
};

struct xxstate {
    ...
    dev_info_t *dip;
    struct reg *regp;
    ...
};

dev_t dev;
struct xxstate *xsp;

#ifdef DEBUG /* in debugging mode, log function call */
cmn_err(CE_CONT, "%!%s%d: xxopen function called.",
      ddi_binding_name(xsp->dip), getminor(dev));
```
EXAMPLE 1 Using cmn_err() (Continued)

19 #endif /* end DEBUG */
20 ...
21 /* display device power failure on system console */
22 if ((xsp->regp->csr & POWER) == OFF)
23 cmn_err(CE_NOTE, "^OFF.",
24 ddi_binding_name(xsp->dip), getminor(dev));
25 ...
26 /* display warning if device has bad VTOC */
27 if (xsp->regp->csr & BADVTOC)
28 cmn_err(CE_WARN, "%s%d: xxopen: Bad VTOC.",
29 ddi_binding_name(xsp->dip), getminor(dev));

EXAMPLE 2 Using the %b conversion specification

This example shows how to use the %b conversion specification. Because of the
leading '?' character in the format string, this message will always be logged, but it
will only be displayed when the kernel is booted in verbose mode.

    cmn_err(CE_CONT, "?reg=0x%b\n", regval, \"Intr\2Err\1Enable\")

EXAMPLE 3 Using regval

When regval is set to (decimal) 13, the following message would be displayed:

    reg=0xd\nIntr,,Enable>

EXAMPLE 4 Error Routine

The third example is an error reporting routine which accepts a variable number of
arguments and displays a single line error message both in the system log and on the
system console. Note the use of vsprintf() to construct the error message before
calling cmn_err().

    #include <sys/varargs.h>
    #include <sys/ddi.h>
    #include <sys/sunddi.h>
    #define MAX_MSG 256;

    void
    xxerror(dev_info_t *dip, int level, const char *fmt,...)
    {
        va_list ap;
        int instance;
        char buf[MAX_MSG], *name;

        instance = ddi_get_instance(dip);
        name = ddi_binding_name(dip);

        /* format buf using fmt and arguments contained in ap */

        va_start(ap, fmt);
        va_start(ap, fmt);
EXEMPLARY

```c
EXAMPLE 4 Error Routine  (Continued)

vsprintf(buf, fmt, ap);
va_end(ap);

/* pass formatted string to cmn_err(9F) */

// cmn_err(9F)

SEE ALSO
dmesg(1M), kernel(1M), printf(3C), ddi_binding_name(9F), sprintf(9F),
va_arg(9F), va_end(9F), va_start(9F), vsprintf(9F)

Writing Device Drivers

WARNINGS

Messages of arbitrary length can be generated using cmn_err(), but if the call
to cmn_err() is made from high-level interrupt context and insufficient memory is
available to create a buffer of the specified size, the message will be truncated to
LOG_MSGSIZE bytes (see <sys/log.h>). For this reason, callers of cmn_err() that
require complete and accurate message generation should post down from high-level
interrupt context before calling cmn_err().

NOTES

// cmn_err(9F)
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig,
cv_timedwait, cv_timedwait_sig — condition variable routines

#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);

void cv_destroy(kcondvar_t *cvp);

void cv_wait(kcondvar_t *cvp, kmutex_t *mp);

void cv_signal(kcondvar_t *cvp);

void cv_broadcast(kcondvar_t *cvp);

int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);

clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE LEVEL SPECIFICATIONS

Solaris DDI specific (Solaris DDI).

PARAMETERS

cvp A pointer to an abstract data type kcondvar_t.

mp A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.

name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)

type The constant CV_DRIVER.

arg A type-specific argument, drivers should pass arg as NULL.

timeout A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.

DESCRIPTION

Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with cv_signal() or
cv_broadcast(), and then release the mutex.

cv_wait() suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal() signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling cv_broadcast(). You must acquire the mutex
passed into cv_wait() before calling cv_signal() or cv_broadcast().

The function cv_wait_sig() is similar to cv_wait() but returns 0 if a signal (for
example, by kill(2)) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function cv_timedwait() is similar to cv_wait(), except that it returns -1
without the condition being signaled after the timeout time has been reached.

The function cv_timedwait_sig() is similar to cv_timedwait() and
cv_wait_sig(), except that it returns -1 without the condition being signaled after
the timeout time has been reached, or 0 if a signal (for example, by kill(2)) is sent to
the thread.

For both cv_timedwait() and cv_timedwait_sig(), time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
ddi_get_lbolt(9F).

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>For cv_wait_sig() and cv_timedwaitSig() indicates that the condition was not necessarily signaled and the function returned because a signal (as in kill(2)) was pending.</td>
</tr>
<tr>
<td>-1</td>
<td>For cv_timedwait() and cv_timedwaitSig() indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>For cv_wait_sig(), cv_timedwait() or cv_timedwaitSig() indicates that the condition was met and the function returned due to a call to cv_signal() or cv_broadcast(), or due to a premature wakeup (see NOTES).</td>
</tr>
</tbody>
</table>

**CONTEXT** These functions can be called from user, kernel or interrupt context. In most cases,
however, cv_wait(), cv_timedwait(), cv_wait_sig(), and
cv_timedwaitSig() should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If cv_wait(), cv_timedwait(), cv_waitSig(), or cv_timedwaitSig() are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

EXAMPLE 1 Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

EXAMPLE 2 Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &~ UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

NOTES It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

SEE ALSO `kill(2)`, `ddi_get_lbolt(9F)`, `mutex(9F)`, `mutex_init(9F)`

Writing Device Drivers
copyb(9F)

NAME  
copyb – copy a message block

SYNOPSIS  
#include <sys/stream.h>

mblk_t *copyb(mblk_t *bp);

INTERFACE
LEVEL
PARAMETERS  
bp Pointer to the message block from which data is copied.

DESCRIPTION  
copyb() allocates a new message block, and copies into it the data from the block
that bp denotes. The new block will be at least as large as the block being copied.
copyb() uses the b_rptr and b_wptr members of bp to determine how many bytes
to copy.

RETURN VALUES  
If successful, copyb() returns a pointer to the newly allocated message block
containing the copied data. Otherwise, it returns a NULL pointer.

CONTEXT  
copyb() can be called from user or interrupt context.

EXAMPLES  
EXAMPLE 1: Using copyb

For each message in the list, test to see if the downstream queue is full with the
canputnext(9F) function (line 21). If it is not full, use copyb to copy a header
message block, and dupmsg(9F) to duplicate the data to be retransmitted. If either
operation fails, reschedule a timeout at the next valid interval.

Update the new header block with the correct destination address (line 34), link the
message to it (line 35), and send it downstream (line 36). At the end of the list,
reschedule this routine.

1 struct retrans {
2    mblk_t *r_mp;
3    int r_address;
4    queue_t *r_outq;
5    struct retrans *r_next;
6 };
7
8 struct protoheader {
...  
9    int h_address;
...  
10 };
11
12 mblk_t *header;
13
14 void
15 retransmit(struct retrans *ret)
16 {
17    mblk_t *bp, *mp;
18    struct protoheader *php;
19
20    while (ret) {
21        if (!canputnext(ret->r_outq)) { /* no room */
EXAMPLE 1: Using copyb  

(Continued)

```c
22     ret = ret->r_next;
23     continue;
24 }
25    bp = copyb(header); /* copy header msg. block */
26    if (bp == NULL)
27       break;
28    mp = dupmsg(ret->r_mp); /* duplicate data */
29    if (mp == NULL) {
30       freeb(bp); /* free the block */
31       break;
32    }
33    php = (struct protoheader *)bp->b_rptr;
34    php->h_address = ret->r_address; /* new header */
35    bp->bp_cont = mp; /* link the message */
36    putnext(ret->r_outq, bp); /* send downstream */
37    ret = ret->r_next;
38 }
39 /* reschedule */
40    (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
41 }
```

SEE ALSO

allocb(9F), canputnext(9F), dupmsg(9F)

Writing Device Drivers

STREAMS Programming Guide
copyin(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>copyin – copy data from a user program to a driver buffer</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/types.h>  
|         | #include <sys/ddi.h>  
|         | int copyin(const void *userbuf, void *driverbuf, size_t cn); |
| INTERFACE | This interface is obsolete. ddi_copyin(9F) should be used instead. |
| LEVEL   | PARAMETERS |
|        | userbuf User program source address from which data is transferred. |
|        | driverbuf Driver destination address to which data is transferred. |
|        | cn Number of bytes transferred. |
| DESCRIPTION | copyin() copies data from a user program source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address. Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move according to address alignment. |
| RETURN VALUES | Under normal conditions, a 0 is returned indicating a successful copy. Otherwise, a −1 is returned if one of the following occurs: |
|        | • Paging fault; the driver tried to access a page of memory for which it did not have read or write access. |
|        | • Invalid user address, such as a user area or stack area. |
|        | • Invalid address that would have resulted in data being copied into the user block. |
|        | • Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred. |
| CONTEXT | copyin() can be called from user context only. |
| EXAMPLES | EXAMPLE 1 An ioctl() Routine |
|        | A driver ioctl(9E) routine (line 10) can be used to get or set device attributes or registers. In the XX_GETREGS condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned. |
|        | 1 struct device { /* layout of physical device registers */  
|        | 2 int control; /* physical device control word */  
|        | 3 int status; /* physical device status word */  
|        | 4 short recv_char; /* receive character from device */  
|        | 5 short xmit_char; /* transmit character to device */  
|        | 6 };  
|        | 7  
|        | 8 extern struct device xx_addr[]; /* phys. device regs. location */
EXAMPLE 1 An ioctl() Routine (Continued)

```c
xx_ioctl(dev_t dev, int cmd, int arg, int mode,
    cred_t *cred_p, int *rval_p)
{
    register struct device *rp = &xx_addr[getminor(dev) >> 4];
    switch (cmd) {
    case XX_GETREGS: /* copy device regs. to user program */
        if (copyin(arg, rp, sizeof(struct device)))
            return(EFAULT);
        break;
    ...
    }
```

ATTRIBUTES See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ioctl(9E), bcopy(9F), copyout(9F), ddi_copyin(9F), ddi_copyout(9F), uiomove(9F).

Writing Device Drivers

NOTES Driver writers who intend to support layered ioctls in their ioctl(9E) routines should use ddi_copyin(9F) instead.

Driver defined locks should not be held across calls to this function.

copyin() should not be used from a streams driver. See M_COPYIN and M_COPYOUT in STREAMS Programming Guide.
copymsg(9F)

NAME

copymsg – copy a message

SYNOPSIS

#include <sys/stream.h>

mblk_t *copymsg(mblk_t *mp);

INTERFACE

Architecture independent level 1 (DDI/DKI).

PARAMETERS

mp Pointer to the message to be copied.

DESCRIPTION

copymsg() forms a new message by allocating new message blocks, and copying the contents of the message referred to by mp (using the copyb(9F) function). It returns a pointer to the new message.

RETURN VALUES

If the copy is successful, copymsg() returns a pointer to the new message. Otherwise, it returns a NULL pointer.

CONTEXT

copymsg() can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1: Using copymsg

The routine lctouc() converts all the lowercase ASCII characters in the message to uppercase. If the reference count is greater than one (line 8), then the message is shared, and must be copied before changing the contents of the data buffer. If the call to the copymsg() function fails (line 9), return NULL (line 10), otherwise, free the original message (line 11). If the reference count was equal to 1, the message can be modified. For each character (line 16) in each message block (line 15), if it is a lowercase letter, convert it to an uppercase letter (line 18). A pointer to the converted message is returned (line 21).

```
1 mblk_t *lctouc(mp)
2 mblk_t *mp;
3 {
4   mblk_t *cmp;
5   mblk_t *tmp;
6   unsigned char *cp;
7 |
8   if (mp->b_datap->db_ref > 1) {
9     if ((cmp = copymsg(mp)) == NULL)
10        return (NULL);
11     freemsg(mp);
12   } else {
13     cmp = mp;
14   }
15   for (tmp = cmp; tmp; tmp = tmp->b_cont) {
16     for (cp = tmp->b_rptr; cp < tmp->b_wptr; cp++) {
17       if ((*cp <= 'z') && (*cp >= 'a'))
18          *cp -= 0x20;
19     }
20   }
21   return(cmp);
22 }
```
EXAMPLE 1: Using copymsg (Continued)

SEE ALSO allocb(9F), copyb(9F), msgb(9S)

Writing Device Drivers
STREAMS Programming Guide
copyout(9F)

NAME

copyout – copy data from a driver to a user program

SYNOPSIS

```c
#include <sys/types.h>
#include <sys/ddi.h>

int copyout(const void *driverbuf, void *userbuf, size_t cn);
```

INTERFACE

This interface is obsolete. ddi_copyout(9F) should be used instead.

LEVEL

PARAMETERS

- `driverbuf` Source address in the driver from which the data is transferred.
- `userbuf` Destination address in the user program to which the data is transferred.
- `cn` Number of bytes moved.

DESCRIPTION

`copyout()` copies data from driver buffers to user data space.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

RETURN VALUES

Under normal conditions, a 0 is returned to indicate a successful copy. Otherwise, a −1 is returned if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.

If a −1 is returned to the caller, driver entry point routines should return EFAULT.

CONTEXT

`copyout()` can be called from user context only.

EXAMPLES

**EXAMPLE 1 An ioctl() Routine**

A driver ioctl(9E) routine (line 10) can be used to get or set device attributes or registers. In the XX_GETREGS condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```c
1 struct device { /* layout of physical device registers */
2     int control; /* physical device control word */
3     int status; /* physical device status word */
4     short recv_char; /* receive character from device */
5     short xmit_char; /* transmit character to device */
6 };
7
8 extern struct device xx_addr[]; /* phys. device regs. location */
9 ...
10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,
```
EXAMPLE 1 An ioctl() Routine  (Continued)

```c
11    cred_t *cred_p, int *rval_p)
12    ...
13    {
14    register struct device *rp = \x_addr[getminor(dev) >> 4];
15    switch (cmd) {
16    case XX_GETREGS: /* copy device regs. to user program */
17    if (copyout(rp, arg, sizeof(struct device)))
18    return(EFAULT);
19    break;
20    ...}
22    }
23    ...
24 }
```

ATTRIBUTES  See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5), ioctl(9E), bcopy(9F), copyin(9F), ddi_copyin(9F), ddi_copyout(9F), uiomove(9F)

Writing Device Drivers

NOTES  Driver writers who intend to support layered ioctl in their ioctl(9E) routines should use ddi_copyout(9F) instead.

Driver defined locks should not be held across calls to this function.

copyout() should not be used from a streams driver. See M_COPYIN and M_COPYOUT in STREAMS Programming Guide.
csx_AccessConfigurationRegister(9F)

NAME  csx_AccessConfigurationRegister – read or write a PC Card Configuration Register

SYNOPSIS  
```c
#include <sys/pccard.h>

int32_t csx_AccessConfigurationRegister(client_handle_t ch,  
access_config_reg_t *acr);
```

INTERFACE LEVEL

PARAMETERS
- `ch`  Client handle returned from csx_RegisterClient(9F).
- `acr`  Pointer to an access_config_reg_t structure.

DESCRIPTION  This function allows a client to read or write a PC Card Configuration Register.

STRUCTURE

MEMBERS

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</td>
</tr>
<tr>
<td>Action</td>
<td>May be set to CONFIG_REG_READ or CONFIG_REG_WRITE. All other values in the Action field are reserved for future use. If the Action field is set to CONFIG_REG_WRITE, the Value field is written to the specified configuration register. Card Services does not read the configuration register after a write operation. For that reason, the Value field is only updated by a CONFIG_REG_READ request.</td>
</tr>
<tr>
<td>Offset</td>
<td>Specifies the byte offset for the desired configuration register from the PC Card configuration register base specified in csx_RequestConfiguration(9F).</td>
</tr>
<tr>
<td>Value</td>
<td>Contains the value read from the PC Card Configuration Register for a read operation. For a write operation, the Value field contains the value to write to the configuration register. As noted above, on return from a write request, the Value field is the value written to the PC Card and not any changed value that may have resulted from the write request (that is, no read after write is performed). A client must be very careful when writing to the COR (Configuration Option Register) at offset 0. This has the potential to change the type of interrupt request generated by the PC Card or place the card in the reset state. Either request may have undefined results. The client should read the register to determine the appropriate setting for the interrupt mode (Bit 6) before writing to the register.</td>
</tr>
</tbody>
</table>

The structure members of access_config_reg_t are:

- `uint32_t Socket; /* socket number*/`
- `uint32_t Action; /* register access operation*/`
- `uint32_t Offset; /* config register offset*/`
- `uint32_t Value; /* value read or written*/`
If a client wants to reset a PC Card, the `csx_ResetFunction(9F)` function should be used. Unlike `csx_AccessConfigurationRegister()`, the `csx_ResetFunction(9F)` function generates a series of event notifications to all clients using the PC Card, so they can re-establish the appropriate card state after the reset operation is complete.

**RETURN VALUES**

- **CS_SUCCESS**
  Successful operation.

- **CS_BAD_ARGS**
  Specified arguments are invalid. Client specifies an offset that is out of range or neither CONFIG_REG_READ or CONFIG_REG_WRITE is set.

- **CS_UNSUPPORTED_MODE**
  Client has not called `csx_RequestConfiguration(9F)` before calling this function.

- **CS_BAD_HANDLE**
  Client handle is invalid.

- **CS_NO_CARD**
  No PC card in socket.

- **CS_UNSUPPORTED_FUNCTION**
  No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

- `csx_ParseTuple(9F)`, `csx_RegisterClient(9F)`,
- `csx_RequestConfiguration(9F)`, `csx_ResetFunction(9F)`

*PCCard 95 Standard, PCMCIA/JEIDA*
csx_ConvertSize(9F)

NAME csx_ConvertSize – convert device sizes

SYNOPSIS #include <sys/pccard.h>

int32_t csx_ConvertSize(convert_size_t *cs);

INTERFACE Solaris DDI Specific (Solaris DDI)
LEVEL PARAMETERS cs Pointer to a convert_size_t structure.

DESCRIPTION csx_ConvertSize() is a Solaris-specific extension that provides a method for clients to convert from one type of device size representation to another, that is, from devsize format to bytes and vice versa.

STRUCTURE MEMBERS The structure members of convert_size_t are:

uint32_t Attributes;
uint32_t bytes;
uint32_t devsize;

The fields are defined as follows:

Attributes This is a bit-mapped field that identifies the type of size conversion to be performed. The field is defined as follows:

CONVERT_BYTES_TO_DEVSIZE
Converts bytes to devsize format.

CONVERT_DEVSIZE_TO_BYTES
Converts devsize format to bytes.

bytes If CONVERT_BYTES_TO_DEVSIZE is set, the value in the bytes field is converted to a devsize format and returned in the devsize field.

devsize If CONVERT_DEVSIZE_TO_BYTES is set, the value in the devsize field is converted to a bytes value and returned in the bytes field.

RETURN VALUES CS_SUCCESS Successful operation.
CS_BAD_SIZE Invalid bytes or devsize.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT This function may be called from user or kernel context.

SEE ALSO csx_ModifyWindow(9F), csx_RequestWindow(9F)

PCCard 95 Standard, PCMCIA/JEIDA
csx_ConvertSpeed(9F)

NAME

csx_ConvertSpeed – convert device speeds

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_ConvertSpeed(convert_speed_t *cs);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

cs Pointer to a convert_speed_t structure.

DESCRIPTION

This function is a Solaris-specific extension that provides a method for clients to convert from one type of device speed representation to another, that is, from devspeed format to nS and vice versa.

STRUCTURE

The structure members of convert_speed_t are:

- `Attributes`: uint32_t
- `nS`: uint32_t
- `devspeed`: uint32_t

The fields are defined as follows:

- `Attributes`: This is a bit-mapped field that identifies the type of speed conversion to be performed. The field is defined as follows:
  - `CONVERT_NS_TO_DEVSPEED`: Converts nS to devspeed format
  - `CONVERT_DEVSPEED_TO_NS`: Converts devspeed format to nS

- `nS`: If `CONVERT_NS_TO_DEVSPEED` is set, the value in the nS field is converted to a devspeed format and returned in the devspeed field.

- `devspeed`: If `CONVERT_DEVSPEED_TO_NS` is set, the value in the devspeed field is converted to an nS value and returned in the nS field.

RETURN VALUES

- `CS_SUCCESS`: Successful operation.
- `CS_BAD_SPEED`: Invalid nS or devspeed.
- `CS_BAD_ATTRIBUTE`: Bad Attributes value.
- `CS_UNSUPPORTED_FUNCTION`: No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

- `csx_ModifyWindow(9F)`, `csx_RequestWindow(9F)`

PC Card 95 Standard, PCMCIA/JEIDA
csx_CS_DDI_Info(9F)

NAME
csx_CS_DDI_Info – obtain DDI information

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_CS_DDI_Info(cs_ddi_info_t *cdi);

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

PARAMETERS

cdi Pointer to a cs_ddi_info_t structure.

DESCRIPTION
This function is a Solaris-specific extension that is used by clients that need to provide
the xx_getinfo driver entry point (see getinfo(9E)). It provides a method for clients to
obtain DDI information based on their socket number and client driver name.

STRUCTURE MEMBERS
The structure members of cs_ddi_info_t are:

uint32_t Socket; /* socket number */
char* driver_name; /* unique driver name */
dev_info_t *dip; /* dip */
int32_t instance; /* instance */

The fields are defined as follows:

Socket This field must be set to the physical socket number that the client
is interested in getting information about.

driver_name This field must be set to a string containing the name of the client
driver to get information about.

If csx_CS_DDI_Info() is used in a client’s xx_getinfo function, then the client will
typically extract the Socket value from the *arg argument and it must set the
driver_name field to the same string used with csx_RegisterClient(9F).

If the driver_name is found on the Socket, the csx_CS_DDI_Info() function
returns both the dev_info pointer and the instance fields for the requested driver
instance.

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_SOCKET Client not found on Socket.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

EXAMPLES
EXAMPLE 1 Using csx_CS_DDI_Info

The following example shows how a client might call the csx_CS_DDI_Info() in
the client’s xx_getinfo function to return the dip or the instance number:

static int
pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
void **result)
{
    int error = DDI_SUCCESS;

108 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
EXAMPLE 1: Using csx_CS_DDI_Info (Continued)

```c
pcepp_state_t *pps;
ss_dii_info_t cs_ddi_info;

switch (cmd) {
    case DDI_INFO_DEVT2DEVINFO:
        cs_ddi_info.Socket = getminor((dev_t)arg) & 0x3f;
        cs_ddi_info.driver_name = pcepp_name;
        if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS)
            return (DDI_FAILURE);
        if (!(pps = ddi_get_soft_state(pcepp_soft_state_p,
                                    cs_ddi_info.instance))) {
            *result = NULL;
        } else {
            *result = pps->dip;
        }
        break;

    case DDI_INFO_DEVT2INSTANCE:
        cs_ddi_info.Socket = getminor((dev_t)arg) & 0x3f;
        cs_ddi_info.driver_name = pcepp_name;
        if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS)
            return (DDI_FAILURE);
        *result = (void *)cs_ddi_info.instance;
        break;

    default:
        error = DDI_FAILURE;
        break;

}

return (error);
```

SEE ALSO getinfo(9E), csx_RegisterClient(9F), ddi_get_instance(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_DeregisterClient(9F)

NAME
csx_DeregisterClient – remove client from Card Services list

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_DeregisterClient(client_handle_t ch);

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

PARAMETERS
ch Client handle returned from csx_RegisterClient(9F).

DESCRIPTION
This function removes a client from the list of registered clients maintained by Card Services. The Client Handle returned by csx_RegisterClient(9F) is passed in the client_handle_t argument.

The client must have returned all requested resources before this function is called. If any resources have not been released, CS_IN_USE is returned.

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_IN_USE Resources not released by this client.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

SEE ALSO
csx_RegisterClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA

WARNINGS
Clients should be prepared to receive callbacks until Card Services returns from this request successfully.
csx_DupHandle(9F)

NAME

csx_DupHandle – duplicate access handle

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_DupHandle(acc_handle_t handle1, acc_handle_t *handle2, uint32_t flags);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle1</td>
<td>The access handle returned from csx_RequestIO(9F) or csx_RequestWindow(9F) that is to be duplicated.</td>
</tr>
<tr>
<td>handle2</td>
<td>A pointer to the newly-created duplicated data access handle.</td>
</tr>
<tr>
<td>flags</td>
<td>The access attributes that will be applied to the new handle.</td>
</tr>
</tbody>
</table>

DESCRIPTION

This function duplicates the handle, handle1, into a new handle, handle2, that has the access attributes specified in the flags argument. Both the original handle and the new handle are active and can be used with the common access functions.

Both handles must be explicitly freed when they are no longer necessary.

The flags argument is bit-mapped. The following bits are defined:

- **WIN_ACC_NEVER_SWAP** Host endian byte ordering
- **WIN_ACC_BIG_ENDIAN** Big endian byte ordering
- **WIN_ACC_LITTLE_ENDIAN** Little endian byte ordering
- **WIN_ACC_STRICT_ORDER** Program ordering references
- **WIN_ACC_UNORDERED_OK** May re-order references
- **WIN_ACC_MERGING_OK** Merge stores to consecutive locations
- **WIN_ACC_LOADCACHING_OK** May cache load operations
- **WIN_ACC_STORECACHING_OK** May cache store operations

WIN_ACC_BIG_ENDIAN and WIN_ACC_LITTLE_ENDIAN describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When WIN_ACC_BIG_ENDIAN or WIN_ACC_LITTLE_ENDIAN is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities. When WIN_ACC_NEVER_SWAP is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following flags bits. Only one of the following bits may be specified:

- **WIN_ACC.Strict_ORDER** The data references must be issued by a CPU in program order. Strict ordering is the default behavior.
- **WIN_ACC.UNORDERED_OK** The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).
WIN_ACC_MERGING_OK

The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. Setting this bit also implies re-ordering.

WIN_ACC_LOADCACHING_OK

The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. Setting this bit also implies merging and re-ordering.

WIN_ACC_STORECACHING_OK

The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. Setting this bit also implies load caching, merging, and re-ordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together.

RETURN VALUES

CS_SUCCESS

Successful operation.

CS_FAILURE

Error in flags argument or handle could not be duplicated for some reason.

CS_UNSUPPORTED_FUNCTION

No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F),
csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Error2Text – convert error return codes to text strings

**SYNOPSIS**

```c
#include <sys/pccard.h>

int32_t csx_Error2Text(error2text_t *er);
```

**INTERFACE LEVEL**

Solaris DDI Specific (Solaris DDI)

**PARAMETERS**

- `error2text_t *er` Pointer to an error2text_t structure.

**DESCRIPTION**

This function is a Solaris-specific extension that provides a method for clients to convert Card Services error return codes to text strings.

The structure members of `error2text_t` are:

```c
type item; /*the error code*/
char test[CS_ERROR_MAX_BUFSIZE]; /*the error code*/
```

A pointer to the text for the Card Services error return code in the `item` field is returned in the `text` field if the error return code is found. The client is not responsible for allocating a buffer to hold the text. If the Card Services error return code specified in the `item` field is not found, the `text` field will be set to a string of the form:

```
"{unknown Card Services return code}"
```

**RETURN VALUES**

- `CS_SUCCESS` Successful operation.
- `CS_UNSUPPORTED_FUNCTION` No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1**: Using the `csxError2Text` function

```c
if ((ret = csx_RegisterClient(&client_handle, &client_reg)) != CS_SUCCESS)
{
    error2text_t error2text;
    error2text.item = ret;
    csx_Error2Text(&error2text);
    cmn_err(CE_CONT, "RegisterClient failed %s (0x%x)",
            error2text.text, ret);
}
```

**SEE ALSO**

- `csx_Event2Text(9F)`
- `PC Card 95 Standard, PCMCIA/JEIDA`
csx_Event2Text(9F)

NAME  csx_Event2Text – convert events to text strings

SYNOPSIS  

```c
#include <sys/pccard.h>

int32_t csx_Event2Text(event2text_t *ev);
```

INTERFACE  Solaris DDI Specific (Solaris DDI)

LEVEL  PARAMETERS  

```c

int32_t csx_Event2Text(event2text_t *ev);
```

PARAMETERS  

`ev`  Pointer to an event2text_t structure.

DESCRIPTION  This function is a Solaris-specific extension that provides a method for clients to convert Card Services events to text strings.

STRUCTURE  Members  

The structure members of event2text_t are:

```c

event_t event; /*the event code*/
char text[CS_EVENT_MAX_BUFSIZE] /*the event code*/
```

The fields are defined as follows:

- `event`: The text for the event code in the `event` field is returned in the `text` field.
- `text`: The text string describing the name of the event.

RETURN VALUES  

- `CS_SUCCESS`: Successful operation.
- `CS_UNSUPPORTED_FUNCTION`: No PCMCIA hardware installed.

CONTEXT  

This function may be called from user or kernel context.

EXAMPLES  

**EXAMPLE 1**: Using `csx_Event2Text()`

```c

xx_event(event_t event, int priority, event_callback_args_t *eca)
{
    event2text_t event2text;

    event2text.event = event;
    csx_Event2Text(&event2text);
    cmn_err(CE_CONT, "event %s (0x%x)", event2text.text, (int)event);
}
```

SEE ALSO  

- `csx_event_handler(9E), csx_Error2Text(9F)`

PC Card 95 Standard, PCMCIA/JEIDA

114  man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
### csx_FreeHandle(9F)

**NAME**
csx_FreeHandle – free access handle

**SYNOPSIS**
```
#include <sys/pccard.h>

int32_t csx_FreeHandle(acc_handle_t *handle);
```

**INTERFACE LEVEL**
Solaris DDI Specific (Solaris DDI)

**PARAMETERS**
- **handle**: The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).

**DESCRIPTION**
This function frees the handle, `handle`. If the handle was created by the csx_DupHandle(9F) function, this function will free the storage associated with this handle, but will not modify any resources that the original handle refers to. If the handle was created by a common access setup function, this function will release the resources associated with this handle.

**RETURN VALUES**
- **CS_SUCCESS**: Successful operation.
- **CS_UNSUPPORTED_FUNCTION**: No PCMCIA hardware installed.

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_DupHandle(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

*PC Card95 Standard, PCMCIA/JEIDA*
csx_Get16(9F)

NAME

csx_Get8, csx_Get16, csx_Get32, csx_Get64 – read data from device address

SYNOPSIS

#include <sys/pccard.h>

uint8_t csx_Get8(acc_handle_t handle, uint32_t offset);

uint16_t csx_Get16(acc_handle_t handle, uint32_t offset);

uint32_t csx_Get32(acc_handle_t handle, uint32_t offset);

uint64_t csx_Get64(acc_handle_t handle, uint64_t offset);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

handle The access handle returned from csx_RequestIO(9F),
          csx_RequestWindow(9F), or csx_DupHandle(9F).

offset The offset in bytes from the base of the mapped resource.

DESCRIPTION

These functions generate a read of various sizes from the mapped memory or device
register.

The csx_Get8(), csx_Get16(), csx_Get32(), and csx_Get64() functions read
8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address
represented by the handle, handle, at an offset in bytes represented by the offset, offset.

Data that consists of more than one byte will automatically be translated to maintain a
consistent view between the host and the device based on the encoded information in
the data access handle. The translation may involve byte swapping if the host and the
device have incompatible endian characteristics.

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions may be called from user, kernel, or interrupt context.

SEE ALSO

csx_DupHandle(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F),
 csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
**NAME**  
csx_Get8, csx_Get16, csx_Get32, csx_Get64 – read data from device address

**SYNOPSIS**  
#include <sys/pccard.h>

```c
uint8_t csx_Get8(acc_handle_t handle, uint32_t offset);
uint16_t csx_Get16(acc_handle_t handle, uint32_t offset);
uint32_t csx_Get32(acc_handle_t handle, uint32_t offset);
uint64_t csx_Get64(acc_handle_t handle, uint64_t offset);
```

**INTERFACE LEVEL**  
Solaris DDI Specific (Solaris DDI)

**PARAMETERS**  
- **handle**  
The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- **offset**  
The offset in bytes from the base of the mapped resource.

**DESCRIPTION**  
These functions generate a read of various sizes from the mapped memory or device register.

The `csx_Get8()`, `csx_Get16()`, `csx_Get32()`, and `csx_Get64()` functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

**RETURN VALUES**  
These functions return the value read from the mapped address.

**CONTEXT**  
These functions may be called from user, kernel, or interrupt context.

**SEE ALSO**  
csx_DupHandle(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F), csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

`PC Card 95 Standard, PCMCIA/JEIDA`
## NAME

csx_Get8, csx_Get16, csx_Get32, csx_Get64 – read data from device address

## SYNOPSIS

```c
#include <sys/pccard.h>

uint8_t csx_Get8(acc_handle_t handle, uint32_t offset);
uint16_t csx_Get16(acc_handle_t handle, uint32_t offset);
uint32_t csx_Get32(acc_handle_t handle, uint32_t offset);
uint64_t csx_Get64(acc_handle_t handle, uint64_t offset);
```

## INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

## PARAMETERS

- **handle**
  The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
- **offset**
  The offset in bytes from the base of the mapped resource.

## DESCRIPTION

These functions generate a read of various sizes from the mapped memory or device register.

The `csx_Get8()`, `csx_Get16()`, `csx_Get32()`, and `csx_Get64()` functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

## RETURN VALUES

These functions return the value read from the mapped address.

## CONTEXT

These functions may be called from user, kernel, or interrupt context.

## SEE ALSO

- csx_DupHandle(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F), csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

**PC Card 95 Standard, PCMCIA/JEIDA**
NAME | csx_Get8, csx_Get16, csx_Get32, csx_Get64 – read data from device address

SYNOPSIS
#include <sys/pccard.h>

uint8_t csx_Get8(acc_handle_t handle, uint32_t offset);
uint16_t csx_Get16(acc_handle_t handle, uint32_t offset);
uint32_t csx_Get32(acc_handle_t handle, uint32_t offset);
uint64_t csx_Get64(acc_handle_t handle, uint64_t offset);

INTERFACE LEVEL PARAMETERS
Solaris DDI Specific (Solaris DDI)

handle The access handle returned from csx_RequestIO(9F),
          csx_RequestWindow(9F), or csx_DupHandle(9F).

offset The offset in bytes from the base of the mapped resource.

DESCRIPTION
These functions generate a read of various sizes from the mapped memory or device register.

The csx_Get8(), csx_Get16(), csx_Get32(), and csx_Get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

RETURN VALUES
These functions return the value read from the mapped address.

CONTEXT
These functions may be called from user, kernel, or interrupt context.

SEE ALSO
csx_DupHandle(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F),
csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_GetEventMask(9F)

NAME

csx_SetEventMask, csx_GetEventMask – set or return the client event mask for the client.

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_SetEventMask(client_handle_t ch, sockevent_t *se);
int32_t csx_GetEventMask(client_handle_t ch, sockevent_t *se);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).
se Pointer to a sockevent_t structure

DESCRIPTION

The function csx_SetEventMask() sets the client or global event mask for the client.

The function csx_GetEventMask() returns the client or global event mask for the client.

csx_RequestSocketMask(9F) must be called before calling csx_SetEventMask() for the client event mask for this socket.

STRUCTURE MEMBERS

The structure members of sockevent_t are:

uint32_t uint32_t /* attribute flags for call */
uint32_t EventMask; /* event mask to set or return */
uint32_t Socket; /* socket number if necessary */

The fields are defined as follows:

Attributes

This is a bit-mapped field that identifies the type of event mask to be returned. The field is defined as follows:

CONF_EVENT_MASK_GLOBAL

Client’s global event mask. If set, the client’s global event mask is returned.

CONF_EVENT_MASK_CLIENT

Client’s local event mask. If set, the client’s local event mask is returned.

EventMask

This field is bit-mapped. Card Services performs event notification based on this field. See csx_event_handler(9E) for valid event definitions and for additional information about handling events.

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

RETURN VALUES

CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
csx_GetEventMask(9F)

<table>
<thead>
<tr>
<th>CS_BAD_SOCKET</th>
<th>csx_RequestSocketMask(9F) not called for CONF_EVENT_MASK_CLIENT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**
These functions may be called from user or kernel context.

**SEE ALSO**

csx_event_handler(9E), csx_RegisterClient(9F),
csx_ReleaseSocketMask(9F), csx_RequestSocketMask(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_GetFirstClient

**NAME**
csx_GetFirstClient, csx_GetNextClient – return first or next client

**SYNOPSIS**
```c
#include <sys/pccard.h>

int32_t csx_GetFirstClient(get_firstnext_client_t *fnc);
int32_t csx_GetNextClient(get_firstnext_client_t *fnc);
```

**INTERFACE LEVEL**
Solaris DDI Specific (Solaris DDI)

**PARAMETERS**
`fnc` Pointer to a `get_firstnext_client_t` structure.

**DESCRIPTION**
The functions `csx_GetFirstClient()` and `csx_GetNextClient()` return information about the first or subsequent PC cards, respectively, that are installed in the system.

**STRUCTURE MEMBERS**
The structure members of `get_firstnext_client_t` are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t</code></td>
<td><code>Socket</code>;</td>
<td>socket number *</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>Attributes</code>;</td>
<td>attributes *</td>
</tr>
<tr>
<td><code>client_handle_t</code></td>
<td><code>client_handle</code>;</td>
<td>client handle *</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>num_clients</code>;</td>
<td>number of clients *</td>
</tr>
</tbody>
</table>

The fields are defined as follows:

- **Socket**
  - If the `CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY` attribute is set, return information only on the PC card installed in this socket.

- **Attributes**
  - This field indicates the type of client. The field is bit-mapped; the following bits are defined:
    - `CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS`
      - Return information on all clients.
    - `CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY`
      - Return client information for the specified socket only.

- **client_handle**
  - The client handle of the PC card driver is returned in this field.

- **num_clients**
  - The number of clients is returned in this field.

**RETURN VALUES**
- **CS_SUCCESS**
  - Successful operation.
- **CS_BAD_HANDLE**
  - Client handle is invalid.
- **CS_BAD_SOCKET**
  - Socket number is invalid.
- **CS_NO_CARD**
  - No PC Card in socket.
- **CS_NO_MORE_ITEMS**
  - PC Card driver does not handle the `CS_EVENT_CLIENT_INFO` event.
- **CS_UNSUPPORTED_FUNCTION**
  - No PCMCIA hardware installed.

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_event_handler(9E)
csx_GetFirstClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_GetFirstTuple(9F)

NAME  csx_GetFirstTuple, csx_GetNextTuple – return Card Information Structure tuple

SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_GetFirstTuple(client_handle_t ch, tuple_t *tu);
int32_t csx_GetNextTuple(client_handle_t ch, tuple_t *tu);
```

INTERFACE

LEVEL

PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch  Client handle returned from csx_RegisterClient(9F).

tu  Pointer to a tuple_t structure.

DESCRIPTION

The functions csx_GetFirstTuple() and csx_GetNextTuple() return the first and next tuple, respectively, of the specified type in the Card Information Structure (CIS) for the specified socket.

STRUCTURE

MEMBERS

The structure members of tuple_t are:

```
uint32_t Socket; /* socket number */
uint32_t Attributes; /* Attributes */
cisdata_t DesiredTuple; /* tuple to search for or flags */
cisdata_t TupleCode; /* tuple type code */
cisdata_t TupleLink; /* tuple data body size */
```

The fields are defined as follows:

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes

This field is bit-mapped. The following bits are defined:

TUPLE_RETURN_LINK

Return link tuples if set. The following are link tuples and will only be returned by this function if the TUPLE_RETURN_LINK bit in the Attributes field is set:

```
CISTPL_NULL  CISTPL_LONGLINK_MFC
CISTPL_LONGLINK_A  CISTPL_LINKTARGET
CISTPL_LONGLINK_C  CISTPL_NO_LINK
CISTPL_LONGLINK_CB  CISTPL_END
```

TUPLE_RETURN_IGNORED_TUPLES

Return ignored tuples if set. Ignored tuples will be returned by this function if the TUPLE_RETURN_IGNORED_TUPLES bit in the Attributes field is set, see tuple(9S) for more information. The CIS is parsed from the location setup by the previous csx_GetFirstTuple() or csx_GetNextTuple() request.

DesiredTuple

This field is the tuple value desired. If it is RETURN_FIRST_TUPLE, the very first tuple of the CIS is returned (if it exists). If this field is set to RETURN_NEXT_TUPLE,
the very next tuple of the CIS is returned (if it exists). If the DesiredTuple field is any other value on entry, the CIS is searched in an attempt to locate a tuple which matches.

TupleCode, TupleLink
These fields are the values returned from the tuple found. If there are no tuples on the card, CS_NO_MORE_ITEMS is returned.

Since the csx_GetFirstTuple(), csx_GetNextTuple(), and csx_GetTupleData(9F) functions all share the same tuple_t structure, some fields in the tuple_t structure are unused or reserved when calling this function and these fields must not be initialized by the client.

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_NO_CARD No PC Card in socket.
CS_NO_CIS No Card Information Structure (CIS) on PC card.
CS_NO_MORE_ITEMS Desired tuple not found.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT These functions may be called from user or kernel context.

SEE ALSO csx_GetTupleData(9F), csx_ParseTuple(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)

PC Card 95Standard, PCMCIA/JEIDA
csx_GetHandleOffset(9F)

NAME csx_GetHandleOffset – return current access handle offset

SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_GetHandleOffset(acc_handle_t handle, uint32_t *offset);
```

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).</td>
</tr>
<tr>
<td>offset</td>
<td>Pointer to a uint32_t in which the current access handle offset is returned.</td>
</tr>
</tbody>
</table>

DESCRIPTION

This function returns the current offset for the access handle, `handle`, in `offset`.

RETURN VALUES

- **CS_SUCCESS**
  - Successful operation.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_RequestIO(9F), csx_RequestIRQ(9F), csx_SetHandleOffset(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
**NAME**
csx_GetMappedAddr – return mapped virtual address

**SYNOPSIS**
```
#include <sys/pccard.h>

int32_t csx_GetMappedAddr(acc_handle_t handle, void **addr);
```

**INTERFACE LEVEL**
Solaris DDI Specific (Solaris DDI)

**PARAMETERS**
- **handle**
  The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- **addr**
  The virtual or I/O port number represented by the handle.

**DESCRIPTION**
This function returns the mapped virtual address or the mapped I/O port number represented by the handle, `handle`.

**RETURN VALUES**
- **CS_SUCCESS**
  The resulting address or I/O port number can be directly accessed by the caller.
- **CS_FAILURE**
  The resulting address or I/O port number can not be directly accessed by the caller; the caller must make all accesses to the mapped area via the common access functions.
- **CS_UNSUPPORTED_FUNCTION**
  No PCMCIA hardware installed.

**CONTEXT**
This function may be called from user, kernel, or interrupt context.

**SEE ALSO**
- `csx_DupHandle(9F)`, `csx_Get8(9F)`, `csx_Put8(9F)`, `csx_RepGet8(9F)`, `csx_RepPut8(9F)`, `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`

*PC Card 95 Standard, PCMCIA/JEIDA*
The functions `csx_GetFirstClient()` and `csx_GetNextClient()` return information about the first or subsequent PC cards, respectively, that are installed in the system.

The structure members of `get_firstnext_client_t` are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t</code></td>
<td>Socket; /* socket number */</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>Attributes; /* attributes */</td>
</tr>
<tr>
<td><code>client_handle_t</code></td>
<td>client_handle; /* client handle */</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>num_clients; /* number of clients */</td>
</tr>
</tbody>
</table>

The fields are defined as follows:

- **Socket**
  - If the `CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY` attribute is set, return information only on the PC card installed in this socket.

- **Attributes**
  - This field indicates the type of client. The field is bit-mapped; the following bits are defined:
    - `CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS`: Return information on all clients.
    - `CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY`: Return client information for the specified socket only.

- **client_handle**
  - The client handle of the PC card driver is returned in this field.

- **num_clients**
  - The number of clients is returned in this field.

**RETURN VALUES**

- **CS_SUCCESS**
  - Successful operation.

- **CS_BAD_HANDLE**
  - Client handle is invalid.

- **CS_BAD_SOCKET**
  - Socket number is invalid.

- **CS_NO_CARD**
  - No PC Card in socket.

- **CS_NO_MORE_ITEMS**
  - PC Card driver does not handle the `CS_EVENT_CLIENT_INFO` event.

- **CS_UNSUPPORTED_FUNCTION**
  - No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

`csx_event_handler(9E)`
csx_GetNextClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_GetFirstTuple(9F)

NAME
csx_GetFirstTuple, csx_GetNextTuple – return Card Information Structure tuple

SYNOPSIS
#include <sys/pccard.h>
int32_t csx_GetFirstTuple(client_handle_t ch, tuple_t *tu);
int32_t csx_GetNextTuple(client_handle_t ch, tuple_t *tu);

INTERFACE
Solaris DDI Specific (Solaris DDI)

PARAMETERS
ch Client handle returned from csx_RegisterClient(9F).

 tu Pointer to a tuple_t structure.

DESCRIPTION
The functions csx_GetFirstTuple() and csx_GetNextTuple() return the first
and next tuple, respectively, of the specified type in the Card Information Structure
(CIS) for the specified socket.

STRUCTURE
The structure members of tuple_t are:

  uint32_t Socket; /* socket number */
  uint32_t Attributes; /* Attributes */
  cisdata_t DesiredTuple; /* tuple to search for or flags */
  cisdata_t TupleCode; /* tuple type code */
  cisdata_t TupleLink; /* tuple data body size */

The fields are defined as follows:

Socket
Not used in Solaris, but for portability with other Card Services implementations, it
should be set to the logical socket number.

Attributes
This field is bit-mapped. The following bits are defined:

TUPLE_RETURN_LINK
Return link tuples if set. The following are link tuples and will only be returned
by this function if the TUPLE_RETURN_LINK bit in the Attributes field is set:

  CISTPL_NULL   CISTPL_LONGLINK_MFC
  CISTPL_LONGLINK_A   CISTPL_LINKTARGET
  CISTPL_LONGLINK_C   CISTPL_NO_LINK
  CISTPL_LONGLINK_CB  CISTPL_END

TUPLE_RETURN_IGNORED_TUPLES
Return ignored tuples if set. Ignored tuples will be returned by this function if
the TUPLE_RETURN_IGNORED_TUPLES bit in the Attributes field is set, see
tuple(9S) for more information. The CIS is parsed from the location setup by the
previous csx_GetFirstTuple() or csx_GetNextTuple() request.

DesiredTuple
This field is the tuple value desired. If it is RETURN_FIRST_TUPLE, the very first
tuple of the CIS is returned (if it exists). If this field is set to RETURN_NEXT_TUPLE,
the very next tuple of the CIS is returned (if it exists). If the DesiredTuple field is any other value on entry, the CIS is searched in an attempt to locate a tuple which matches.

**TupleCode, TupleLink**
These fields are the values returned from the tuple found. If there are no tuples on the card, CS_NO_MORE_ITEMS is returned.

Since the csx_GetFirstTuple(), csx_GetNextTuple(), and csx_GetTupleData(9F) functions all share the same tuple_t structure, some fields in the tuple_t structure are unused or reserved when calling this function and these fields must not be initialized by the client.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC card.</td>
</tr>
<tr>
<td>CS_NO_MORE_ITEMS</td>
<td>Desired tuple not found.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**
These functions may be called from user or kernel context.

**SEE ALSO**
csx_GetTupleData(9F), csx_ParseTuple(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)

*PC Card 95Standard, PCMCIA/JEIDA*
csx_GetStatus(9F)

NAME

csx_GetStatus – return the current status of a PC Card and its socket

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_GetStatus(client_handle_t ch, get_status_t *gs);

INTERFACE LEVEL PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

gs Pointer to a get_status_t structure.

DESCRIPTION

This function returns the current status of a PC Card and its socket.

STRUCTURE MEMBERS

The structure members of get_status_t are:

uint32_t Socket; /* socket number*/
uint32_t CardState; /* "live" card status for this client*/
uint32_t SocketState; /* latched socket values */
uint32_t raw_CardState; /* raw live card status */

The fields are defined as follows:

Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

CardState The CardState field is the bit-mapped output data returned from Card Services. The bits identify what Card Services thinks the current state of the installed PC Card is. The bits are:

CS_STATUS_WRITE_PROTECTED Card is write protected
CS_STATUS_CARD_LOCKED Card is locked
CS_STATUS_EJECTION_REQUEST Ejection request in progress
CS_STATUS_INSERTION_REQUEST Insertion request in progress
CS_STATUS_BATTERY_DEAD Card battery is dead
CS_STATUS_BATTERY_DEAD Card battery is dead (BVD1)
CS_STATUS_BATTERY_LOW Card battery is low (BVD2)
CS_STATUS_CARD_READY Card is READY
CS_STATUS_CARD_INSERTED Card is inserted
CS_STATUS_REQ_ATTN
Extended status attention request

CS_STATUS_RES_EVT1
Extended status reserved event status

CS_STATUS_RES_EVT2
Extended status reserved event status

CS_STATUS_RES_EVT3
Extended status reserved event status

CS_STATUS_VCC_50
5.0 Volts Vcc Indicated

CS_STATUS_VCC_33
3.3 Volts Vcc Indicated

CS_STATUS_VCC_XX
X.X Volts Vcc Indicated

The state of the CS_STATUS_CARD_INSERTED bit indicates whether the PC Card associated with this driver instance, not just any card, is inserted in the socket. If an I/O card is installed in the specified socket, card state is returned from the PRR (Pin Replacement Register) and the ESR (Extended Status Register) (if present). If certain state bits are not present in the PRR or ESR, a simulated state bit value is returned as defined below:

CS_STATUS_WRITE_PROTECTED
Not write protected

CS_STATUS_BATTERY_DEAD
Power good

PCS_STATUS_BATTERY_LOW
Power good

CS_STATUS_CARD_READY
Ready

CS_STATUS_REQ_ATTN
Not set

CS_STATUS_RES_EVT1
Not set

CS_STATUS_RES_EVT2
Not set

CS_STATUS_RES_EVT3
Not set

The SocketState field is a bit-map of the current card and socket state. The bits are:

Kernel Functions for Drivers  133
csx_GetStatus(9F)

<table>
<thead>
<tr>
<th>CS_SOCK_STATUS_WRITE_PROTECT_CHANGE</th>
<th>Write Protect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECS_SOCK_STATUS_CARD_LOCK_CHANGE</td>
<td>Card Lock Change</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_EJECTION_PENDING</td>
<td>Ejection Request</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_INSERTION_PENDING</td>
<td>Insertion Request</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_BATTERY_DEAD_CHANGE</td>
<td>Battery Dead</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_BATTERY_LOW_CHANGE</td>
<td>Battery Low</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_CARD_READYCHANGE</td>
<td>Ready Change</td>
</tr>
<tr>
<td>CS_SOCK_STATUS_CARD_INSERTION_CHANGE</td>
<td>Card is inserted</td>
</tr>
</tbody>
</table>

The state reported in the SocketState field may be different from the state reported in the CardState field. Clients should normally depend only on the state reported in the CardState field.

The state reported in the SocketState field may be different from the state reported in the CardState field. Clients should normally depend only on the state reported in the CardState field.

**raw_CardState**
The `raw_CardState` field is a Solaris-specific extension that allows the client to determine if any card is inserted in the socket. The bit definitions in the `raw_CardState` field are identical to those in the `CardState` field with the exception that the `CS_STATUS_CARD_INSERTED` bit in the `raw_CardState` field is set whenever any card is inserted into the socket.

### RETURN VALUES

| CS_SUCCESS | Successful operation. |
| CS_BAD_HANDLE | Client handle is invalid. |
| CS_BAD_SOCKET | Error getting socket state. |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed. |

**CS_NO_CARD** will not be returned if there is no PC Card present in the socket.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

csx_RegisterClient(9F)
csx_GetStatus(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_GetTupleData(9F)

NAME  csx_GetTupleData – return the data portion of a tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_GetTupleData(client_handle_t ch, tuple_t *tu);

INTERFACE SPECIFICATION

Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch  Client handle returned from csx_RegisterClient(9F).

tu  Pointer to a tuple_t structure.

DESCRIPTION

This function returns the data portion of a tuple, as returned by the csx_GetFirstTuple(9F) and csx_GetNextTuple(9F) functions.

STRUCTURE MEMBERS

The structure members of tuple_t are:

The fields are defined as follows:

uint32_t  Socket;  /* socket number */
uint32_t  Attributes;  /* tuple attributes*/
cisdata_t DesiredTuple;  /* tuple to search for*/
cisdata_t TupleOffset;  /* tuple data offset*/
cisdata_t TupleDataMax;  /* max tuple data size*/
cisdata_t TupleDataLen;  /* actual tuple data length*/
cisdata_t TupleData[CIS_MAX_TUPLE_DATA_LEN];  /* tuple body data buffer*/
cisdata_t TupleCode;  /* tuple type code*/
cisdata_t TupleLink;  /* tuple link */

Socket  Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes  Initialized by csx_GetFirstTuple(9F) or csx_GetNextTuple(9F); the client must not modify the value in this field.

DesiredTuple  Initialized by csx_GetFirstTuple(9F) or csx_GetNextTuple(9F); the client must not modify the value in this field.

TupleOffset  This field allows partial tuple information to be retrieved, starting anywhere within the tuple.

TupleDataMax  This field is the size of the tuple data buffer that Card Services uses to return raw tuple data from csx_GetTupleData(9F). It can be larger than the number of bytes in the tuple data body. Card Services ignores any value placed here by the client.

TupleDataLen  This field is the actual size of the tuple data body. It represents the number of tuple data body bytes returned.
**csx_GetTupleData(9F)**

**TupleData**
This field is an array of bytes containing the raw tuple data body contents.

**TupleCode**
Initialized by `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`, the client must not modify the value in this field.

**TupleLink**
Initialized by `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`, the client must not modify the value in this field.

**RETURN VALUES**
- **CS_SUCCESS**
  Successful operation.
- **CS_BAD_HANDLE**
  Client handle is invalid.
- **CS_BAD_ARGS**
  Data from prior `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)` is corrupt.
- **CS_NO_CARD**
  No PC Card in socket.
- **CS_NO_CIS**
  No Card Information Structure (CIS) on PC Card.
- **CS_NO_MORE_ITEMS**
  Card Services was not able to read the tuple from the PC Card.
- **CS_UNSUPPORTED_FUNCTION**
  No PCMCIA hardware installed.

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
- `csx_GetFirstTuple(9F)`, `csx_ParseTuple(9F)`, `csx_RegisterClient(9F)`, `csx_ValidateCIS(9F)`, `tuple(9S)`
- *PC Card 95 Standard, PCMCIA/JEIDA*
csx_MakeDeviceNode(9F)

NAME

csx_MakeDeviceNode, csx_RemoveDeviceNode - create and remove minor nodes on behalf of the client

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_MakeDeviceNode(client_handle_t ch, make_device_node_t *dn);

int32_t csx_RemoveDeviceNode(client_handle_t ch, remove_device_node_t *dn);

INTERFACE LEVEL

PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch  Client handle returned from csx_RegisterClient(9F).

dn  Pointer to a make_device_node_t or remove_device_node_t structure.

DESCRIPTION

csx_MakeDeviceNode() and csx_RemoveDeviceNode() are Solaris-specific extensions to allow the client to request that device nodes in the filesystem are created or removed, respectively, on its behalf.

STRUCTURE MEMBERS

The structure members of make_device_node_t are:

uint32_t Action; /* device operation */
uint32_t NumDevNodes; /* number of nodes to create */
devnode_desc_t *devnode_desc; /* description of device nodes */

The structure members of remove_device_node_t are:

uint32_t Action; /* device operation */
uint32_t NumDevNodes; /* number of nodes to remove */
devnode_desc_t *devnode_desc; /* description of device nodes */

The structure members of devnode_desc_t are:

char *name; /* device node path and name */
int32_t spec_type; /* device special type (block or char) */
int32_t minor_num; /* device node minor number */
char *node_type; /* device node type */

The Action field is used to specify the operation that csx_MakeDeviceNode() and csx_RemoveDeviceNode() should perform.

The following Action values are defined for csx_MakeDeviceNode():

CREATE_DEVICE_NODE
    Create NumDevNodes minor nodes

The following Action values are defined for csx_RemoveDeviceNode():

REMOVEDEVICE_NODE
    Remove NumDevNodes minor nodes

REMOVE_ALL_DEVICE_NODES
    Remove all minor nodes for this client
For csx_MakeDeviceNode(), if the Action field is:

CREATE_DEVICE_NODE
The NumDevNodes field must be set to the number of minor devices to create, and the client must allocate the quantity of devnode_desc_t structures specified by NumDevNodes and fill out the fields in the devnode_desc_t structure with the appropriate minor node information. The meanings of the fields in the devnode_desc_t structure are identical to the parameters of the same name to the ddi_create_minor_node(9F) DDI function.

For csx_RemoveDeviceNode(), if the Action field is:

REMOVE_DEVICE_NODE
The NumDevNodes field must be set to the number of minor devices to remove, and the client must allocate the quantity of devnode_desc_t structures specified by NumDevNodes and fill out the fields in the devnode_desc_t structure with the appropriate minor node information. The meanings of the fields in the devnode_desc_t structure are identical to the parameters of the same name to the ddi_remove_minor_node(9F) DDI function.

REMOVE_ALL_DEVICE_NODES
The NumDevNodes field must be set to 0 and the devnode_desc_t structure pointer must be set to NULL. All device nodes for this client will be removed from the filesystem.

RETURN VALUES
CS_SUCCESS  Successful operation.
CS_BAD_HANDLE  Client handle is invalid.
CS_BAD_ATTRIBUTE  The value of one or more arguments is invalid.
CS_BAD_ARGS  Action is invalid.
CS_OUT_OF_RESOURCE  Unable to create or remove device node.
CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

CONTEXT
These functions may be called from user or kernel context.

SEE ALSO
csx_RegisterClient(9F), ddi_create_minor_node(9F), ddi_remove_minor_node(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_MapLogSocket(9F)

NAME
   csx_MapLogSocket – return the physical socket number associated with the client handle

SYNOPSIS
   #include <sys/pccard.h>

   int32_t csx_MapLogSocket(client_handle_t ch, map_log_socket_t *Is);

INTERFACE
   Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS
   ch  Client handle returned from csx_RegisterClient(9F).

   Is  Pointer to a map_log_socket_t structure.

DESCRIPTION
   This function returns the physical socket number associated with the client handle.

STRUCTURE
   The structure members of map_log_socket_t are:

      uint32_t  LogSocket;  /* logical socket number */
      uint32_t  PhyAdapter; /* physical adapter number */
      uint32_t  PhySocket;  /* physical socket number */

   The fields are defined as follows:

      LogSocket  Not used by this implementation of Card Services and can be set to any arbitrary value.

      PhyAdapter Returns the physical adapter number, which is always 0 in the Solaris implementation of Card Services.

      PhySocket  Returns the physical socket number associated with the client handle. The physical socket number is typically used as part of an error or message string or if the client creates minor nodes based on the physical socket number.

RETURN VALUES
   CS_SUCCESS  Successful operation.

   CS_BAD_HANDLE  Client handle is invalid.

   CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

CONTEXT
   This function may be called from user or kernel context.

SEE ALSO
   csx_RegisterClient(9F)

   PC Card 95 Standard, PCMCIA/JEIDA

140  man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
NAME

csx_MapMemPage – map the memory area on a PC Card

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_MapMemPage(window_handle_t wh, map_mem_page_t *mp);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

wh Window handle returned from csx_RequestWindow(9F).

mp Pointer to a map_mem_page_t structure.

DESCRIPTION

This function maps the memory area on a PC Card into a page of a window allocated with the csx_RequestWindow(9F) function.

STRUCTURE

The structure members of map_mem_page_t are:

uint32_t CardOffset; /* card offset */
uint32_t Page; /* page number */

The fields are defined as follows:

CardOffset The absolute offset in bytes from the beginning of the PC Card to map into system memory.

Page Used internally by Card Services; clients must set this field to 0 before calling this function.

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_BAD_OFFSET Offset is invalid.

CS_BAD_PAGE Page is not zero.

CS_NO_CARD No PC Card in socket.

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_ModifyWindow(9F), csx_ReleaseWindow(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_ModifyConfiguration(9F)

NAME | csx_ModifyConfiguration – modify socket and PC Card Configuration Register
SYNOPSIS | #include <sys/pccard.h>
            int32_t csx_ModifyConfiguration(client_handle_t ch,
                                            modify_config_t *mc);
INTERFACE LEVEL | Solaris DDI Specific (Solaris DDI)
PARAMETERS | ch Client handle returned from csx_RegisterClient(9F).
            mc Pointer to a modify_config_t structure.
DESCRIPTION | This function allows a socket and PC Card configuration to be modified. This function
            can only modify a configuration requested via csx_RequestConfiguration(9F).
STRUCTURE MEMBERS | The structure members of modify_config_t are:
            uint32_t Socket; /* socket number */
            uint32_t Attributes; /* attributes to modify */
            uint32_t Vpp1; /* Vpp1 value */
            uint32_t Vpp2; /* Vpp2 value */

            The fields are defined as follows:
            Socket Not used in Solaris, but for portability with other Card Services
                     implementations, it should be set to the logical socket number.
            Attributes This field is bit-mapped. The following bits are defined:
                     CONF_ENABLE_IRQ_STEERING
                        Enable IRQ steering. Set to connect the PC Card IREQ line to a
                        previously selected system interrupt.
                     CONF_IRQ_CHANGE_VALID
                        IRQ change valid. Set to request the IRQ steering enable to be
                        changed.
                     CONF_VPP1_CHANGE_VALID
                        Vpp1 change valid. These bits are set to request a change to the
                        corresponding voltage level for the PC Card.
                     CONF_VPP2_CHANGE_VALID
                        Vpp2 change valid. These bits are set to request a change to the
                        corresponding voltage level for the PC Card.
                     CONF_VSOVERRIDE
                        Override VS pins. For Low Voltage keyed cards, must be set if a
                        client desires to apply a voltage inappropriate for this card to
                        any pin. After card insertion and prior to the first
                        csx_RequestConfiguration(9F) call for this client, the
                        voltage levels applied to the card will be those specified by the
                        Card Interface Specification. (See WARNINGS.)
Represent voltages expressed in tenths of a volt. Values from 0 to 25.5 volts may be set. To be valid, the exact voltage must be available from the system. To be compliant with the PC Card 95 Standard, PCMCIA/JEIDA, systems must always support 5.0 volts for both Vcc and Vpp. (See WARNINGS.)

<table>
<thead>
<tr>
<th>Vpp1, Vpp2</th>
</tr>
</thead>
</table>

RETURN VALUES

CS_SUCCESS
Successful operation.

CS_BAD_HANDLE
Client handle is invalid or csx_RequestConfiguration(9F) not done.

CS_BAD_SOCKET
Error getting/setting socket hardware parameters.

CS_BAD_VPP
Requested Vpp is not available on socket.

CS_NO_CARD
No PC Card in socket.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

SEE ALSO

csx_RegisterClient(9F), csx_ReleaseConfiguration(9F),
csx_ReleaseIO(9F), csx_ReleaseIRQ(9F), csx_RequestConfiguration(9F),
csx_RequestIO(9F), csx_RequestIRQ(9F)

PC Card 95 Standard, PCMCIA/JEIDA

WARNINGS
1. CONF_VSOVERRIDE is provided for clients that have a need to override the information provided in the CIS. The client must exercise caution when setting this as it overrides any voltage level protection provided by Card Services.

2. Using csx_ModifyConfiguration() to set Vpp to 0 volts may result in the loss of a PC Card’s state. Any client setting Vpp to 0 volts is responsible for insuring that the PC Card’s state is restored when power is re-applied to the card.

NOTES

Mapped IO addresses can only be changed by first releasing the current configuration and IO resources with csx_ReleaseConfiguration(9F) and csx_ReleaseIO(9F), requesting new IO resources and a new configuration with csx_RequestIO(9F), followed by csx_RequestConfiguration(9F).

IRQ priority can only be changed by first releasing the current configuration and IRQ resources with csx_ReleaseConfiguration(9F) and csx_ReleaseIRQ(9F), requesting new IRQ resources and a new configuration with csx_RequestIRQ(9F), followed by csx_RequestConfiguration(9F).

Vcc can not be changed using csx_ModifyConfiguration(). Vcc may be changed by first invoking csx_ReleaseConfiguration(9F), followed by csx_RequestConfiguration(9F) with a new Vcc value.
csx_ModifyWindow(9F)

**NAME**
csx_ModifyWindow – modify window attributes

**SYNOPSIS**
```
#include <sys/pccard.h>

int32_t csx_ModifyWindow(window_handle_t wh, modify_win_t *mw);
```

**INTERFACE LEVEL PARAMETERS**
- **wh**: Window handle returned from csx_RequestWindow(9F).
- **mw**: Pointer to a modify_win_t structure.

**DESCRIPTION**
This function modifies the attributes of a window allocated by the csx_RequestWindow(9F) function.

Only some of the window attributes or the access speed field may be modified by this request. The csx_MapMemPage(9F) function is also used to set the offset into PC Card memory to be mapped into system memory for paged windows. The csx_RequestWindow(9F) and csx_ReleaseWindow(9F) functions must be used to change the window base or size.

**STRUCTURE MEMBERS**
The structure members of modify_win_t are:

```
uint32_t Attributes; /* window flags */
uint32_t AccessSpeed; /* window access speed */
```

The fields are defined as follows:

- **Attributes**
  This field is bit-mapped and defined as follows:
  
  - **WIN_MEMORY_TYPE_CM**
    Window points to Common Memory area. Set this to map the window to Common Memory.
  
  - **WIN_MEMORY_TYPE_AM**
    Window points to Attribute Memory area. Set this to map the window to Attribute Memory.
  
  - **WIN_ENABLE**
    Enable Window. The client must set this to enable the window.
  
  - **WIN_ACCESS_SPEED_VALID**
    AccessSpeed valid. The client must set this when the AccessSpeed field has a value that the client wants set for the window.

- **AccessSpeed**
  The bit definitions for this field use the format of the extended speed byte of the Device ID tuple. If the mantissa is 0 (noted as reserved in the PC Card 95 Standard), the lower bits are a binary code representing a speed from the list below. Numbers in the first column are codes; items in the second column are speeds.

<table>
<thead>
<tr>
<th>Code</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved: do not use</td>
</tr>
<tr>
<td>1</td>
<td>250 nsec</td>
</tr>
</tbody>
</table>
csx_ModifyWindow(9F)

<table>
<thead>
<tr>
<th>2</th>
<th>200 nsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>150 nsec</td>
</tr>
<tr>
<td>4</td>
<td>100 nsec</td>
</tr>
<tr>
<td>5 - 7</td>
<td>Reserved: do not use</td>
</tr>
</tbody>
</table>

It is recommended that clients use the csx_ConvertSpeed(9F) function to generate the appropriate AccessSpeed values rather than manually perturbing the AccessSpeed field.

RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Window handle is invalid.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_BAD_OFFSET</td>
<td>Error getting/setting window hardware parameters.</td>
</tr>
<tr>
<td>CS_BAD_WINDOW</td>
<td>Error getting/setting window hardware parameters.</td>
</tr>
<tr>
<td>CS_BAD_SPEED</td>
<td>AccessSpeed is invalid.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_ConvertSpeed(9F), csx_MapMemPage(9F), csx_ReleaseWindow(9F),
csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_BATTERY(9F)

NAME  csx_Parse_CISTPL_BATTERY – parse the Battery Replacement Date tuple

SYNOPSIS
#include <sys/pccard.h>
int32_t csx_Parse_CISTPL_BATTERY(client_handle_t ch, tuple_t *tu, cistpl_battery_t *cb);

INTERFACE LEVEL PARAMETERS
Solaris DDI Specific (Solaris DDI)

ch  Client handle returned from csx_RegisterClient(9F).

tu  Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cb  Pointer to a cistpl_battery_t structure which contains the parsed CISTPL_BATTERY tuple information upon return from this function.

DESCRIPTION
This function parses the Battery Replacement Date tuple, CISTPL_BATTERY, into a form usable by PC Card drivers.

The CISTPL_BATTERY tuple is an optional tuple which shall be present only in PC Cards with battery-backed storage. It indicates the date on which the battery was replaced, and the date on which the battery is expected to need replacement. Only one CISTPL_BATTERY tuple is allowed per PC Card.

STRUCTURE MEMBERS
The structure members of cistpl_battery_t are:

uint32_t rday; /* date battery last replaced */
uint32_t xday; /* date battery due for replacement */

The fields are defined as follows:

rday  This field indicates the date on which the battery was last replaced.

xday  This field indicates the date on which the battery should be replaced.

RETURN VALUES
CS_SUCCESS  Successful operation.
CS_BAD_HANDLE  Client handle is invalid.
CS_UNKNOWN_TUPLE  Parser does not know how to parse tuple.
CS_NO_CARD  No PC Card in socket.
CS_NO_CIS  No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

SEE ALSO
csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_BYTEORDER(9F)

NAME  csx_Parse_CISTPL_BYTEORDER -- parse the Byte Order tuple

SYNOPSIS  
```
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_BYTEORDER(client_handle_t ch, tuple_t *tu, cistpl_byteorder_t *cbo);
```

LEVEL  Solaris DDI Specific (Solaris DDI)

PARAMETERS  
- `ch`  Client handle returned from `csx_RegisterClient(9F)`.
- `tu`  Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- `cbo`  Pointer to a `cistpl_byteorder_t` structure which contains the parsed CISTPL_BYTEORDER tuple information upon return from this function.

DESCRIPTION  
This function parses the Byte Order tuple, CISTPL_BYTEORDER, into a form usable by PC Card drivers.

The CISTPL_BYTEORDER tuple shall only appear in a partition tuple set for a memory-like partition. It specifies two parameters: the order for multi-byte data, and the order in which bytes map into words for 16-bit cards.

STRUCTURE  
The structure members of `cistpl_byteorder_t` are:

```
uint32_t order;  /* byte order code */
uint32_t map;  /* byte mapping code */
```

The fields are defined as follows:

- **order**  This field specifies the byte order for multi-byte numeric data.
  - TPLBYTEORDER_LOW  Little endian order
  - TPLBYTEORDER_VS  Vendor specific

- **map**  This field specifies the byte mapping for 16-bit or wider cards.
  - TPLBYTEMAP_LOW  Byte zero is least significant byte
  - TPLBYTEMAP_HIGH  Byte zero is most significant byte
  - TPLBYTEMAP_VS  Vendor specific mapping

RETURN VALUES  
- CS_SUCCESS  Successful operation.
- CS_BAD_HANDLE  Client handle is invalid.
- CS_UNKNOWN_TUPLE  Parser does not know how to parse tuple.
- CS_NO_CARD  No PC Card in socket.
csx_Parse_CISTPL_BYTEORDER(9F)

| CS_NO_CIS | No Card Information Structure (CIS) PC Card. |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed. |

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_CFTABLE_ENTRY(9F)

NAME

csx_Parse_CISTPL_CFTABLE_ENTRY – parse 16-bit Card Configuration Table Entry tuple

SYNOPSIS

#include <sys/pcard.h>

int32_t csx_Parse_CISTPL_CFTABLE_ENTRY(client_handle_t ch, tuple_t *tu, cistpl_cftable_entry_t *cft);

INTERFACE LEVEL PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch     Client handle returned from csx_RegisterClient(9F).

tu     Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cft    Pointer to a cistpl_cftable_entry_t structure which contains the parsed CISTPL_CFTABLE_ENTRY tuple information upon return from this function.

DESCRIPTION

This function parses the 16 bit Card Configuration Table Entry tuple, CISTPL_CFTABLE_ENTRY, into a form usable by PC Card drivers.

The CISTPL_CFTABLE_ENTRY tuple is used to describe each possible configuration of a PC Card and to distinguish among the permitted configurations. The CISTPL_CONFIG tuple must precede all CISTPL_CFTABLE_ENTRY tuples.

STRUCTURE MEMBERS

The structure members of cistpl_cftable_entry_t are:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>uint32_t</td>
<td>valid descriptions */</td>
</tr>
<tr>
<td>ifc</td>
<td>uint32_t</td>
<td>interface description */</td>
</tr>
<tr>
<td>pin</td>
<td>uint32_t</td>
<td>values for PRR */</td>
</tr>
<tr>
<td>index</td>
<td>uint32_t</td>
<td>configuration index number */</td>
</tr>
<tr>
<td>pd</td>
<td>cistpl_cftable_entry_pd_t</td>
<td>power requirements */</td>
</tr>
<tr>
<td>speed</td>
<td>cistpl_cftable_entry_speed_t</td>
<td>device speed description */</td>
</tr>
<tr>
<td>io</td>
<td>cistpl_cftable_entry_io_t</td>
<td>device I/O map */</td>
</tr>
<tr>
<td>irq</td>
<td>cistpl_cftable_entry_irq_t</td>
<td>device IRQ utilization */</td>
</tr>
<tr>
<td>mem</td>
<td>cistpl_cftable_entry_mem_t</td>
<td>device memory space */</td>
</tr>
<tr>
<td>misc</td>
<td>cistpl_cftable_entry_misc_t</td>
<td>miscellaneous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device features */</td>
</tr>
</tbody>
</table>

The flags field is defined and bit-mapped as follows:

CISTPL_CFTABLE_TPCE_DEFAULT
This is a default configuration

CISTPL_CFTABLE_TPCE_IF
If configuration byte exists

CISTPL_CFTABLE_TPCE_FS_PWR
Power information exists

CISTPL_CFTABLE_TPCE_FS_TD
Timing information exists
csx_Parse_CISTPL_CFTABLE_ENTRY(9F)

| CISTPL_CFTABLE_TPCE_FS_IO                      | I/O information exists |
| CISTPL_CFTABLE_TPCE_FS_IRQ                     | IRQ information exists |
| CISTPL_CFTABLE_TPCE_FS_MEM                     | MEM space information exists |
| CISTPL_CFTABLE_TPCE_FS_MISC                    | MISC information exists |
| CISTPL_CFTABLE_TPCE_FS_STCE_EV                 | STCE_EV exists |
| CISTPL_CFTABLE_TPCE_FS_STCE_PD                 | STCE_PD exists |

If the `CISTPL_CFTABLE_TPCE_IF` flag is set, the `ifc` field is bit-mapped and defined as follows:

| CISTPL_CFTABLE_TPCE_IF_MEMORY                  | Memory interface |
| CISTPL_CFTABLE_TPCE_IF_IO_MEM                 | IO and memory |
| CISTPL_CFTABLE_TPCE_IF_CUSTOM_0               | Custom interface 0 |
| CISTPL_CFTABLE_TPCE_IF_CUSTOM_1               | Custom interface 1 |
| CISTPL_CFTABLE_TPCE_IF_CUSTOM_2               | Custom interface 2 |
| CISTPL_CFTABLE_TPCE_IF_CUSTOM_3               | Custom interface 3 |
| CISTPL_CFTABLE_TPCE_IF_MASK                  | Interface type mask |
| CISTPL_CFTABLE_TPCE_IF_BVD                    | BVD active in PRR |
| CISTPL_CFTABLE_TPCE_IF_WP                     | WP active in PRR |
| CISTPL_CFTABLE_TPCE_IF_RDY                    | RDY active in PRR |
| CISTPL_CFTABLE_TPCE_IF_MWAIT                 | WAIT - mem cycles |

`pin` is a value for the Pin Replacement Register.
index is a configuration index number.

The structure members of cistpl_cftable_entry_pd_t are:

```c
uint32_t flags; /* which descriptions are valid */
cistpl_cftable_entry_pwr_t pd_vcc; /* VCC power description */
cistpl_cftable_entry_pwr_t pd_vpp1; /* Vpp1 power description */
cistpl_cftable_entry_pwr_t pd_vpp2; /* Vpp2 power description */
```

This flags field is bit-mapped and defined as follows:

- `CISTPL_CFTABLE_TPCE_FS_PWR_VCC`: Vcc description valid
- `CISTPL_CFTABLE_TPCE_FS_PWR_VPP1`: Vpp1 description valid
- `CISTPL_CFTABLE_TPCE_FS_PWR_VPP2`: Vpp2 description valid

The structure members of cistpl_cftable_entry_pwr_t are:

```c
uint32_t nomV; /* nominal supply voltage */
uint32_t nomV_flags;
uint32_t minV; /* minimum supply voltage */
uint32_t minV_flags;
uint32_t maxV; /* maximum supply voltage */
uint32_t maxV_flags;
uint32_t staticI; /* continuous supply current */
uint32_t staticI_flags;
uint32_t avgI; /* max current required averaged over 1 sec. */
uint32_t avgI_flags;
uint32_t peakI; /* max current required averaged over 10mS */
uint32_t peakI_flags;
uint32_t pdownI; /* power down supply current required */
uint32_t pdownI_flags;
```

nomV, minV, maxV, staticI, avgI, peakI_flag, and pdownI are defined and bit-mapped as follows:

- `CISTPL_CFTABLE_PD_NOMV`: Nominal supply voltage
- `CISTPL_CFTABLE_PD_MINV`: Minimum supply voltage
- `CISTPL_CFTABLE_PD_MAXV`: Maximum supply voltage
- `CISTPL_CFTABLE_PD_STATICI`: Continuous supply current
- `CISTPL_CFTABLE_PD_AVGI`: Maximum current required averaged over 1 second
- `CISTPL_CFTABLE_PD_PEAKI`: Maximum current required averaged over 10mS
CISTPL_CFTABLE_PD_PDOWNI
Power down supply current required
nomV_flags, minV_flags, maxV_flags, staticI_flags, avgI_flags,
peakI_flags, and pdownI_flags are defined and bit-mapped as follows:

CISTPL_CFTABLE_PD_EXISTS
This parameter exists

CISTPL_CFTABLE_PD_MUL10
Multiply return value by 10

CISTPL_CFTABLE_PD_NC_SLEEP
No connection on sleep/power down

CISTPL_CFTABLE_PD_ZERO
Zero value required

CISTPL_CFTABLE_PD_NC
No connection ever

The structure members of cistpl_cftable_entry_speed_t are:

```c
uint32_t flags; /* which timing information is present */
uint32_t wait; /* max WAIT time in device speed format */
uint32_t nS_wait; /* max WAIT time in nS */
uint32_t rdybsy; /* max RDY/BSY time in device speed format */
uint32_t nS_rdybsy; /* max RDY/BSY time in nS */
uint32_t rsvd; /* max RSVD time in device speed format */
uint32_t nS_rsvd; /* max RSVD time in nS */
```

The flags field is bit-mapped and defined as follows:

CISTPL_CFTABLE_TPCE_FS_TD_WAIT
WAIT timing exists

CISTPL_CFTABLE_TPCE_FS_TD_RDY
RDY/BSY timing exists

CISTPL_CFTABLE_TPCE_FS_TD_RSVD
RSVD timing exists

The structure members of cistpl_cftable_entry_io_t are:

```c
uint32_t flags; /* direct copy of TPCE_IO byte in tuple */
uint32_t addr_lines; /* number of decoded I/O address lines */
uint32_t ranges; /* number of I/O ranges */
cistpl_cftable_entry_io_range_t
    range[CISTPL_CFTABLE_ENTRY_MAX_IO_RANGES];
```

The flags field is defined and bit-mapped as follows:

CISTPL_CFTABLE_TPCE_FS_IO_BUS
Bus width mask

CISTPL_CFTABLE_TPCE_FS_IO_BUS8
8-bit flag
The structure members of \texttt{cistpl_cftable_entry_io_range_t} are:

\begin{verbatim}
uint32_t addr; /* I/O start address */
uint32_t length; /* I/O register length */
\end{verbatim}

The structure members of \texttt{cistpl_cftable_entry_irq_t} are:

\begin{verbatim}
uint32_t flags; /* direct copy of TPCE_IR byte in tuple */
uint32_t irqs; /* bit mask for each allowed IRQ */
\end{verbatim}

The structure members of \texttt{cistpl_cftable_entry_mem_t} are:

\begin{verbatim}
uint32_t flags; /* memory descriptor type and host addr info */
uint32_t windows; /* number of memory space descriptors */
cistpl_cftable_entry_mem_window_t window[CISTPL_CFTABLE_ENTRY_MAX_MEM_WINDOWS];
\end{verbatim}

The \texttt{flags} field is defined and bit-mapped as follows:

- CISTPL_CFTABLE_TPCE_FS_MEM3
  - Space descriptors
- CISTPL_CFTABLE_TPCE_FS_MEM2
  - host_addr=card_addr
- CISTPL_CFTABLE_TPCE_FS_MEM1
  - Card address=0 any host address
- CISTPL_CFTABLE_TPCE_FS_MEM_HOST
  - If host address is present in MEM3

The structure members of \texttt{cistpl_cftable_entry_mem_window_t} are:

\begin{verbatim}
uint32_t length; /* length of this window */
uint32_t card_addr; /* card address */
uint32_t host_addr; /* host address */
\end{verbatim}

The structure members of \texttt{cistpl_cftable_entry_misc_t} are:

\begin{verbatim}
uint32_t flags; /* miscellaneous features flags */
\end{verbatim}

The \texttt{flags} field is defined and bit-mapped as follows:

- CISTPL_CFTABLE_TPCE_MI_MTC_MASK
  - Max twin cards mask
- CISTPL_CFTABLE_TPCE_MI_AUDIO
  - Audio on BVD2
- CISTPL_CFTABLE_TPCE_MI_READONLY
  - R/O storage
csx_Parse_CISTPL_CFTABLE_ENTRY(9F)

CISTPL_CFTABLE_TPCE_MI_PWRDOWN
   Powerdown capable

CISTPL_CFTABLE_TPCE_MI_DRQ_MASK
   DMAREQ mask

CISTPL_CFTABLE_TPCE_MI_DRQ_SPK
   DMAREQ on SPKR

CISTPL_CFTABLE_TPCE_MI_DRQ_IOIS
   DMAREQ on IOIS16

CISTPL_CFTABLE_TPCE_MI_DRQ_INP
   DMAREQ on INPACK

CISTPL_CFTABLE_TPCE_MI_DMA_8
   DMA width 8 bits

CISTPL_CFTABLE_TPCE_MI_DMA_16
   DMA width 16 bits

RETURN VALUES

CS_SUCCESS
   Successful operation.

CS_BAD_HANDLE
   Client handle is invalid.

CS_UNKNOWN_TUPLE
   Parser does not know how to parse tuple.

CS_NO_CARD
   No PC Card in socket.

CS_NO_CIS
   No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION
   No PCMCIA hardware installed.

CONTEXT
   This function may be called from user or kernel context.

SEE ALSO
   csx_GetFirstTuple(9F), csx_GetTupleData(9F),
   csx_Parse_CISTPL_CONFIG(9F), csx_RegisterClient(9F),
   csxValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_CONFIG(9F)

NAME

csx_Parse_CISTPL_CONFIG – parse Configuration tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_CONFIG(client_handle_t ch, tuple_t *tu, cistpl_config_t *cc);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cc Pointer to a cistpl_config_t structure which contains the parsed CISTPL_CONFIG tuple information upon return from this function.

DESCRIPTION

This function parses the Configuration tuple, CISTPL_CONFIG, into a form usable by PC Card drivers. The CISTPL_CONFIG tuple is used to describe the general characteristics of 16-bit PC Cards containing I/O devices or using custom interfaces. It may also describe PC Cards, including Memory Only cards, which exceed nominal power supply specifications, or which need descriptions of their power requirements or other information.

STRUCTURE

MEMBERS

The structure members of cistpl_config_t are:

uint32_t present; /* register present flags */
uint32_t nr; /* number of config registers found */
uint32_t hr; /* highest config register index found */
uint32_t regs[CISTPL_CONFIG_MAX_CONFIG_REGS]; /* reg offsets */
uint32_t base; /* base offset of config registers */
uint32_t last; /* last config index */

The fields are defined as follows:

present This field indicates which configuration registers are present on the PC Card.

CONFIGOPTION_REG_PRESENT
    Configuration Option Register present

CONFIGSTATUS_REG_PRESENT
    Configuration Status Register present

CONFIG_PINREPL_REG_PRESENT
    Pin Replacement Register present

CONFIG_COPY_REG_PRESENT
    Copy Register present

CONFIG_EXSTAT_REG_PRESENT
    Extended Status Register present

CONFIG_IOBASE0_REG_PRESENT
    IO Base 0 Register present

Kernel Functions for Drivers 155
config_io1b_reg_present
IO Base 1 Register present

config_io2b_reg_present
IO Base2 Register present

config_io3b_reg_present
IO Base3 Register present

config_io_limit_reg_present
IO Limit Register present

nr
This field specifies the number of configuration registers that are present on the PC Card.

hr
This field specifies the highest configuration register number that is present on the PC Card.

regs
This array contains the offset from the start of Attribute Memory space for each configuration register that is present on the PC Card. If a configuration register is not present on the PC Card, the value in the corresponding entry in the regs array is undefined.

base
This field contains the offset from the start of Attribute Memory space to the base of the PC Card configuration register space.

last
This field contains the value of the last valid configuration index for this PC Card.

RETURN VALUES
CS_SUCCESS
Successful operation.

CS_BAD_HANDLE
Client handle is invalid.

CS_UNKNOWN_TUPLE
Parser does not know how to parse tuple.

CS_NO_CARD
No PC Card in socket.

CS_NO_CIS
No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

SEE ALSO
csx_get_first_tuple(9F), csx_get_tuple_data(9F),
csx_parse_cistpl_cftable_entry(9F), csx_register_client(9F),
csx_validate_cis(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
PC Card drivers should not attempt to use configurations beyond the "last" member in the cistpl_config_t structure.
csx_Parse_CISTPL_DATE(9F)

NAME | csx_Parse_CISTPL_DATE – parse the Card Initialization Date tuple

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_DATE(client_handle_t ch, tuple_t *tu,
                              cistpl_date_t *cd);

INTERFACE LEVEL

PARAMETERS

ch | Client handle returned from csx_RegisterClient(9F).

tu | Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cd | Pointer to a cistpl_date_t structure which contains the parsed
    CISTPL_DATE tuple information upon return from this function.

DESCRIPTION

This function parses the Card Initialization Date tuple, CISTPL_DATE, into a form
usable by PC Card drivers.

The CISTPL_DATE tuple is an optional tuple. It indicates the date and time at which
the card was formatted. Only one CISTPL_DATE tuple is allowed per PC Card.

STRUCTURE MEMBERS

The structure members of cistpl_date_t are:

uint32_t time;
uint32_t day

The fields are defined as follows:

- **time** | This field indicates the time at which the PC Card was initialized.
- **day** | This field indicates the date the PC Card was initialized.

RETURN VALUES

- **CS_SUCCESS** | Successful operation.
- **CS_BAD_HANDLE** | Client handle is invalid.
- **CS_UNKNOWN_TUPLE** | Parser does not know how to parse tuple.
- **CS_NO_CARD** | No PC Card in socket.
- **CS_NO_CIS** | No Card Information Structure (CIS) on PC Card.
- **CS_UNSUPPORTED_FUNCTION** | No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_DEVICE(9F)

NAME

csx_Parse_CISTPL_DEVICE, csx_Parse_CISTPL_DEVICE_A, csx_Parse_CISTPL_DEVICE_OC, csx_Parse_CISTPL_DEVICE_OA – parse Device Information tuples

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_DEVICE(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_A(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OC(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OA(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch  Client handle returned from csx_RegisterClient(9F).

tu  Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cd  Pointer to a cistpl_device_t structure which contains the parsed CISTPL_DEVICE, CISTPLDEVICE_A, CISTPLDEVICE_OC, or CISTPLDEVICE_OA tuple information upon return from these functions, respectively.

DESCRIPTION

csx_Parse_CISTPL_DEVICE() and csx_Parse_CISTPLDEVICE_A() parse the 5 volt Device Information tuples, CISTPLDEVICE and CISTPLDEVICE_A, respectively, into a form usable by PC Card drivers.

csx_Parse_CISTPLDEVICE_OC() and csx_Parse_CISTPLDEVICE_OA() parse the Other Condition Device Information tuples, CISTPLDEVICE_OC and CISTPLDEVICE_OA, respectively, into a form usable by PC Card drivers.

The CISTPLDEVICE and CISTPLDEVICE_A tuples are used to describe the card’s device information, such as device speed, device size, device type, and address space layout information for Common Memory or Attribute Memory space, respectively.

The CISTPLDEVICE_OC and CISTPLDEVICE_OA tuples are used to describe the information about the card’s device under a set of operating conditions for Common Memory or Attribute Memory space, respectively.

STRUCTURE MEMBERS

The structure members of cistpl_device_t are:

uint32_t num_devices; /* number of devices found */
cistpl_device_node_t devnode[CISTPLDEVICE_MAX_DEVICES];

The structure members of cistpl_device_node_t are:
The fields are defined as follows:

flags

This field indicates whether or not the device is writable, and describes a Vcc voltage at which the PC Card can be operated.

CISTPL_DEVICE_WPS
  Write Protect Switch bit is set

Bits which are applicable only for CISTPL_DEVICE_OC and CISTPL_DEVICE_OA are:

CISTPL_DEVICE_OC_MWAIT
  Use MWAIT

CISTPL DEVICE OC Vcc MASK
  Mask for Vcc value

CISTPL_DEVICE_OC Vcc5
  5.0 volt operation

CISTPL_DEVICE_OC Vcc3
  3.3 volt operation

CISTPL_DEVICE_OC VccXX
  X.X volt operation

CISTPL_DEVICE_OC VccYY
  Y.Y volt operation

speed

The device speed value described in the device speed code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the speed information will be ignored.

nS_speed

The device speed value described in nanosecond units.

size

The device size value described in the device size code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the size information will be ignored.

size_in_bytes

The device size value described in byte units.

type

This is the device type code field which is defined as follows:
CISTPL DEVICE_DTYPE_NULL
   No device
CISTPL DEVICE_DTYPE_ROM
   Masked ROM
CISTPL DEVICE_DTYPE_OTPROM
   One Time Programmable ROM
CISTPL DEVICE_DTYPE_EEPROM
   UV EPROM
CISTPL DEVICE_DTYPE_EEPROM
   EEPROM
CISTPL DEVICE_DTYPE_FLASH
   FLASH
CISTPL DEVICE_DTYPE_SRAM
   Static RAM
CISTPL DEVICE_DTYPE_DRAM
   Dynamic RAM
CISTPL DEVICE_DTYPE_FUNCSPEC
   Function-specific memory address range
CISTPL DEVICE_DTYPE_EXTEND
   Extended type follows

RETURN VALUES
   CS_SUCCESS                Successful operation.
   CS_BAD_HANDLE            Client handle is invalid.
   CS_UNKNOWN_TUPLE         Parser does not know how to parse tuple.
   CS_NO_CARD               No PC Card in socket.
   CS_NO_CIS                No Card Information Structure (CIS) on PC Card.
   CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

CONTEXT
   These functions may be called from user or kernel context.

SEE ALSO
   csx_GetFirstTuple(9F), csx_GetTupleData(9F),
   csx_Parse_CISTPL_JEDEC_C(9F), csx_RegisterClient(9F),
   csx.ValidateCIS(9F), tuple(9S)

   PC Card 95 Standard, PCMCIA/JEIDA
NAME  csx_Parse_CISTPL_DEVICE, csx_Parse_CISTPLDEVICE_A, csx_Parse_CISTPLDEVICE_OC, csx_Parse_CISTPLDEVICE_OA – parse Device Information tuples

SYNOPSIS  
#include <sys/pccard.h>

int32_t csx_Parse_CISTPLDEVICE(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);
int32_t csx_Parse_CISTPLDEVICE_A(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);
int32_t csx_Parse_CISTPLDEVICE_OC(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);
int32_t csx_Parse_CISTPLDEVICE_OA(client_handle_t ch, tuple_t *tu, cistpl_device_t *cd);

INTERFACE LEVEL PARAMETERS  
Solaris DDI Specific (Solaris DDI)
ch  Client handle returned from csx_RegisterClient(9F).
tu  Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
cd  Pointer to a cistpl_device_t structure which contains the parsed CISTPLDEVICE, CISTPLDEVICE_A, CISTPLDEVICE_OC, or CISTPLDEVICE_OA tuple information upon return from these functions, respectively.

DESCRIPTION  
csx_Parse_CISTPLDEVICE() and csx_Parse_CISTPLDEVICE_A() parse the 5 volt Device Information tuples, CISTPLDEVICE and CISTPLDEVICE_A, respectively, into a form usable by PC Card drivers.

csx_Parse_CISTPLDEVICE_OC() and csx_Parse_CISTPLDEVICE_OA() parse the Other Condition Device Information tuples, CISTPLDEVICE_OC and CISTPLDEVICE_OA, respectively, into a form usable by PC Card drivers.

The CISTPLDEVICE and CISTPLDEVICE_A tuples are used to describe the card’s device information, such as device speed, device size, device type, and address space layout information for Common Memory or Attribute Memory space, respectively.

The CISTPLDEVICE_OC and CISTPLDEVICE_OA tuples are used to describe the information about the card’s device under a set of operating conditions for Common Memory or Attribute Memory space, respectively.

STRUCTURE MEMBERS  
The structure members of cistpl_device_t are:

uint32_t num_devices; /* number of devices found */
cistpl_device_node_t devnode[CISTPLDEVICE_MAX_DEVICES];

The structure members of cistpl_device_node_t are:
The fields are defined as follows:

**flags**
This field indicates whether or not the device is writable, and describes a Vcc voltage at which the PC Card can be operated.

- **CISTPL_DEVICE_WPS**
  Write Protect Switch bit is set

- Bits which are applicable only for CISTPL_DEVICE_OC and CISTPL_DEVICE_OA are:
  - **CISTPLDEVICEOC_MWAIT**
    Use MWAIT
  - **CISTPLDEVICEOC_Vcc_MASK**
    Mask for Vcc value
  - **CISTPLDEVICEOC_Vcc5**
    5.0 volt operation
  - **CISTPLDEVICEOC_Vcc33**
    3.3 volt operation
  - **CISTPLDEVICEOC_VccXX**
    X.X volt operation
  - **CISTPLDEVICEOC_VccYY**
    Y.Y volt operation

**speed**
The device speed value described in the device speed code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the speed information will be ignored.

**nS_speed**
The device speed value described in nanosecond units.

**size**
The device size value described in the device size code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the size information will be ignored.

**size_in_bytes**
The device size value described in byte units.

**type**
This is the device type code field which is defined as follows:
csx_Parse_CISTPL_DEVICE_A(9F)

CISTPL_DEVICE_DTYPE_NULL
   No device
CISTPL_DEVICE_DTYPE_ROM
   Masked ROM
CISTPL_DEVICE_DTYPE_OTPROM
   One Time Programmable ROM
CISTPL_DEVICE_DTYPE_EPROM
   UV EPROM
CISTPL_DEVICE_DTYPE_EEPROM
   EEPROM
CISTPL_DEVICE_DTYPE_FLASH
   FLASH
CISTPLDEVICE_DTYPE_SRAM
   Static RAM
CISTPLDEVICE_DTYPE_DRAM
   Dynamic RAM
CISTPLDEVICE_DTYPE_FUNCSPEC
   Function-specific memory address range
CISTPLDEVICE_DTYPE_EXTEND
   Extended type follows

RETURN VALUES
CS_SUCCESS
   Successful operation.
CS_BAD_HANDLE
   Client handle is invalid.
CS_UNKNOWN_TUPLE
   Parser does not know how to parse tuple.
CS_NO_CARD
   No PC Card in socket.
CS_NO_CIS
   No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION
   No PCMCIA hardware installed.

CONTEXT
These functions may be called from user or kernel context.

SEE ALSO
 csx_GetFirstTuple(9F), csx_GetTupleData(9F),
 csx_Parse_CISTPL_JEDEC_C(9F), csx_RegisterClient(9F),
 csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPLDEVICEGEO(9F)

NAME
csx_Parse_CISTPLDEVICEGEO – parse the Device Geo tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPLDEVICEGEO(client_handle_t ch, tuple_t *tp, cistpl_devicegeo_t *pt);

INTERFACE
Solaris DDI Specific (Solaris DDI)

LEVEL
PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tp Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

pt Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo tuple information upon return from this function.

DESCRIPTION

This function parses the Device Geo tuple, CISTPLDEVICEGEO, into a form usable by PC Card drivers.

The CISTPLDEVICEGEO tuple describes the device geometry of common memory partitions.

STRUCTURE

The structure members of cistpl_devicegeo_t are:

uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].bus;
uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].ebs;
uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].rbs;
uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].wbs;
uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].part;
uint32_t info[CISTPLDEVICEGEO_MAX_PARTITIONS].hwil;

The fields are defined as follows:

info[CISTPLDEVICEGEO_MAX_PARTITIONS].bus This field indicates the card interface width in bytes for the given partition.

info[CISTPLDEVICEGEO_MAX_PARTITIONS].ebs This field indicates the minimum erase block size for the given partition.

info[CISTPLDEVICEGEO_MAX_PARTITIONS].rbs This field indicates the minimum read block size for the given partition.

info[CISTPLDEVICEGEO_MAX_PARTITIONS].wbs This field indicates the minimum write block size for the given partition.

info[CISTPLDEVICEGEO_MAX_PARTITIONS].part This field indicates the segment partition subdivisions for the given partition.

info[CISTPLDEVICEGEO_MAX_PARTITIONS].hwil This field indicates the hardware interleave

RETURN VALUES

CS_SUCCESS Successful operation.
csx_Parse_CISTPL_DEVICEGEO(9F)

<table>
<thead>
<tr>
<th>CS_BAD_HANDLE</th>
<th>Client handle is invalid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

csx_GetFirstTuple(9F), csx_GetNextTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_DEVICEGEO_A(9F), csx_RegisterClient(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_DEVICEGEO_A

**NAME**
csx_Parse_CISTPL_DEVICEGEO_A – parse the Device Geo A tuple

**SYNOPSIS**
```
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_DEVICEGEO_A(client_handle_t ch, tuple_t *tp, cistpl_devicegeo_t *pt);
```

**INTERFACE LEVEL PARAMETERS**
- **ch** Client handle returned from csx_RegisterClient(9F).
- **tp** Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
- **pt** Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo A tuple information upon return from this function.

**DESCRIPTION**
This function parses the Device Geo A tuple, CISTPL_DEVICEGEO_A, into a form usable by PC Card drivers.

The CISTPL_DEVICEGEO_A tuple describes the device geometry of attribute memory partitions.

**STRUCTURE MEMBERS**
The structure members of cistpl_devicegeo_t are:
```
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil;
```

The fields are defined as follows:
- **info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus** This field indicates the card interface width in bytes for the given partition.
- **info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs** This field indicates the minimum erase block size for the given partition.
- **info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs** This field indicates the minimum read block size for the given partition.
- **info[CISTPL DEVICEGEO_MAX_PARTITIONS].wbs** This field indicates the minimum write block size for the given partition.
- **info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part** This field indicates the segment partition subdivisions for the given partition.
- **info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil** This field indicates the hardware interleave for the given partition.

**RETURN VALUES**
- **CS_SUCCESS** Successful operation.
csx_Parse_CISTPLDEVICEGEO_A(9F)

<table>
<thead>
<tr>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_BAD_HANDLE</td>
</tr>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
</tr>
</tbody>
</table>

**CS_BAD_HANDLE**
Client handle is invalid.

**CS_UNKNOWN_TUPLE**
Parser does not know how to parse tuple.

**CS_NO_CARD**
No PC Card in socket.

**CS_NO_CIS**
No Card Information Structure (CIS) on PC Card.

**CS_UNSUPPORTED_FUNCTION**
No PCMCIA hardware installed.

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
- csx_GetFirstTuple(9F), csx_GetNextTuple(9F), csx_GetTupleData(9F),
- csx_Parse_CISTPLDEVICEGEO(9F), csx_RegisterClient(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_DEVICE_A(9F)

**NAME**

csx(Parse) CISTPL_DEVICE, csx(Parse) CISTPL_DEVICE_A,

**SYNOPSIS**

```c
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_DEVICE(client_handle_t ch, tuple_t *tu,
cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_A(client_handle_t ch, tuple_t *tu,
cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OC(client_handle_t ch, tuple_t
*tu, cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OA(client_handle_t ch, tuple_t
*tu, cistpl_device_t *cd);
```

**INTERFACE LEVEL**

Solaris DDI Specific (Solaris DDI)

**PARAMETERS**

- **ch** Client handle returned from csx_RegisterClient(9F).
- **tu** Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
  csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
- **cd** Pointer to a cistpl_device_t structure which contains the parsed
  CISTPL_DEVICE, CISTPL_DEVICE_A, CISTPL_DEVICE_OC, or
  CISTPL_DEVICE_OA tuple information upon return from these functions,
  respectively.

**DESCRIPTION**

csx_Parse_CISTPL_DEVICE() and csx_Parse_CISTPL_DEVICE_A() parse the 5
volt Device Information tuples, CISTPL_DEVICE and CISTPL_DEVICE_A,
respectively, into a form usable by PC Card drivers.

csx_Parse_CISTPL_DEVICE_OC() and csx_Parse_CISTPL_DEVICE_OA() parse
the Other Condition Device Information tuples, CISTPL_DEVICE_OC and
CISTPL_DEVICE_OA, respectively, into a form usable by PC Card drivers.

The CISTPL_DEVICE and CISTPL_DEVICE_A tuples are used to describe the card’s
device information, such as device speed, device size, device type, and address space
layout information for Common Memory or Attribute Memory space, respectively.

The CISTPL_DEVICE_OC and CISTPL_DEVICE_OA tuples are used to describe the
information about the card’s device under a set of operating conditions for Common
Memory or Attribute Memory space, respectively.

**STRUCTURE MEMBERS**

The structure members of cistpl_device_t are:

```c
uint32_t num_devices; /* number of devices found */
cistpl_device_node_t devnode[CISTPL_DEVICE_MAX_DEVICES];
```

The structure members of cistpl_device_node_t are:

Kernel Functions for Drivers 169
The fields are defined as follows:

**flags**
This field indicates whether or not the device is writable, and describes a Vcc voltage at which the PC Card can be operated.

- **CISTPL_DEVICE_WPS**
  Write Protect Switch bit is set

  Bits which are applicable only for CISTPL_DEVICE_OC and CISTPL_DEVICE_OA are:

  - **CISTPL_DEVICE_OC_MWAIT**
    Use MWAIT
  - **CISTPL_DEVICE_OC_Vcc_MASK**
    Mask for Vcc value
  - **CISTPL_DEVICE_OC_Vcc5**
    5.0 volt operation
  - **CISTPL_DEVICE_OC_Vcc33**
    3.3 volt operation
  - **CISTPL_DEVICE_OC_VccXX**
    X.X volt operation
  - **CISTPLDEVICE_OC_VccYY**
    Y.Y volt operation

**speed**
The device speed value described in the device speed code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the speed information will be ignored.

**nS_speed**
The device speed value described in nanosecond units.

**size**
The device size value described in the device size code unit. If this field is set to CISTPL_DEVICE_SPEED_SIZE_IGNORE, then the size information will be ignored.

**size_in_bytes**
The device size value described in byte units.

**type**
This is the device type code field which is defined as follows:
CS_SUCCESS
No device
CS_BAD_HANDLE
Masked ROM
CS_UNKNOWN_TUPLE
One Time Programmable ROM
CS_UNSUPPORTED_FUNCTION
UV EPROM
CS_NO_CARD
One Time Programmable EPROM
CS_NO_CIS
EEPROM
CS_UNSUPPORTED_FUNCTION
Extended type follows
FUNCTION SPECIFIC MEMORY ADDRESS RANGE
RETURN VALUES
Successful operation.
CS_BAD_HANDLE
Parser does not know how to parse tuple.
CS_NO_CARD
Extended type follows
No PC Card in socket.
CS_NO_CIS
No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.
CONTEXT
These functions may be called from user or kernel context.
SEE ALSO
csx_GetFirstTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_JEDEC_C(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)
PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_DEVICE(9F)

NAME csx_Parse_CISTPL_DEVICE, csx_Parse_CISTPL DEVICE_A,
 csx_Parse_CISTPL_DEVICE OC, csx_Parse_CISTPL_DEVICE_OA – parse Device
 Information tuples

SYNOPSIS #include <sys/pccard.h>

int32_t csx_Parse_CISTPL_DEVICE(client_handle_t ch, tuple_t *tu,
cistpl_device_t *cd);
int32_t csx_Parse_CISTPL DEVICE_A(client_handle_t ch, tuple_t *tu,
cistpl_device_t *cd);
int32_t csx_Parse_CISTPL DEVICE OC(client_handle_t ch, tuple_t
 *tu, cistpl_device_t *cd);
int32_t csx_Parse_CISTPL DEVICE_OA(client_handle_t ch, tuple_t
 *tu, cistpl_device_t *cd);

INTERFACE LEVEL
PARAMETERS Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cd Pointer to a cistpl_device_t structure which contains the parsed
CISTPL DEVICE, CISTPL DEVICE A, CISTPL DEVICE OC, or
CISTPL DEVICE OA tuple information upon return from these functions,
respectively.

DESCRIPTION csx_Parse_CISTPL DEVICE() and csx_Parse_CISTPL_DEVICE_A() parse the 5
volt Device Information tuples, CISTPL DEVICE and CISTPL DEVICE_A,
respectively, into a form usable by PC Card drivers.

csx_Parse_CISTPL DEVICE OC() and csx_Parse_CISTPLDEVICE OA() parse
the Other Condition Device Information tuples, CISTPL DEVICE OC and
CISTPL DEVICE OA, respectively, into a form usable by PC Card drivers.

The CISTPL DEVICE and CISTPL DEVICE_A tuples are used to describe the card’s
device information, such as device speed, device size, device type, and address space
layout information for Common Memory or Attribute Memory space, respectively.

The CISTPL DEVICE OC and CISTPL DEVICE OA tuples are used to describe the
information about the card’s device under a set of operating conditions for Common
Memory or Attribute Memory space, respectively.

STRUCTURE MEMBERS The structure members of cistpl_device_t are:

uint32_t num_devices; /* number of devices found */
cistpl_device_node_t devnode[CISTPL DEVICE_MAX_DEVICES];

The structure members of cistpl_device_node_t are:
The fields are defined as follows:

**flags**

This field indicates whether or not the device is writable, and describes a Vcc voltage at which the PC Card can be operated.

- **CISTPL_DEVICE_WPS**
  - Write Protect Switch bit is set
- Bits which are applicable only for **CISTPL_DEVICE_OC** and **CISTPL_DEVICE_OA** are:
  - **CISTPL_DEVICE_OC_MWAIT**
    - Use MWAIT
  - **CISTPL_DEVICE_OC_Vcc_MASK**
    - Mask for Vcc value
  - **CISTPL_DEVICE_OC_Vcc5**
    - 5.0 volt operation
  - **CISTPL_DEVICE_OC_Vcc33**
    - 3.3 volt operation
  - **CISTPL_DEVICE_OC_VccXX**
    - X.X volt operation
  - **CISTPL_DEVICE_OC_VccYY**
    - Y.Y volt operation

**speed**

The device speed value described in the device speed code unit. If this field is set to **CISTPLDEVICE_SPEED_SIZE_IGNORE**, then the speed information will be ignored.

**nS_speed**

The device speed value described in nanosecond units.

**size**

The device size value described in the device size code unit. If this field is set to **CISTPLDEVICE_SPEED_SIZE_IGNORE**, then the size information will be ignored.

**size_in_bytes**

The device size value described in byte units.

**type**

This is the device type code field which is defined as follows:
csx_Parse_CISTPL_DEVICE_OC(9F)

CISTPL_DEVICE_DTYPE_NULL
No device
CISTPL_DEVICE_DTYPE_ROM
Masked ROM
CISTPL_DEVICE_DTYPE_OTPROM
One Time Programmable ROM
CISTPL_DEVICE_DTYPE_EPROM
UV EPROM
CISTPL_DEVICE_DTYPE_EEPROM
EEPROM
CISTPL_DEVICE_DTYPE_FLASH
FLASH
CISTPL_DEVICE_DTYPE_SRAM
Static RAM
CISTPL_DEVICE_DTYPE_DRAM
Dynamic RAM
CISTPL_DEVICE_DTYPE_FUNCSPEC
Function-specific memory address range
CISTPL_DEVICE_DTYPE_EXTEND
Extended type follows

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.
CS_NO_CARD No PC Card in socket.
CS_NO_CIS No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT These functions may be called from user or kernel context.

SEE ALSO csx_GetFirstTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_JEDEC_C(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_FORMAT – parse the Data Recording Format tuple

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_FORMAT(client_handle_t ch, tuple_t *tu, cistpl_format_t *pt);

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

pt Pointer to a cistpl_format_t structure which contains the parsed CISTPL_FORMAT tuple information upon return from this function.

This function parses the Data Recording Format tuple, CISTPL_FORMAT, into a form usable by PC Card drivers.

The CISTPL_FORMAT tuple indicates the data recording format for a device partition.

The structure members of cistpl_format_t are:

type;
edc_length;
edc_type;
offset;
nbytes;
dev.disk.bksize;
dev.disk.nblocks;
dev.disk.edcloc;
dev.mem.flags;
dev.mem.reserved;
dev.mem.address;
dev.mem.edcloc;

The fields are defined as follows:

type This field indicates the type of device:

TPLFMTTYPE_DISK disk-like device

TPLFMTTYPE_MEM memory-like device

TPLFMTTYPE_VS vendor-specific device

edc_length This field indicates the error detection code length.
edc_type This field indicates the error detection code type.
offset This field indicates the offset of the first byte of data in this partition.
csx_Parse_CISTPL_FORMAT(9F)

nbytes This field indicates the number of bytes of data in this partition

dev.disk.bksize This field indicates the block size, for disk devices.
dev.disk.nblocks This field indicates the number of blocks, for disk devices.
dev.disk.edcloc This field indicates the location of the error detection code, for disk devices.
dev.mem.flags This field provides flags, for memory devices. Valid flags are:

TPLFMTFLAGS_ADDR address is valid

TPLFMTFLAGS_AUTO automatically map memory region

dev.mem.reserved This field is reserved.
dev.mem.address This field indicates the physical address, for memory devices.
dev.mem.edcloc This field indicates the location of the error detection code, for memory devices.

RETURN VALUES

CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.
CS_NO_CARD No PC Card in socket.
CS_NO_CIS No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT This function may be called from user or kernel context.

SEE ALSO csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_FUNCE – parse Function Extension tuple

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_FUNCE(client_handle_t ch, tuple_t *tu, cistpl_funce_t *cf, uint32_t fid);

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cf Pointer to a cistpl_funce_t structure which contains the parsed CISTPL_FUNCE tuple information upon return from this function.

fid The function ID code to which this CISTPL_FUNCE tuple refers. See csx_Parse_CISTPL_FUNCID(9F).

This function parses the Function Extension tuple, CISTPL_FUNCE, into a form usable by PC Card drivers.

The CISTPL_FUNCE tuple is used to describe information about a specific PCCard function. The information provided is determined by the Function Identification tuple, CISTPL_FUNCID, that is being extended. Each function has a defined set of extension tuples.

The structure members of cistpl_funce_t are:

uint32_t function; /* type of extended data */
uint32_t subfunction;
union {
    struct serial {
        uint32_t ua; /* UART in use */
        uint32_t uc; /* UART capabilities */
    } serial;
    struct modem {
        uint32_t fc; /* supported flow control methods */
        uint32_t cb; /* size of DCE command buffer */
        uint32_t eb; /* size of DCE to DCE buffer */
        uint32_t tb; /* size of DTE to DCE buffer */
    } modem;
    struct data_modem {
        uint32_t ud; /* highest data rate */
        uint32_t ms; /* modulation standards */
        uint32_t em; /* err correct proto and */
        /* non-CCITT modulation */
        uint32_t dc; /* data compression protocols */
        uint32_t cm; /* command protocols */
        uint32_t ex; /* escape mechanisms */
        uint32_t dy; /* standardized data encryption */
        uint32_t ef; /* miscellaneous end user features */
        uint32_t ncd; /* number of country codes */
    } data_modem;
    uchar_t cd[16]; /* CCITT country code */
}
struct fax {
    uint32_t uf; /* highest data rate in DTE/UART */
    uint32_t fm; /* CCITT modulation standards */
    uint32_t fy; /* standardized data encryption */
    uint32_t fs; /* feature selection */
    uint32_t ncf; /* number of country codes */
    uchar_t cf[16]; /* CCITT country codes */
} fax;
struct voice {
    uint32_t uv; /* highest data rate */
    uint32_t nsr;
    uint32_t sr[16]; /* voice sampling rates (*100) */
    uint32_t nss;
    uint32_t ss[16]; /* voice sample sizes (*10) */
    uint32_t nsc;
    uint32_t sc[16]; /* voice compression methods */
} voice;
struct lan {
    uint32_t tech; /* network technology */
    uint32_t speed; /* media bit or baud rate */
    uint32_t media; /* network media supported */
    uint32_t con; /* open/closed connector standard */
    uint32_t id_sz; /* length of lan station id */
    uchar_t id[16]; /* station ID */
} lan;
}
data;

The fields are defined as follows:

function This field identifies the type of extended information provided about a function by the CISTPL_FUNCE tuple. This field is defined as follows:

TPLFE_SUB_SERIAL
    Serial port interface

TPLFE_SUB_MODEM_COMMON
    Common modem interface

TPLFE_SUB_MODEM_DATA
    Data modem services

TPLFE_SUB_MODEM_FAX
    Fax modem services

TPLFE_SUB_VOICE
    Voice services

TPLFE_CAP_MODEM_DATA
    Capabilities of the data modem interface

TPLFE_CAP_MODEM_FAX
    Capabilities of the fax modem interface

TPLFE_CAP_MODEM_VOICE
    Capabilities of the voice modem interface

178  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 20 Dec 1996
TPLFE_CAP_SERIAL_DATA
Serial port interface for data modem services

TPLFE_CAP_SERIAL_FAX
Serial port interface for fax modem services

TPLFE_CAP_SERIAL_VOICE
Serial port interface for voice modem services

subfunction
This is for identifying a sub-category of services
provided by a function in the CISTPL_FUNCE tuple.
The numeric value of the code is in the range of 1 to 15.

ua
This is the serial port UART identification and is
defined as follows:

TPLFE_UA_8250
Intel 8250

TPLFE_UA_16450
NS 16450

TPLFE_UA_16550
NS 16550

uc
This identifies the serial port UART capabilities and is
defined as follows:

TPLFE_UC_PARITY_SPACE
Space parity supported

TPLFE_UC_PARITY_MARK
Mark parity supported

TPLFE_UC_PARITY_ODD
Odd parity supported

TPLFE_UC_PARITY_EVEN
Even parity supported

TPLFE_UC_CS5
5 bit characters supported

TPLFE_UC_CS6
6 bit characters supported

TPLFE_UC_CS7
7 bit characters supported

TPLFE_UC_CS8
8 bit characters supported

TPLFE_UC_STOP_1
1 stop bit supported
fc
This identifies the modem flow control methods and is defined as follows:

TPLFE_FC_TX_XONOFF
Transmit XON/XOFF

TPLFE_FC_RX_XONOFF
Receiver XON/XOFF

TPLFE_FC_TX_HW
Transmit hardware flow control (CTS)

TPLFE_FC_RX_HW
Receiver hardware flow control (RTS)

TPLFE_FC_TRANS
Transparent flow control

ms
This identifies the modem modulation standards and is defined as follows:

TPLFE_MS_BELL103
300bps

TPLFE_MS_V21
300bps (V.21)

TPLFE_MS_V23
600/1200bps (V.23)

TPLFE_MS_V22AB
1200bps (V.22A V.22B)

TPLFE_MS_BELL212
2400bps (US Bell 212)

TPLFE_MS_V22BIS
2400bps (V.22bis)

TPLFE_MS_V26
2400bps leased line (V.26)

TPLFE_MS_V26BIS
2400bps (V.26bis)

TPLFE_MS_V27BIS
4800/2400bps leased line (V.27bis)
TPLFE_MS_V29
9600/7200/4800 leased line (V.29)

TPLFE_MS_V32
Up to 9600bps (V.32)

TPLFE_MS_V32BIS
Up to 14400bps (V.32bis)

TPLFE_MS_VFAST
Up to 28800 V.FAST

em
This identifies modem error correction/detection protocols and is defined as follows:
TPLFE_EM_MNP
MNP levels 2-4

TPLFE_EM_V42
CCITT LAPM (V.42)

dc
This identifies modem data compression protocols and is defined as follows:
TPLFE_DC_V42BI
CCITT compression V.42

TPLFE_DC_MNP5
MNP compression (uses MNP 2, 3 or 4)

cm
This identifies modem command protocols and is defined as follows:
TPLFE_CM_AT1
ANSI/EIA/TIA 602 "Action" commands

TPLFE_CM_AT2
ANSI/EIA/TIA 602 "ACE/DCE IF Params"

TPLFE_CM_AT3
ANSI/EIA/TIA 602 "Ace Parameters"

TPLFE_CM_MNP_AT
MNP specification AT commands

TPLFE_CM_V25BIS
V.25bis calling commands

TPLFE_CM_V25A
V.25bis test procedures

TPLFE_CM_DMCL
DMCL command mode

ex
This identifies the modem escape mechanism and is defined as follows:
csx_Parse_CISTPL_FUNCE(9F)

TPLFE_EX_BREAK
   BREAK support standardized

TPLFE_EX_PLUS
   +++ returns to command mode

TPLFE_EX_UD
   User defined escape character

dy
   This identifies modem standardized data encryption
   and is a reserved field for future use and must be set to
   0.

ef
   This identifies modem miscellaneous features and is
   defined as follows:

TPLFE_EF_CALLERID
   Caller ID is supported

fm
   This identifies fax modulation standards and is defined
   as follows:

TPLFE_FM_V21C2
   300bps (V.21-C2)

TPLFE_FM_V27TER
   4800/2400bps (V.27ter)

TPLFE_FM_V29
   9600/7200/4800 leased line (V.29)

TPLFE_FM_V17
   14.4K/12K/9600/7200bps (V.17)

TPLFE_FM_V33
   4.4K/12K/9600/7200 leased line (V.33)

fs
   This identifies the fax feature selection and is defined
   as follows:

TPLFE_FS_T3
   Group 2 (T.3) service class

TPLFE_FS_T4
   Group 3 (T.4) service class

TPLFE_FS_T6
   Group 4 (T.6) service class

TPLFE_FS_ECM
   Error Correction Mode

TPLFE_FS_VOICEREQ
   Voice requests allowed
Polling support

File transfer support

Password support

This identifies the LAN technology type and is defined as follows:

Arcnet

Ethernet

Token Ring

Local Talk

FDDI/CDDI

ATM

Wireless

Generic interface

Unshielded twisted pair

Shielded twisted pair

Thin coax

Thick coax

Fiber

Spread spectrum radio 902-928 MHz
csx_Parse_CISTPL_FUNC(9F)

| TPLFE_LAN_MEDIA_SSR_2_4 | Spread spectrum radio 2.4 GHz |
| TPLFE_LAN_MEDIA_SSR_5_4 | Spread spectrum radio 5.4 GHz |
| TPLFE_LAN_MEDIA_DIFFUSE_IR | Diffuse infra red |
| TPLFE_LAN_MEDIA_PTP_IR | Point to point infra red |

RETURN VALUES

CS_SUCCESS
Successful operation.

CS_BAD_HANDLE
Client handle is invalid.

CS_UNKNOWN_TUPLE
Parser does not know how to parse tuple.

CS_NO_CARD
No PC Card in socket.

CS_NO_CIS
No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
This function may be called from user or kernel context.

SEE ALSO
csx_GetFirstTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_FUNCID(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_FUNCID(client_handle_t ch, tuple_t *tu, cistpl_funcid_t *cf);

Solaris DDI Specific (Solaris DDI)

**ch** Client handle returned from csx_RegisterClient(9F).

**tu** Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

**cf** Pointer to a cistpl_funcid_t structure which contains the parsed CISTPL_FUNCID tuple information upon return from this function.

This function parses the Function Identification tuple, CISTPL_FUNCID, into a form usable by PC Card drivers.

The CISTPL_FUNCID tuple is used to describe information about the functionality provided by a PC Card. Information is also provided to enable system utilities to decide if the PC Card should be configured during system initialization. If additional function specific information is available, one or more function extension tuples of type CISTPL_FUNCE follow this tuple (see csx_Parse_CISTPL_FUNCE(9F)).

The structure members of cistpl_funcid_t are:

```
uint32_t function; /* PC Card function code */
uint32_t sysinit; /* system initialization mask */
```

The fields are defined as follows:

**function**

- **TPLFUNC_MULTI** Vendor-specific multifunction card
- **TPLFUNC_MEMORY** Memory card
- **TPLFUNC_SERIAL** Serial I/O port
- **TPLFUNC_PARALLEL** Parallel printer port
- **TPLFUNC_FIXED** Fixed disk, silicon or removable
- **TPLFUNC_VIDEO** Video interface
- **TPLFUNC_LAN** Local Area Network adapter
csx_Parse_CISTPL_FUNCID(9F)

TPLFUNC_AIMS
  Auto Incrementing Mass Storage
TPLFUNC_SCSI
  SCSI bridge
TPLFUNC_SECURITY
  Security cards
TPLFUNC_VENDOR_SPECIFIC
  Vendor specific
TPLFUNC_UNKNOWN
  Unknown function(s)

sysinit
  This field is bit-mapped and defined as follows:
  TPLINIT_POST
    POST should attempt configure
  TPLINIT_ROM
    Map ROM during sys init

RETURN VALUES

CS_SUCCESS
  Successful operation.
CS_BAD_HANDLE
  Client handle is invalid.
CS_UNKNOWN_TUPLE
  Parser does not know how to parse tuple.
CS_NO_CARD
  No PC Card in socket.
CS_NO_CIS
  No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION
  No PCMCIA hardware installed.

CONTEXT
  This function may be called from user or kernel context.

SEE ALSO
  csx_GetFirstTuple(9F), csx_GetTupleData(9F),
  csx_Parse_CISTPL_FUNCID(9F), csx_RegisterClient(9F),
  csxValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_GEOMETRY(9F)

NAME

csx_Parse_CISTPL_GEOMETRY – parse the Geometry tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_GEOMETRY(client_handle_t ch, tuple_t *tu,
cistpl_geometry_t *pt);

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch    Client handle returned from csx_RegisterClient(9F).

tu    Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

pt    Pointer to a cistpl_geometry_t structure which contains the parsed
CISTPL_GEOMETRY tuple information upon return from this function.

DESCRIPTION

This function parses the Geometry tuple, CISTPL_GEOMETRY, into a form usable by
PC Card drivers.

The CISTPL_GEOMETRY tuple indicates the geometry of a disk-like device.

STRUCTURE MEMBERS

The structure members of cistpl_geometry_t are:

    uint32_t    spt;
    uint32_t    tpc;
    uint32_t    ncyl;

The fields are defined as follows:

    spt     This field indicates the number of sectors per track.
    tpc     This field indicates the number of tracks per cylinder.
    ncyl    This field indicates the number of cylinders.

RETURN VALUES

CS_SUCCESS             Successful operation.
CS_BAD_HANDLE          Client handle is invalid.
CS_UNKNOWN_TUPLE       Parser does not know how to parse tuple.
CS_NO_CARD             No PC Card in socket.
CS_NO_CIS              No Card Information Structure (CIS) on PC
                        Card.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_JEDEC_A(9F)

NAME  csx_Parse_CISTPL_JEDEC_C, csx_Parse_CISTPL_JEDEC_A – parse JEDEC Identifier tuples

SYNOPSIS  

```c
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_JEDEC_C(client_handle_t ch, tuple_t *tu,
                                 cistpl_jedec_t *cj);

int32_t csx_Parse_CISTPL_JEDEC_A(client_handle_t ch, tuple_t *tu,
                                 cistpl_jedec_t *cj);
```

INTERFACE  Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Client handle returned from csx_RegisterClient(9F).</td>
</tr>
<tr>
<td>tu</td>
<td>Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</td>
</tr>
<tr>
<td>cj</td>
<td>Pointer to a cistpl_jedec_t structure which contains the parsed CISTPL_JEDEC_C or CISTPL_JEDEC_A tuple information upon return from these functions, respectively.</td>
</tr>
</tbody>
</table>

DESCRIPTION  csx_Parse_CISTPL_JEDEC_C() and csx_Parse_CISTPL_JEDEC_A() parse the JEDEC Identifier tuples, CISTPL_JEDEC_C and CISTPL_JEDEC_A, respectively, into a form usable by PC Card drivers.

The CISTPL_JEDEC_C and CISTPL_JEDEC_A tuples are optional tuples provided for cards containing programmable devices. They describe information for Common Memory or Attribute Memory space, respectively.

STRUCTURE  The structure members of cistpl_jedec_t are:

```c
uint32_t nid; /* # of JEDEC identifiers present */
jedec_ident_t jid[CISTPL_JEDEC_MAX_IDENTIFIERS];
```

The structure members of jedec_ident_t are:

```c
uint32_t id; /* manufacturer id */
uint32_t info; /* manufacturer specific info */
```

RETURN VALUES  

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

CONTEXT  These functions may be called from user or kernel context.
SEE ALSO

csx_GetFirstTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_DEVICE(9F), csx_RegisterClient(9F),
csx.ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_JEDEC_C(9F)

NAME

csx_Parse_CISTPL_JEDEC_C, csx_Parse_CISTPL_JEDEC_A – parse JEDEC Identifier tuples

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_JEDEC_C(client_handle_t ch, tuple_t *tu,
        cistpl_jedec_t *cj);

int32_t csx_Parse_CISTPL_JEDEC_A(client_handle_t ch, tuple_t *tu,
        cistpl_jedec_t *cj);

INTERFACE LEVEL

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
        csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cj Pointer to a cistpl_jedec_t structure which contains the parsed
        CISTPL_JEDEC_C or CISTPL_JEDEC_A tuple information upon return
        from these functions, respectively.

DESCRIPTION

csx_Parse_CISTPL_JEDEC_C() and csx_Parse_CISTPL_JEDEC_A() parse the
JEDEC Identifier tuples, CISTPL_JEDEC_C and CISTPL_JEDEC_A, respectively, into
a form usable by PC Card drivers.

The CISTPL_JEDEC_C and CISTPL_JEDEC_A tuples are optional tuples provided for
cards containing programmable devices. They describe information for Common
Memory or Attribute Memory space, respectively.

STRUCTURE MEMBERS

The structure members of cistpl_jedec_t are:

    uint32_t nid; /* # of JEDEC identifiers present */
    jedec_ident_t jid[CISTPL_JEDEC_MAX_IDENTIFIERS];

The structure members of jedec_ident_t are:

    uint32_t id; /* manufacturer id */
    uint32_t info; /* manufacturer specific info */

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD No PC Card in socket.

CS_NO_CIS No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

These functions may be called from user or kernel context.
SEE ALSO

- `csx_GetFirstTuple(9F), csx_GetTupleData(9F),`
- `csx_Parse_CISTPL_DEVICE(9F), csx_RegisterClient(9F),`
- `csx.ValidateCIS(9F), tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_LINKTARGET(9F)

NAME  csx_Parse_CISTPL_LINKTARGET – parse the Link Target tuple

SYNOPSIS  

```c
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_LINKTARGET(client_handle_t ch, tuple_t *tu, cistpl_linktarget_t *pt);
```

 INTERFACE

Solaris DDI Specific (Solaris DDI)

 PARAMETERS

- `ch`: Client handle returned from csx_RegisterClient(9F).
- `tu`: Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- `pt`: Pointer to a `cistpl_linktarget_t` structure which contains the parsed CISTPL_LINKTARGET tuple information upon return from this function.

 DESCRIPTION

This function parses the Link Target tuple, CISTPL_LINKTARGET, into a form usable by PCCard drivers.

The CISTPL_LINKTARGET tuple is used to verify that tuple chains other than the primary chain are valid. All secondary tuple chains are required to contain this tuple as the first tuple of the chain.

 STRUCTURE MEMBERS

The structure members of `cistpl_linktarget_t` are:

- `length`: `uint32_t`
- `tpltg_tag`: `char[CIS_MAX_TUPLE_DATA_LEN]`

The fields are defined as follows:

- `length`: This field indicates the number of bytes in `tpltg_tag`.
- `tpltg_tag`: This field provides the Link Target tuple information.

 RETURN VALUES

- `CS_SUCCESS`: Successful operation.
- `CS_BAD_HANDLE`: Client handle is invalid.
- `CS_UNKNOWN_TUPLE`: Parser does not know how to parse tuple.
- `CS_NO_CARD`: No PC Card in socket.
- `CS_NO_CIS`: No Card Information Structure (CIS) on PC Card.
- `CS_UNSUPPORTED_FUNCTION`: No PCMCIA hardware installed.

 CONTEXT

This function may be called from user or kernel context.

 SEE ALSO

- `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_RegisterClient(9F)`, `csx_ValidateCIS(9F)`, `tuple(9S)`

- `PC Card 95 Standard`, `PCMCIA/JEIDA`
## NAME

`csx_Parse_CISTPL_LONGLINK_A`, `csx_Parse_CISTPL_LONGLINK_C` – parse the Long Link A and C tuples

## SYNOPSIS

```c
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_LONGLINK_A(client_handle_t ch, tuple_t *tu, cistpl_longlink_ac_t *pt);

int32_t csx_Parse_CISTPL_LONGLINK_C(client_handle_t ch, tuple_t *tu, cistpl_longlink_ac_t *pt);
```

## INTERFACE LEVEL PARAMETERS

- **ch**  
  Client handle returned from `csx_RegisterClient(9F)`.
- **tu**  
  Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- **pt**  
  Pointer to a `cistpl_longlink_ac_t` structure which contains the parsed `CISTPL_LONGLINK_A` or `CISTPL_LONGLINK_C` tuple information upon return from this function.

## DESCRIPTION

This function parses the Long Link A and C tuples, `CISTPL_LONGLINK_A` and `CISTPL_LONGLINK_C`, into a form usable by PC Card drivers.

The `CISTPL_LONGLINK_A` and `CISTPL_LONGLINK_C` tuples provide links to Attribute and Common Memory.

## STRUCTURE MEMBERS

The structure members of `cistpl_longlink_ac_t` are:

- `uint32_t flags;`
- `uint32_t tpll_addr;`

The fields are defined as follows:

- **flags**  
  This field indicates the type of memory:
  - `CISTPL_LONGLINK_AC_AM`  
    long link to Attribute Memory
  - `CISTPL_LONGLINK_AC_CM`  
    long link to Common Memory

- **tpll_addr**  
  This field provides the offset from the beginning of the specified address space.

## RETURN VALUES

- **CS_SUCCESS**  
  Successful operation.
- **CS_BAD_HANDLE**  
  Client handle is invalid.
- **CS_UNKNOWN_TUPLE**  
  Parser does not know how to parse tuple.
- **CS_NO_CARD**  
  No PC Card in socket.
- **CS_NO_CIS**  
  No Card Information Structure (CIS) on PC Card.
csx_Parse_CISTPL_LONGLINK_A(9F)

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

---

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.
csx_Parse_CISTPL_LONGLINK_A, csx_Parse_CISTPL_LONGLINK_C – parse the Long Link A and C tuples

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_LONGLINK_A(client_handle_t ch, tuple_t *tu, cistpl_longlink_ac_t *pt);

int32_t csx_Parse_CISTPL_LONGLINK_C(client_handle_t ch, tuple_t *tu, cistpl_longlink_ac_t *pt);

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

pt Pointer to a cistpl_longlink_ac_t structure which contains the parsed CISTPL_LONGLINK_A or CISTPL_LONGLINK_C tuple information upon return from this function.

This function parses the Long Link A and C tuples, CISTPL_LONGLINK_A and CISTPL_LONGLINK_C, into a form usable by PC Card drivers.

The CISTPL_LONGLINK_A and CISTPL_LONGLINK_C tuples provide links to Attribute and Common Memory.

The structure members of cistpl_longlink_ac_t are:

uint32_t flags;
uint32_t tpll_addr;

The fields are defined as follows:

flags This field indicates the type of memory:

CISTPL_LONGLINK_AC_AM long link to Attribute Memory
CISTPL_LONGLINK_AC_CM long link to Common Memory

tpll_addr This field provides the offset from the beginning of the specified address space.

CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.
CS_NO_CARD No PC Card in socket.
CS_NO_CIS No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>This function may be called from user or kernel context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEE ALSO</td>
<td>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</td>
</tr>
</tbody>
</table>

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_LONGLINK_MFC(9F)

### NAME

csx_Parse_CISTPL_LONGLINK_MFC – parse the Multi-Function tuple

### SYNOPSIS

```c
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_LONGLINK_MFC(client_handle_t ch, tuple_t *tu, cistpl_longlink_mfc_t *pt);
```

### INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

### PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ch</code></td>
<td>Client handle returned from csx_RegisterClient(9F).</td>
</tr>
<tr>
<td><code>tu</code></td>
<td>Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</td>
</tr>
<tr>
<td><code>pt</code></td>
<td>Pointer to a cistpl_longlink_mfc_t structure which contains the parsed CISTPL_LONGLINK_MFC tuple information upon return from this function.</td>
</tr>
</tbody>
</table>

### DESCRIPTION

This function parses the Multi-Function tuple, CISTPL_LONGLINK_MFC, into a form usable by PC Card drivers.

The CISTPL_LONGLINK_MFC tuple describes the start of the function-specific CIS for each function on a multi-function card.

### STRUCTURE MEMBERS

The structure members of cistpl_longlink_mfc_t are:

```c
uint32_t nfuncs;
uint32_t nregs;
uint32_t function[CIS_MAX_FUNCTIONS].tas
uint32_t function[CIS_MAX_FUNCTIONS].addr
```

The fields are defined as follows:

- **nfuncs**
  - This field indicates the number of functions on the PC card.

- **nregs**
  - This field indicates the number of configuration register sets.

- **function[CIS_MAX_FUNCTIONS].tas**
  - This field provides the target address space for each function on the PC card. This field can be one of:
    - CISTPL_LONGLINK_MFC_TAS_AM
      - CIS in attribute memory
    - CISTPL_LONGLINK_MFC_TAS_CM
      - CIS in common memory

- **function[CIS_MAX_FUNCTIONS].addr**
  - This field provides the target address offset for each function on the PC card.

### RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
</tbody>
</table>
csx_Parse_CISTPL_LONGLINK_MFC(9F)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
`csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_MANFID(9F)

### NAME
`csx_Parse_CISTPL_MANFID` – parse Manufacturer Identification tuple

### SYNOPSIS
```
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_MANFID(client_handle_t ch, tuple_t *tu,
    cistpl_manfid_t *cm);
```

### INTERFACE LEVEL
**Solaris DDI Specific (Solaris DDI)**

#### PARAMETERS
- `ch`:
  Client handle returned from `csx_RegisterClient(9F)`.
- `tu`:
  Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- `cm`:
  Pointer to a `cistpl_manfid_t` structure which contains the parsed CISTPL_MANFID tuple information upon return from this function.

### DESCRIPTION
This function parses the Manufacturer Identification tuple, CISTPL_MANFID, into a form usable by PC Card drivers.

The CISTPL_MANFID tuple is used to describe the information about the manufacturer of a PC Card. There are two types of information, the PC Card’s manufacturer and a manufacturer card number.

### STRUCTURE
The structure members of `cistpl_manfid_t` are:

```c
uint32_t manf; /* PCMCIA assigned manufacturer code */
uint32_t card; /* manufacturer information
    (part number and/or revision) */
```

### RETURN VALUES
- `CS_SUCCESS`:
  Successful operation.
- `CS_BAD_HANDLE`:
  Client handle is invalid.
- `CS_UNKNOWN_TUPLE`:
  Parser does not know how to parse tuple.
- `CS_NO_CARD`:
  No PC Card in socket.
- `CS_NO_CIS`:
  No Card Information Structure (CIS) on PC card.
- `CS_UNSUPPORTED_FUNCTION`:
  No PCMCIA hardware installed.

### CONTEXT
This function may be called from user or kernel context.

### SEE ALSO
- `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_RegisterClient(9F)`
- `csx_ValidateCIS(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*
### NAME

csx_Parse_CISTPL_ORG(9F)

### SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_ORG(client_handle_t ch, tuple_t *tu, cistpl_org_t *pt);
```

### INTERFACE LEVEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ch</code></td>
<td>Client handle returned from csx_RegisterClient(9F).</td>
</tr>
<tr>
<td><code>tu</code></td>
<td>Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</td>
</tr>
<tr>
<td><code>pt</code></td>
<td>Pointer to a cistpl_org_t structure which contains the parsed CISTPL_ORG tuple information upon return from this function.</td>
</tr>
</tbody>
</table>

### DESCRIPTION

This function parses the Data Organization tuple, CISTPL_ORG, into a form usable by PC Card drivers.

The CISTPL_ORG tuple provides a text description of the organization.

### STRUCTURE MEMBERS

The structure members of cistpl_org_t are:

```
uint32_t type;
char desc[CIS_MAX_TUPLE_DATA_LEN];
```

The fields are defined as follows:

- **type**
  - This field indicates type of data organization.

- **desc[CIS_MAX_TUPLE_DATA_LEN]**
  - This field provides the text description of this organization.

### RETURN VALUES

- **CS_SUCCESS**
  - Successful operation.

- **CS_BAD_HANDLE**
  - Client handle is invalid.

- **CS_UNKNOWN_TUPLE**
  - Parser does not know how to parse tuple.

- **CS_NO_CARD**
  - No PC Card in socket.

- **CS_NO_CIS**
  - No Card Information Structure (CIS) on PC Card.

- **CS_UNSUPPORTED_FUNCTION**
  - No PCMCIA hardware installed.

### CONTEXT

This function may be called from user or kernel context.

### SEE ALSO

- csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)

- *PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_SPCL(9F)

NAME

csx_Parse_CISTPL_SPCL – parse the Special Purpose tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_SPCL(client_handle_t ch, tuple_t *tu,
                                cistpl_spcl_t *csp);

INTERFACE LEVEL

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

csp Pointer to a cistpl_spcl_t structure which contains the parsed
      CISTPL_SPCL tuple information upon return from this function.

DESCRIPTION

This function parses the Special Purpose tuple, CISTPL_SPCL, into a form usable by
PC Card drivers.

The CISTPL_SPCL tuple is identified by an identification field that is assigned by
PCMCIA or JEIDA. A sequence field allows a series of CISTPL_SPCL tuples to be
used when the data exceeds the size that can be stored in a single tuple; the maximum
data area of a series of CISTPL_SPCL tuples is unlimited. Another field gives the
number of bytes in the data field in this tuple.

STRUCTURE MEMBERS

The structure members of cistpl_date_t are:

uint32_t id;    /* tuple contents identification */
uint32_t seq;   /* data sequence number */
uint32_t bytes; /* number of bytes following */
uchar_t data[CIS_MAX_TUPLE_DATA_LEN];

The fields are defined as follows:

id This field contains a PCMCIA or JEIDA assigned value that identifies this
      series of one or more CISTPL_SPCL tuples. These field values are assigned
      by contacting either PCMCIA or JEIDA.

seq This field contains a data sequence number. CISTPL_SPCL_SEQ_END is
      the last tuple in sequence.

bytes This field contains the number of data bytes in the
      data[CIS_MAX_TUPLE_DATA_LEN].

data The data component of this tuple.

RETURN VALUES

CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.
CS_NO_CARD No PC Card in socket.
CS_NO_CIS No Card Information Structure (CIS) on PC Card.
csx_Parse_CISTPL_SPCL(9F)

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>CS_UNSUPPORTED_FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

This function may be called from user or kernel context.

SEE ALSO

- csx_GetFirstTuple(9F)
- csx_GetTupleData(9F)
- csx_RegisterClient(9F)
- csx_ValidateCIS(9F)
- tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Parse_CISTPL_SWIL(9F)

NAME

csx_Parse_CISTPL_SWIL — parse the Software Interleaving tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_SWIL(client_handle_t ch, tuple_t *tu,
cistpl_swil_t *pt);

INTERFACE LEVEL PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

pt Pointer to a cistpl_swil_t structure which contains the parsed
CISTPL_SWIL tuple information upon return from this function.

DESCRIPTION

This function parses the Software Interleaving tuple, CISTPL_SWIL, into a form
usable by PC Card drivers.

The CISTPL_SWIL tuple provides the software interleaving of data within a partition
on the card.

STRUCTURE MEMBERS

The structure members of cistpl_swil_t are:

uint32_t intrlv;

The fields are defined as follows:

intrlv This field provides the software interleaving for a partition.

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD No PC Card in socket.

CS_NO_CIS No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA
csx_Parse_CISTPL_VERS_1(9F)

NAME csx_Parse_CISTPL_VERS_1 – parse Level-1 Version/Product Information tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_VERS_1(client_handle_t ch, tuple_t *tu,
                                 cistpl_vers_1_t *cv1);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

tu Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
       csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cv1 Pointer to a cistpl_vers_1_t structure which contains the parsed
       CISTPL_VERS_1 tuple information upon return from this function.

DESCRIPTION

This function parses the Level-1 Version/Product Information tuple,
CISTPL_VERS_1, into a form usable by PC Card drivers.

The CISTPL_VERS_1 tuple is used to describe the card Level-1 version compliance
and card manufacturer information.

STRUCTURE

The structure members of cistpl_vers_1_t are:

uint32_t major; /* major version number */
uint32_t minor; /* minor version number */
uint32_t ns; /* number of information strings */
char pi[CISTPL_VERS_1_MAX_PROD_STRINGS]
       [CIS_MAX_TUPLE_DATA_LEN];
       /* pointers to product information strings */

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD No PC Card in socket.

CS_NO_CIS No Card Information Structure (CIS) on PC Card.

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
         csx_ValidateCIS(9F), tuple(9S)

         PC Card 95 Standard, PCMCIA/JEIDA

204 man pages section 9: DDI and DKI Kernel Functions • Last Revised 20 Dec 1996
#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_VERS_2(client_handle_t ch, tuple_t *tu, cistpl_vers_2_t *cv2);

**SYNOPSIS**

Solaris DDI Specific (Solaris DDI)

- *ch* Client handle returned from `csx_RegisterClient(9F)`.
- *tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- *cv2* Pointer to a `cistpl_vers_2_t` structure which contains the parsed `CISTPL_VERS_2` tuple information upon return from this function.

**DESCRIPTION**

This function parses the Level-2 Version and Information tuple, `CISTPL_VERS_2`, into a form usable by PC Card drivers.

The `CISTPL_VERS_2` tuple is used to describe the card Level-2 information which has the logical organization of the card's data.

**STRUCTURE MEMBERS**

The structure members of `cistpl_vers_2_t` are:

- `uint32_t vers; /* version number */`
- `uint32_t comply; /* level of compliance */`
- `uint32_t dindex; /* byte address of first data byte in card */`
- `uint32_t vspec8; /* vendor specific (byte 8) */`
- `uint32_t vspec9; /* vendor specific (byte 9) */`
- `uint32_t nhdr; /* number of copies of CIS present on device */`
- `char oem[CIS_MAX_TUPLE_DATA_LEN]; /* Vendor of software that formatted card */`
- `char info[CIS_MAX_TUPLE_DATA_LEN]; /* Informational message about card */`

**RETURN VALUES**

- `CS_SUCCESS` Successful operation.
- `CS_BAD_HANDLE` Client handle is invalid.
- `CS_UNKNOWN_TUPLE` Parser does not know how to parse tuple.
- `CS_NO_CARD` No PC Card in socket.
- `CS_NO_CIS` No Card Information Structure (CIS) on PC Card.
- `CS_UNSUPPORTED_FUNCTION` No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

`csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_ParseTuple(9F)

NAME    csx_ParseTuple – generic tuple parser

SYNOPSIS #include <sys/pccard.h>

int32_t csx_ParseTuple(client_handle_t ch, tuple_t *tu, cisparse_t *cp, cisdata_t cd);

INTERFACE Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch    Client handle returned from csx_RegisterClient(9F).

tu    Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).

cp    Pointer to a cisparse_t structure that unifies all tuple parsing structures.

cd    Extended tuple data for some tuples.

DESCRIPTION This function is the generic tuple parser entry point.

STRUCTURE The structure members of cisparse_t are:

typedef union cisparse_t {
    cistpl_config_t     cistpl_config;
    cistpl_device_t     cistpl_device;
    cistpl_vers_1_t     cistpl_vers_1;
    cistpl_vers_2_t     cistpl_vers_2;
    cistpl_jedec_t      cistpl_jedec;
    cistpl_format_t     cistpl_format;
    cistpl_geometry_t   cistpl_geometry;
    cistpl_byteorder_t  cistpl_byteorder;
    cistpl_date_t       cistpl_date;
    cistpl_battery_t    cistpl_battery;
    cistpl_org_t        cistpl_org;
    cistpl_manfid_t     cistpl_manfid;
    cistpl_funcid_t     cistpl_funcid;
    cistpl_funce_t      cistpl_funce;
    cistpl_cftable_entry_t cistpl_cftable_entry;
    cistpl_linktarget_t cistpl_linktarget;
    cistpl_longlink_ac_t cistpl_longlink_ac;
    cistpl_longlink_mfc_t cistpl_longlink_mfc;
    cistpl_spcl_t       cistpl_spcl;
    cistpl_swil_t       cistpl_swil;
    cistpl_bar_t        cistpl_bar;
    cistpl_devicegeo_t  cistpl_devicegeo;
    cistpl_longlink_cb_t cistpl_longlink_cb;
    cistpl_get_tuple_name_t cistpl_get_tuple_name;
} cisparse_t;

RETURN VALUES

CS_SUCCESS   Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD   No PC Card in socket.

CS_BAD_CIS   Generic parser error.
csx_ParseTuple(9F)

<table>
<thead>
<tr>
<th>CS_NO_CIS</th>
<th>No Card Information Structure (CIS) on PC Card.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_GetFirstTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPL_BATTERY(9F), csx_Parse_CISTPL_BYTEORDER(9F),
csx_Parse_CISTPL_CFTABLE_ENTRY(9F), csx_Parse_CISTPL_CONFIG(9F),
csx_Parse_CISTPL_DATE(9F), csx_Parse_CISTPL_DEVICE(9F),
csx_Parse_CISTPL_FUNCE(9F), csx_Parse_CISTPL_FUNCID(9F),
csx_Parse_CISTPL_JEDEC_C(9F), csx_Parse_CISTPL_MANFID(9F),
csx_Parse_CISTPL_SPCL(9F), csx_Parse_CISTPL_VERS_1(9F),
csx_Parse_CISTPL_VERS_2(9F), csx_RegisterClient(9F),
csx_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_Put16(9F)

NAME

csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register

SYNOPSIS

#include <sys/pccard.h>

void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

handle The access handle returned from csx_RequestIO(9F),
          csx_RequestWindow(9F), or csx_DupHandle(9F).

offset The offset in bytes from the base of the mapped resource.

value The data to be written to the device.

DESCRIPTION

These functions generate a write of various sizes to the mapped memory or device
register.

The csx_Put8(), csx_Put16(), csx_Put32(), and csx_Put64() functions write
8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address
represented by the handle, handle, at an offset in bytes represented by the offset, offset.

Data that consists of more than one byte will automatically be translated to maintain a
consistent view between the host and the device based on the encoded information in
the data access handle. The translation may involve byte swapping if the host and the
device have incompatible endian characteristics.

CONTEXT

These functions may be called from user, kernel, or interrupt context.

SEE ALSO

csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_RepGet8(9F),
csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA

208 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register

#include <sys/pccard.h>

void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);

**INTERFACE LEVEL**

**PARAMETERS**

- **handle**: The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
- **offset**: The offset in bytes from the base of the mapped resource.
- **value**: The data to be written to the device.

**DESCRIPTION**

These functions generate a write of various sizes to the mapped memory or device register.

The csx_Put8(), csx_Put16(), csx_Put32(), and csx_Put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions may be called from user, kernel, or interrupt context.

**SEE ALSO**

csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_RepGet8(9F), csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
NAME  csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register

SYNOPSIS  

```c
#include <sys/pccard.h>

void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

PARAMETERS  

- `handle`  The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- `offset`  The offset in bytes from the base of the mapped resource.
- `value`  The data to be written to the device.

DESCRIPTION  These functions generate a write of various sizes to the mapped memory or device register.

The `csx_Put8()`, `csx_Put16()`, `csx_Put32()`, and `csx_Put64()` functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

CONTEXT  These functions may be called from user, kernel, or interrupt context.

SEE ALSO  `csx_DupHandle(9F)`, `csx_Get8(9F)`, `csx_GetMappedAddr(9F)`, `csx_RepGet8(9F)`, `csx_RepPut8(9F)`, `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`

PC Card 95 Standard, PCMCIA/JEIDA
csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register

#include <sys/pccard.h>

void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);

handle

The access handle returned from csx_RequestIO(9F),
csx_RequestWindow(9F), or csx_DupHandle(9F).

offset

The offset in bytes from the base of the mapped resource.

value

The data to be written to the device.

These functions generate a write of various sizes to the mapped memory or device
register.

The csx_Put8(), csx_Put16(), csx_Put32(), and csx_Put64() functions write
8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address
represented by the handle, handle, at an offset in bytes represented by the offset, offset.

Data that consists of more than one byte will automatically be translated to maintain a
consistent view between the host and the device based on the encoded information in
the data access handle. The translation may involve byte swapping if the host and the
device have incompatible endian characteristics.

These functions may be called from user, kernel, or interrupt context.

SEE ALSO

csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_RepGet8(9F),
csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_RegisterClient(9F)

NAME  csx_RegisterClient – register a client

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_RegisterClient(client_handle_t *ch, client_reg_t *cr);

INTERFACE
Solaris DDI Specific (Solaris DDI)

LEVEL
PARAMETERS
ch Pointer to a client_handle_t structure.
mc Pointer to a client_reg_t structure.

DESCRIPTION
This function registers a client with Card Services and returns a unique client handle
for the client. The client handle must be passed to csx_DeregisterClient(9F)
when the client terminates.

STRUCTURE
The structure members of client_reg_t are:

uint32_t Attributes;
uint32_t EventMask;
event_callback_args_t event_callback_args;
uint32_t Version; /* CS version to expect */
csfuction_t *event_handler;
ddi_iblock_cookie_t *iblk_cookie; /* event iblk cookie */
ddi_idevice_cookie_t *idev_cookie; /* event idev cookie */
dev_info_t *dip; /* client’s dip */
char driver_name[MODMAXNAMELEN];

The fields are defined as follows:

Attributes
This field is bit-mapped and defined as follows:

INFO_MEM_CLIENT
Memory client device driver.

INFO_MTD_CLIENT
Memory Technology Driver client.

INFO_IO_CLIENT
IO client device driver.

INFO_CARD_SHARE
Generate artificial CS_EVENT_CARD_INSERTION and
CS_EVENT_REGISTRATION_COMPLETE events.

INFO_CARD_EXCL
Generate artificial CS_EVENT_CARD_INSERTION and
CS_EVENT_REGISTRATION_COMPLETE events.

INFO_MEM_CLIENT
INFO_MTD_CLIENT
INFO_IO_CLIENT
These bits are mutually exclusive (that is, only one bit may be set), but one of
the bits must be set.
If either of these bits is set, the client will receive a
CS_EVENT_REGISTRATION_COMPLETE event when Card Services has
completed its internal client registration processing and after a successful call
to csx_RequestSocketMask(9F).

Also, if either of these bits is set, and if a card of the type that the client can
control is currently inserted in the socket (and after a successful call to
csx_RequestSocketMask(9F)), the client will receive an artificial
CS_EVENT_CARD_INSERTION event.

Event Mask
This field is bit-mapped and specifies the client’s global event mask. Card Services
performs event notification based on this field. See csx_event_handler(9E) for
valid event definitions and for additional information about handling events.

event_callback_args
The event_callback_args structure members are:

```c
void* client_data;
```

The client_data field may be used to provide data available to the event handler
(see csx_event_handler(9E)). Typically, this is the client driver’s soft state
pointer.

Version
This field contains the specific Card Services version number that the client expects
to use. Typically, the client will use the CS_VERSION macro to specify to Card
Services which version of Card Services the client expects.

event_handler
The client event callback handler entry point is passed in the event_handler
field.

iblk_cookie
idev_cookie
These fields must be used by the client to set up mutexes that are used in the
client’s event callback handler when handling high priority events.

dip
The client must set this field with a pointer to the client’s dip.

driver_name
The client must copy a driver-unique name into this member. This name must be
identical across all instances of the driver.

**RETURN VALUES**

CS_SUCCESS
Successful operation.

CS_BAD_ATTRIBUTE
No client type or more than one client type specified.
csx_RegisterClient(9F)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_OUT_OF_RESOURCE</td>
<td>Card Services is unable to register client.</td>
</tr>
<tr>
<td>CS_BAD_VERSION</td>
<td>Card Services version is incompatible with client.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client has already registered for this socket.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**
This function may be called from user or kernel context.

**SEE ALSO**
csx_DeregisterClient(9F), csx_RequestSocketMask(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_ReleaseConfiguration(9F)

NAME
csx_ReleaseConfiguration – release PC Card and socket configuration

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_ReleaseConfiguration(client_handle_t ch,
                                release_config_t *rc);

INTERFACE
Solaris DDI Specific (Solaris DDI)

LEVEL
PARAMETERS
ch Client handle returned from csx_RegisterClient(9F).
rc Pointer to a release_config_t structure.

DESCRIPTION
This function returns a PC Card and socket to a simple memory only interface and sets
the card to configuration zero by writing a 0 to the PC card’s COR (Configuration
Option Register).

Card Services may remove power from the socket if no clients have indicated their
usage of the socket by an active csx_RequestConfiguration(9F) or
csx_RequestWindow(9F).

Card Services is prohibited from resetting the PC Card and is not required to cycle
power through zero (0) volts.

After calling csx_ReleaseConfiguration() any resources requested via the
request functions csx_RequestIO(9F), csx_RequestIRQ(9F), or
csx_RequestWindow(9F) that are no longer needed should be returned to Card
Services via the corresponding csx_ReleaseIO(9F), csx_ReleaseIRQ(9F), or
csx_ReleaseWindow(9F) functions. csx_ReleaseConfiguration() must be
called to release the current card and socket configuration before releasing any
resources requested by the driver via the request functions named above.

STRUCTURE
MEMBERS
The structure members of release_config_t are:

uint32_t Socket; /* socket number */

The Socket field is not used in Solaris, but for portability with other Card Services
implementations, it should be set to the logical socket number.

RETURN VALUES
CS_SUCCESS
Successful operation.

CS_BAD_HANDLE
Client handle is invalid or csx_RequestConfiguration(9F) not done.

CS_BAD_SOCKET
Error getting or setting socket hardware parameters.

CS_NO_CARD
No PC card in socket.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.
csx_ReleaseConfiguration(9F)

CONTEXT This function may be called from user or kernel context.

SEE ALSO csx_RegisterClient(9F), csx_RequestConfiguration(9F),
csx_RequestIO(9F), csx_RequestIRQ(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/ JEIDA
csx_ReleaseIO(9F)

NAME

csx_RequestIO, csx_ReleaseIO – request or release I/O resources for the client

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir);
int32_t csx_ReleaseIO(client_handle_t ch, io_req_t *ir);

INTERFACE LEVEL PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch  Client handle returned from csx_RegisterClient(9F).

ir  Pointer to an io_req_t structure.

DESCRIPTION

The functions csx_RequestIO() and csx_ReleaseIO() request or release, respectively, I/O resources for the client.

If a client requires I/O resources, csx_RequestIO() must be called to request I/O resources from Card Services; then csx_RequestConfiguration(9F) must be used to establish the configuration. csx_RequestIO() can be called multiple times until a successful set of I/O resources is found. csx_RequestConfiguration(9F) only uses the last configuration specified.

csx_RequestIO() fails if it has already been called without a corresponding csx_ReleaseIO().

csx_ReleaseIO() releases previously requested I/O resources. The Card Services window resource list is adjusted by this function. Depending on the adapter hardware, the I/O window might also be disabled.

STRUCTURE MEMBERS

The structure members of io_req_t are:

uint32_t Socket; /* socket number*/
uint32_t Baseport1.base; /* I/O range base port address */
acc_handle_t Baseport1.handle; /* I/O range base address */
uint32_t Baseport1.handle; /* or port num */
uint32_t NumPorts1; /* first I/O range number contiguous ports */
uint32_t Attributes1; /* first I/O range attributes */
uint32_t Baseport2.base; /* I/O range base port address */
acc_handle_t Baseport2.handle; /* I/O range base address or port num */
uint32_t NumPorts2; /* second I/O range number contiguous ports */
uint32_t Attributes2; /* second I/O range attributes */
uint32_t IOAddrLines; /* number of I/O address lines decoded */

The fields are defined as follows:

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
Two I/O address ranges can be requested by `csx_RequestIO()`. Each I/O address range is specified by the BasePort, NumPorts, and Attributes fields. If only a single I/O range is being requested, the NumPorts2 field must be reset to 0.

When calling `csx_RequestIO()`, the BasePort.base field specifies the first port address requested. Upon successful return from `csx_RequestIO()`, the BasePort.handle field contains an access handle, corresponding to the first byte of the allocated I/O window, which the client must use when accessing the PC Card’s I/O space via the common access functions. A client must not make any assumptions as to the format of the returned BasePort.handle field value.

If the BasePort.base field is set to 0, Card Services returns an I/O resource based on the available I/O resources and the number of contiguous ports requested. When BasePort.base is 0, Card Services aligns the returned resource in the host system’s I/O address space on a boundary that is a multiple of the number of contiguous ports requested, rounded up to the nearest power of two. For example, if a client requests two I/O ports, the resource returned will be a multiple of two. If a client requests five contiguous I/O ports, the resource returned will be a multiple of eight.

If multiple ranges are being requested, at least one of the BasePort.base fields must be non-zero.

NumPorts
This field is the number of contiguous ports being requested.

Attributes
This field is bit-mapped. The following bits are defined:

- **IO_DATA_WIDTH_8**
  I/O resource uses 8-bit data path.

- **IO_DATA_WIDTH_16**
  I/O resource uses 16-bit data path.

- **WIN_ACC_NEVER_SWAP**
  Host endian byte ordering.

- **WIN_ACC_BIG_ENDIAN**
  Big endian byte ordering

- **WIN_ACC_LITTLE_ENDIAN**
  Little endian byte ordering.

- **WIN_ACC_STRICT_ORDER**
  Program ordering references.

- **WIN_ACC_UNORDERED_OK**
  May re-order references.
WIN_ACC_MERGING_OK
  Merge stores to consecutive locations.

WIN_ACC_LOADCACHING_OK
  May cache load operations.

WIN_ACC_STORECACHING_OK
  May cache store operations.

For some combinations of host system busses and adapter hardware, the width of
an I/O resource can not be set via RequestIO(); on those systems, the host bus
cycle access type determines the I/O resource data path width on a per-cycle basis.

WIN_ACC_BIG_ENDIAN and WIN_ACC_LITTLE_ENDIAN describe the endian
characteristics of the device as big endian or little endian, respectively. Even though
most of the devices will have the same endian characteristics as their busses, there
are examples of devices with an I/O processor that has opposite endian
characteristics of the busses. When WIN_ACC_BIG_ENDIAN or WIN_ACC_LITTLE_ENDIAN is set, byte swapping will automatically be performed by the system if the
host machine and the device data formats have opposite endian characteristics. The
implementation may take advantage of hardware platform byte swapping
capabilities.

When WIN_ACC_NEVER_SWAP is specified, byte swapping will not be invoked in
the data access functions. The ability to specify the order in which the CPU will
reference data is provided by the following Attributes bits. Only one of the
following bits may be specified:

WIN_ACC_STRICT_ORDER
  The data references must be issued by a CPU in program order. Strict ordering is
  the default behavior.

WIN_ACC_UNORDERED_OK
  The CPU may re-order the data references. This includes all kinds of re-ordering
  (that is, a load followed by a store may be replaced by a store followed by a
  load).

WIN_ACC_MERGING_OK
  The CPU may merge individual stores to consecutive locations. For example, the
  CPU may turn two consecutive byte stores into one halfword store. It may also
  batch individual loads. For example, the CPU may turn two consecutive byte
  loads into one halfword load. IO_MERGING_OK_ACC also implies re-ordering.

WIN_ACC_LOADCACHING_OK
  The CPU may cache the data it fetches and reuse it until another store occurs.
  The default behavior is to fetch new data on every load.
  WIN_ACC_LOADCACHING_OK also implies merging and re-ordering.

WIN_ACC_STORECACHING_OK
  The CPU may keep the data in the cache and push it to the device (perhaps with
  other data) at a later time. The default behavior is to push the data right away.
  WIN_ACC_STORECACHING_OK also implies load caching, merging, and
These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together. All other bits in the Attributes field must be set to 0.

IOAddrLines
This field is the number of I/O address lines decoded by the PC Card in the specified socket.

On some systems, multiple calls to csx_RequestIO() with different BasePort, NumPorts, and/or IOAddrLines values will have to be made to find an acceptable combination of parameters that can be used by Card Services to allocate I/O resources for the client. (See NOTES).

RETURN VALUES

CS_SUCCESS
Successful operation.

CS_BAD_ATTRIBUTE
Invalid Attributes specified.

CS_BAD_BASE
BasePort value is invalid.

CS_BAD_HANDLE
Client handle is invalid.

CS_CONFIGURATION_LOCKED
csx_RequestConfiguration(9F) has already been done.

CS_IN_USE
csx_RequestIO() has already been done without a corresponding csx_ReleaseIO().

CS_NO_CARD
No PC Card in socket.

CS_BAD_WINDOW
Unable to allocate I/O resources.

CS_OUT_OF_RESOURCE
Unable to allocate I/O resources.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
These functions may be called from user or kernel context.

SEE ALSO
csx_RegisterClient(9F), csx_RequestConfiguration(9F)

PC Card 95 Standard, PCMCIA/JEIDA
It is important for clients to try to use the minimum amount of I/O resources necessary. One way to do this is for the client to parse the CIS of the PC Card and call `csx_RequestIO()` first with any IOAddrLines values that are 0 or that specify a minimum number of address lines necessary to decode the I/O space on the PC Card. Also, if no convenient minimum number of address lines can be used to decode the I/O space on the PC Card, it is important to try to avoid system conflicts with well-known architectural hardware features.
NAME csx_RequestIRQ, csx_ReleaseIRQ – request or release IRQ resource

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_RequestIRQ(client_handle_t ch, irq_req_t *ir);
int32_t csx_ReleaseIRQ(client_handle_t ch, irq_req_t *ir);

INTERFACE LEVEL PARAMETERS

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

ir Pointer to an irq_req_t structure.

DESCRIPTION

The function csx_RequestIRQ() requests an IRQ resource and registers the client’s IRQ handler with Card Services.

If a client requires an IRQ, csx_RequestIRQ() must be called to request an IRQ resource as well as to register the client’s IRQ handler with Card Services. The client will not receive callbacks at the IRQ callback handler until csx_RequestConfiguration(9F) or csx_ModifyConfiguration(9F) has successfully returned when either of these functions are called with the CONF_ENABLE_IRQ_STEERING bit set.

The function csx_ReleaseIRQ() releases a previously requested IRQ resource.

The Card Services IRQ resource list is adjusted by csx_ReleaseIRQ(). Depending on the adapter hardware, the host bus IRQ connection might also be disabled. Client IRQ handlers always run above lock level and so should take care to perform only Solaris operations that are appropriate for an above-lock-level IRQ handler.

csx_RequestIRQ() fails if it has already been called without a corresponding csx_ReleaseIRQ().

STRUCTURE MEMBERS

The structure members of irq_req_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* IRQ attribute flags */
csfunction_t *irq_handler; /* IRQ handler */
void *irq_handler_arg; /* IRQ handler argument */
ddi_iblock_cookie_t *iblk_cookie; /* IRQ interrupt block */

The fields are defined as follows:

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes

This field is bit-mapped. It specifies details about the type of IRQ desired by the client. The following bits are defined:
IRQ_TYPE_EXCLUSIVE
IRQ is exclusive to this socket. This bit must be set. It indicates that the system
IRQ is dedicated to this PC Card.

irq_handler
The client IRQ callback handler entry point is passed in the irq_handler field.

irq_handler_arg
The client can use the irq_handler_arg field to pass client-specific data to the
client IRQ callback handler.

iblk_cookie
idev_cookie
These fields must be used by the client to set up mutexes that are used in the
client’s IRQ callback handler.

For a specific csx_ReleaseIRQ() call, the values in the irq_req_t structure must
be the same as those returned from the previous csx_RequestIRQ() call; otherwise,
CS_BAD_ARGS is returned and no changes are made to Card Services resources or the
socket and adapter hardware.

RETURN VALUES
CS_SUCCESS
Successful operation.

CS_BAD_ARGS
IRQ description does not match allocation.

CS_BAD_ATTRIBUTE
IRQ_TYPE_EXCLUSIVE not set, or an unsupported or reserved bit is set.

CS_BAD_HANDLE
Client handle is invalid or csx_RequestConfiguration(9F) not done.

CS_BAD_IRQ
Unable to allocate IRQ resources.

CS_IN_USE
csx_RequestIRQ() already done or a previous csx_RequestIRQ() has not
been done for a corresponding csx_ReleaseIRQ().

CS_CONFIGURATION_LOCKED
csx_RequestConfiguration(9F) already done or
csx_ReleaseConfiguration(9F) has not been done.

CS_NO_CARD
No PC Card in socket.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
These functions may be called from user or kernel context.

SEE ALSO
csx_ReleaseConfiguration(9F), csx_RequestConfiguration(9F)
csx_ReleaseIRQ(9F)

PC Card Card 95 Standard, PCMCIA/JEIDA
**csx_RequestSocketMask, csx_ReleaseSocketMask** – set or clear the client’s client event mask

```c
#include <sys/pccard.h>

int32_t csx_RequestSocketMask(client_handle_t ch, request_socket_mask_t *sm);
int32_t csx_ReleaseSocketMask(client_handle_t ch, release_socket_mask_t *rm);
```

**INTERFACE LEVEL**
- Solaris DDI Specific (Solaris DDI)

**PARAMETERS**
- `ch` Client handle returned from `csx_RegisterClient(9F)`.
- `sm` Pointer to a `request_socket_mask_t` structure.
- `rm` Pointer to a `release_socket_mask_t` structure.

**DESCRIPTION**
- The function `csx_RequestSocketMask()` sets the client’s client event mask and enables the client to start receiving events at its event callback handler. Once this function returns successfully, the client can start receiving events at its event callback handler. Any pending events generated from the call to `csx_RegisterClient(9F)` will be delivered to the client after this call as well. This allows the client to set up the event handler mutexes before the event handler gets called.
- `csx_RequestSocketMask()` must be used before calling `csx_GetEventMask(9F)` or `csx_SetEventMask(9F)` for the client event mask for this socket.
- `csx_ReleaseSocketMask()` clears the client’s client event mask.

**STRUCTURE MEMBERS**
- The structure members of `request_socket_mask_t` are:
  ```c
  uint32_t Socket; /* socket number */
  uint32_t EventMask; /* event mask to set or return */
  ```
- The structure members of `release_socket_mask_t` are:
  ```c
  uint32_t Socket; /* socket number */
  ```

The fields are defined as follows:
- **Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
- **EventMask** This field is bit-mapped. Card Services performs event notification based on this field. See `csx_event_handler(9E)` for valid event definitions and for additional information about handling events.

**RETURN VALUES**
- **CS_SUCCESS** Successful operation.
- **CS_BAD_HANDLE** Client handle is invalid.
- **CS_IN_USE** `csx_ReleaseSocketMask()` has not been done.
CS_BAD_SOCKET: csx_RequestSocketMask() has not been done.

CS_UNSUPPORTED_FUNCTION: No PCMCIA hardware installed.

CONTEXT: These functions may be called from user or kernel context.

SEE ALSO: csx_event_handler(9E), csx_GetEventMask(9F), csx_RegisterClient(9F), csx_SetEventMask(9F)

PC Card 95 Standard, PCMCIA/JEIDA
NAME
csx_RequestWindow, csx_ReleaseWindow – request or release window resources

SYNOPSIS
#include <sys/pccard.h>

int32_t csx_RequestWindow(client_handle_t ch, window_handle_t *wh, win_req_t *wr);

int32_t csx_ReleaseWindow(window_handle_t wh);

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

PARAMETERS
ch Client handle returned from csx_RegisterClient(9F).
wh Pointer to a window_handle_t structure.
wr Pointer to a win_req_t structure.

DESCRIPTION
The function csx_RequestWindow() requests a block of system address space be assigned to a PC Card in a socket.

The function csx_ReleaseWindow() releases window resources which were obtained by a call to csx_RequestWindow(). No adapter or socket hardware is modified by this function.

The csx_MapMemPage(9F) and csx_ModifyWindow(9F) functions use the window handle returned by csx_RequestWindow(). This window handle must be freed by calling csx_ReleaseWindow() when the client is done using this window.

The PC Card Attribute or Common Memory offset for this window is set by csx_MapMemPage(9F).

STRUCTURE MEMBERS
The structure members of win_req_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* window flags */
uint32_t Base.base; /* requested window */
    /* base address */
    /* base of window */
acc_handle_t Base.handle; /* returned handle for
    /* base of window */
uint32_t Size; /* window size requested */
    /* or granted */
uint32_t win_params.AccessSpeed; /* window access speed */
uint32_t win_params.IOAddrLines; /* IO address lines decoded */
uint32_t ReqOffset; /* required window offset */

The fields are defined as follows:

Socket
Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes
This field is bit-mapped. It is defined as follows:
WIN_MEMORY_TYPE_IO  Window points to I/O space
WIN_MEMORY_TYPE_CM  Window points to Common Memory space
WIN_MEMORY_TYPE_AM  Window points to Attribute Memory space
WIN_ENABLE          Enable window
WIN_DATA_WIDTH_8    Set window to 8-bit data path
WIN_DATA_WIDTH_16   Set window to 16-bit data path
WIN_ACC_NEVER_SWAP  Host endian byte ordering
WIN_ACC_BIG_ENDIAN  Big endian byte ordering
WIN_ACC_LITTLE_ENDIAN Little endian byte ordering
WIN_ACC_STRICT_ORDER Program ordering references
WIN_ACC_UNORDERED_OK May re-order references
WIN_ACC_MERGING_OK  Merge stores to consecutive locations
WIN_ACC_LOADCACHING_OK May cache load operations
WIN_ACC_STORECACHING_OK May cache store operations

WIN_MEMORY_TYPE_IO
WIN_MEMORY_TYPE_CM
WIN_MEMORY_TYPE_AM  These bits select which type of window is being requested. One of these bits must be set.
WIN_ENABLE          The client must set this bit to enable the window.
WIN_ACC_BIG_ENDIAN  These bits describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When either of these bits are set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.
WIN_ACC_LITTLE_ENDIAN
WIN_ACC_NEVER_SWAP  When this is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following Attributes bits, only one of which may be specified:
WIN_ACC_STRICT_ORDER The data references must be issued by a CPU in program order. Strict ordering is the default behavior.
WIN_ACC_UNORDERED_OK  The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).
The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. This bit also implies re-ordering.

The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. This bit also implies merging and re-ordering.

The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. This bit also implies load caching, merging, and re-ordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together.

All other bits in the Attributes field must be set to 0.

On successful return from `csx_RequestWindow()`, `WIN_OFFSET_SIZE` is set in the Attributes field when the client must specify card offsets to `csx_MapMemPage(9F)` that are a multiple of the window size.

**Base.base**
This field must be set to 0 on calling `csx_RequestWindow()`.

**Base.handle**
On successful return from `csx_RequestWindow()`, the Base.handle field contains an access handle corresponding to the first byte of the allocated memory window which the client must use when accessing the PC Card’s memory space via the common access functions. A client must not make any assumptions as to the format of the returned Base.handle field value.

**Size**
On calling `csx_RequestWindow()`, the Size field is the size in bytes of the memory window requested. Size may be zero to indicate that Card Services should provide the smallest sized window available. On successful return from `csx_RequestWindow()`, the Size field contains the actual size of the window allocated.

`csx_ReleaseWindow(9F)`
csx_ReleaseWindow(9F)

**win_params.AccessSpeed**

This field specifies the access speed of the window if the client is requesting a memory window. The AccessSpeed field bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is 0 (noted as reserved in the PC Card 95 Standard), the lower bits are a binary code representing a speed from the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Reserved - do not use).</td>
</tr>
<tr>
<td>1</td>
<td>250 nsec</td>
</tr>
<tr>
<td>2</td>
<td>200 nsec</td>
</tr>
<tr>
<td>3</td>
<td>150 nsec</td>
</tr>
<tr>
<td>4</td>
<td>100 nsec</td>
</tr>
<tr>
<td>5-7</td>
<td>(Reserved—do not use.)</td>
</tr>
</tbody>
</table>

To request a window that supports the WAIT signal, OR-in the WIN_USE_WAIT bit to the AccessSpeed value before calling this function.

It is recommended that clients use the csx_ConvertSpeed(9F) function to generate the appropriate AccessSpeed values rather than manually perturbing the AccessSpeed field.

**win_params.IOAddrLines**

If the client is requesting an I/O window, the IOAddrLines field is the number of I/O address lines decoded by the PC Card in the specified socket. Access to the I/O window is not enabled until csx_RequestConfiguration(9F) has been invoked successfully.

**ReqOffset**

This field is a Solaris-specific extension that can be used by clients to generate optimum window offsets passed to csx_MapMemPage(9F).

**RETURN VALUES**

**CS_SUCCESS**

Successful operation.

**CS_BAD_ATTRIBUTE**

Attributes are invalid.

**CS_BAD_SPEED**

Speed is invalid.

**CS_BAD_HANDLE**

Client handle is invalid.

**CS_BAD_SIZE**

Window size is invalid.
CS NO CARD
    No PC Card in socket.

CS OUT OF RESOURCE
    Unable to allocate window.

CS UNSUPPORTED FUNCTION
    No PCMCIA hardware installed.

CONTEXT
    These functions may be called from user or kernel context.

SEE ALSO
    csx_ConvertSpeed(9F), csx_MapMemPage(9F), csx_ModifyWindow(9F),
    csx_RegisterClient(9F), csx_RequestConfiguration(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_MakeDeviceNode, csx_RemoveDeviceNode – create and remove minor nodes on behalf of the client

#include <sys/pccard.h>

int32_t csx_MakeDeviceNode(client_handle_t ch, make_device_node_t *dn);

int32_t csx_RemoveDeviceNode(client_handle_t ch, remove_device_node_t *dn);

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient(9F).

dn Pointer to a make_device_node_t or remove_device_node_t structure.

csx_MakeDeviceNode() and csx_RemoveDeviceNode() are Solaris-specific extensions to allow the client to request that device nodes in the filesystem are created or removed, respectively, on its behalf.

The structure members of make_device_node_t are:

uint32_t Action; /* device operation */
uint32_t NumDevNodes; /* number of nodes to create */
devnode_desc_t *devnode_desc; /* description of device nodes */

The structure members of remove_device_node_t are:

uint32_t Action; /* device operation */
uint32_t NumDevNodes; /* number of nodes to remove */
devnode_desc_t *devnode_desc; /* description of device nodes */

The structure members of devnode_desc_t are:

char *name; /* device node path and name */
int32_t spec_type; /* device special type (block or char) */
int32_t minor_num; /* device node minor number */
char *node_type; /* device node type */

The Action field is used to specify the operation that csx_MakeDeviceNode() and csx_RemoveDeviceNode() should perform.

The following Action values are defined for csx_MakeDeviceNode():

CREATE_DEVICE_NODE
    Create NumDevNodes minor nodes

The following Action values are defined for csx_RemoveDeviceNode():

REMOVE DEVICE_NODE
    Remove NumDevNodes minor nodes

REMOVE_ALL_DEVICE_NODES
    Remove all minor nodes for this client

232 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
For csx_MakeDeviceNode(), if the Action field is:

CREATE_DEVICE_NODE
The NumDevNodes field must be set to the number of minor devices to create, and the client must allocate the quantity of devnode_desc_t structures specified by NumDevNodes and fill out the fields in the devnode_desc_t structure with the appropriate minor node information. The meanings of the fields in the devnode_desc_t structure are identical to the parameters of the same name to the ddi_create_minor_node(9F) DDI function.

For csx_RemoveDeviceNode(), if the Action field is:

REMOVE_DEVICE_NODE
The NumDevNodes field must be set to the number of minor devices to remove, and the client must allocate the quantity of devnode_desc_t structures specified by NumDevNodes and fill out the fields in the devnode_desc_t structure with the appropriate minor node information. The meanings of the fields in the devnode_desc_t structure are identical to the parameters of the same name to the ddi_remove_minor_node(9F) DDI function.

REMOVE_ALL_DEVICE_NODES
The NumDevNodes field must be set to 0 and the devnode_desc_t structure pointer must be set to NULL. All device nodes for this client will be removed from the filesystem.

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid.
CS_BAD_ATTRIBUTE The value of one or more arguments is invalid.
CS_BAD_ARGS Action is invalid.
CS_OUT_OF_RESOURCE Unable to create or remove device node.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT These functions may be called from user or kernel context.

SEE ALSO csx_RegisterClient(9F), ddi_create_minor_node(9F), ddi_remove_minor_node(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Kernel Functions for Drivers 233
csx_RepGet16(9F)

NAME csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register

SYNOPSIS #include <sys/pccard.h>

void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

INTERFACE Solaris DDI Specific (Solaris DDI)

PARAMETERS

handle The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
hostaddr Source host address.
offset The offset in bytes from the base of the mapped resource.
repcount Number of data accesses to perform.
flags Device address flags.

DESCRIPTION These functions generate multiple reads of various sizes from the mapped memory or device register.

The csx_RepGet8(), csx_RepGet16(), csx_RepGet32(), and csx_RepGet64() functions generate repcount reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data read is stored consecutively into the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum read operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

CONTEXT These functions may be called from user, kernel, or interrupt context.
SEE ALSO

| csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F) |

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register

#include <sys/pccard.h>

void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

Solaris DDI Specific (Solaris DDI)

handle The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).

hostaddr Source host address.

offset The offset in bytes from the base of the mapped resource.

repcount Number of data accesses to perform.

flags Device address flags.

These functions generate multiple reads of various sizes from the mapped memory or device register.

The csx_RepGet8(), csx_RepGet16(), csx_RepGet32(), and csx_RepGet64() functions generate repcount reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data read is stored consecutively into the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum read operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

These functions may be called from user, kernel, or interrupt context.
SEE ALSO

| csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F) |

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RepGet64(9F)

NAME
csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register

SYNOPSIS
#include <sys/pccard.h>

void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

INTERFACE LEVEL PARAMETERS
Solaris DDI Specific (Solaris DDI)

handle The access handle returned from csx_RequestIO(9F),
csx_RequestWindow(9F), or csx_DupHandle(9F).

hostaddr Source host address.

offset The offset in bytes from the base of the mapped resource.

repcount Number of data accesses to perform.

flags Device address flags.

DESCRIPTION
These functions generate multiple reads of various sizes from the mapped memory or device register.

The csx_RepGet8(), csx_RepGet16(), csx_RepGet32(), and csx_RepGet64() functions generate repcount reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data read is stored consecutively into the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum read operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

CONTEXT
These functions may be called from user, kernel, or interrupt context.

238 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
SEE ALSO

- csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F),
- csx_RepPut8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register

**NAME**
csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64

**SYNOPSIS**
```
#include <sys/pccard.h>

void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
```

**INTERFACE LEVEL**
Solaris DDI Specific (Solaris DDI)

**PARAMETERS**
- **handle**
  The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
- **hostaddr**
  Source host address.
- **offset**
  The offset in bytes from the base of the mapped resource.
- **repcount**
  Number of data accesses to perform.
- **flags**
  Device address flags.

**DESCRIPTION**
These functions generate multiple reads of various sizes from the mapped memory or device register.

The csx_RepGet8(), csx_RepGet16(), csx_RepGet32(), and csx_RepGet64() functions generate repcount reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data read is stored consecutively into the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum read operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

**CONTEXT**
These functions may be called from user, kernel, or interrupt context.
SEE ALSO

- `csx_DupHandle(9F)`
- `csx_Get8(9F)`
- `csx_GetMappedAddr(9F)`
- `csx_Put8(9F)`
- `csx_RepPut8(9F)`
- `csx_RequestIO(9F)`
- `csx_RequestWindow(9F)`

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RepPut16(9F)

NAME

csx_RepPut8, csx_RepPut16, csx_RepPut32, csx_RepPut64 – write repetitively to the device register

SYNOPSIS

#include <sys/pccard.h>

void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

handle The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).

hostaddr Source host address.

offset The offset in bytes from the base of the mapped resource.

repcount Number of data accesses to perform.

flags Device address flags.

DESCRIPTION

These functions generate multiple writes of various sizes to the mapped memory or device register.

The csx_RepPut8(), csx_RepPut16(), csx_RepPut32(), and csx_RepPut64() functions generate repcount writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data written is read consecutively from the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum write operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.

CONTEXT

These functions may be called from user, kernel, or interrupt context.
SEE ALSO

| csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F), csx_RepGet8(9F), csx_RequestIO(9F), csx_RequestWindow(9F) |

*PC Card 95 Standard, PCMCIA/JEIDA*
### NAME
`csx_RepPut8`, `csx_RepPut16`, `csx_RepPut32`, `csx_RepPut64` - write repetitively to the device register

### SYNOPSIS
```c
#include <sys/pccard.h>

void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
```

### DESCRIPTION
These functions generate multiple writes of various sizes to the mapped memory or device register.

The `csx_RepPut8()`, `csx_RepPut16()`, `csx_RepPut32()`, and `csx_RepPut64()` functions generate `repcount` writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`. The data written is read consecutively from the buffer pointed to by the host address pointer, `hostaddr`.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the `flags` argument is set to `CS_DEV_AUTOINCR`, these functions increment the device offset, `offset`, after each datum write operation. However, when the `flags` argument is set to `CS_DEV_NO_AUTOINCR`, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.

### CONTEXT
These functions may be called from user, kernel, or interrupt context.
SEE ALSO

- `csx_DupHandle(9F)`, `csx_Get8(9F)`, `csx_GetMappedAddr(9F)`, `csx_Put8(9F)`, `csx_RepGet8(9F)`, `csx_RequestIo(9F)`, `csx_RequestWindow(9F)`

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RepPut8(9F)

NAME   csx_RepPut8, csx_RepPut16, csx_RepPut32, csx_RepPut64 – write repetitively to the device register

SYNOPSIS #include <sys/pccard.h>

void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

INTERFACE Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS handle The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
hostaddr Source host address.
offset The offset in bytes from the base of the mapped resource.
repcount Number of data accesses to perform.
flags Device address flags.

DESCRIPTION These functions generate multiple writes of various sizes to the mapped memory or device register.

The csx_RepPut8(), csx_RepPut16(), csx_RepPut32(), and csx_RepPut64() functions generate repcount writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data written is read consecutively from the buffer pointed to by the host address pointer, hostaddr.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum write operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.

CONTEXT These functions may be called from user, kernel, or interrupt context.
SEE ALSO | csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F),
 | csx_RepGet8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA
## NAME

csx_RepPut8, csx_RepPut16, csx_RepPut32, csx_RepPut64 – write repetitively to the device register

## SYNOPSIS

```c
#include <sys/pccard.h>

void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
```

## INTERFACE LEVEL PARAMETERS

- **handle**: The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- **hostaddr**: Source host address.
- **offset**: The offset in bytes from the base of the mapped resource.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags.

## DESCRIPTION

These functions generate multiple writes of various sizes to the mapped memory or device register.

The `csx_RepPut8()`, `csx_RepPut16()`, `csx_RepPut32()`, and `csx_RepPut64()` functions generate `repcount` writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`. The data written is read consecutively from the buffer pointed to by the host address pointer, `hostaddr`.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the `flags` argument is set to `CS_DEV_AUTOINCR`, these functions increment the device offset, `offset`, after each datum write operation. However, when the `flags` argument is set to `CS_DEV_NO_AUTOINCR`, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.

## CONTEXT

These functions may be called from user, kernel, or interrupt context.
See Also: csx_DupHandle(9F), csx_Get8(9F), csx_GetMappedAddr(9F), csx_Put8(9F),
csx_RepGet8(9F), csx_RequestIO(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Kernel Functions for Drivers  249
csx_RequestConfiguration(9F)

NAME

csx_RequestConfiguration – configure the PC Card and socket

SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_RequestConfiguration(client_handle_t ch, config_req_t *cr);
```

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

- `ch` Client handle returned from csx_RegisterClient(9F).
- `cr` Pointer to a config_req_t structure.

DESCRIPTION

This function configures the PC Card and socket. It must be used by clients that require I/O or IRQ resources for their PC Card.

csx_RequestIO(9F) and csx_RequestIRQ(9F) must be used before calling this function to specify the I/O and IRQ requirements for the PC Card and socket if necessary. csx_RequestConfiguration() establishes the configuration in the socket adapter and PC Card, and it programs the Base and Limit registers of multi-function PC Cards if these registers exist. The values programmed into these registers depend on the IO requirements of this configuration.

STRUCTURE MEMBERS

The structure members of config_req_t are:

- `uint32_t Socket; /* socket number */`
- `uint32_t Attributes; /* configuration attributes */`
- `uint32_t Vcc; /* Vcc value */`
- `uint32_t Vpp1; /* Vpp1 value */`
- `uint32_t Vpp2; /* Vpp2 value */`
- `uint32_t IntType; /* socket interface type - mem or IO */`
- `uint32_t ConfigBase; /* offset from start of AM space */`
- `uint32_t Status; /* value to write to STATUS register */`
- `uint32_t Pin; /* value to write to PRR */`
- `uint32_t Copy; /* value to write to COPY register */`
- `uint32_t ConfigIndex; /* value to write to COR */`
- `uint32_t Present; /* which config registers present */`
- `uint32_t ExtendedStatus; /* value to write to EXSTAT register */`

The fields are defined as follows:

Socket
- Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes
- This field is bit-mapped. It indicates whether the client wishes the IRQ resources to be enabled and whether Card Services should ignore the VS bits on the socket interface. The following bits are defined:

  CONF_ENABLE_IRQ_STEERING
  Enable IRQ Steering. Set to connect the PC Card IREQ line to a system interrupt previously selected by a call to csx_RequestIRQ(9F). If CONF_ENABLE_IRQ_STEERING is set, once csx_RequestConfiguration()
has successfully returned, the client may start receiving IRQ callbacks at the IRQ
callback handler established in the call to csx_RequestIRQ(9F).

CONF_VSOVERRIDE
Override VS pins. After card insertion and prior to the first successful
 csx_RequestConfiguration(), the voltage levels applied to the card shall
be those indicated by the card’s physical key and/or the VS[2:1] voltage sense
pins. For Low Voltage capable host systems (hosts which are capable of VS pin
decoding), if a client desires to apply a voltage not indicated by the VS pin
decoding, then CONF_VSOVERRIDE must be set in the Attributes field;
otherwise, CS_BAD_VCC shall be returned.

Vcc, Vpp1, Vpp2
These fields all represent voltages expressed in tenths of a volt. Values from zero (0)
to 25.5 volts may be set. To be valid, the exact voltage must be available from the
system. PC Cards indicate multiple Vcc voltage capability in their CIS via the
CISTPL_CFTABLE_ENTRY tuple. After card insertion, Card Services processes the
CIS, and when multiple Vcc voltage capability is indicated, Card Services will
allow the client to apply Vcc voltage levels which are contrary to the VS pin
decoding without requiring the client to set CONF_VSOVERRIDE.

IntType
This field is bit-mapped. It indicates how the socket should be configured. The
following bits are defined:

SOCKET_INTERFACE_MEMORY
Memory only interface.

SOCKET_INTERFACE_MEMORY_AND_IO
Memory and I/O interface.

ConfigBase
This field is the offset in bytes from the beginning of attribute memory of the
configuration registers.

Present
This field identifies which of the configuration registers are present. If present, the
corresponding bit is set. This field is bit-mapped as follows:

CONFIG_OPTION_REG_PRESENT
Configuration Option Register (COR) present

CONFIG_STATUS_REG_PRESENT
Configuration Status Register (CCSR) present

CONFIG_PINREPL_REG_PRESENT
Pin Replacement Register (PRR) present

CONFIG_COPY_REG_PRESENT
Socket and Copy Register (SCR) present

CONFIG_ESR_REG_PRESENT
Extended Status Register (ESR) present
csx_RequestConfiguration(9F)

**Status, Pin, Copy, ExtendedStatus**
These fields represent the initial values that should be written to those registers if they are present, as indicated by the Present field.

The Pin field is also used to inform Card Services which pins in the PC Card’s PRR (Pin Replacement Register) are valid. Only those bits which are set are considered valid. This affects how status is returned by the csx_GetStatus(9F) function. If a particular signal is valid in the PRR, both the mask (STATUS) bit and the change (EVENT) bit must be set in the Pin field. The following PRR bit definitions are provided for client use:

- **PRR_WP_STATUS** WRITE PROTECT mask
- **PRR_READY_STATUS** READY mask
- **PRR_BVD2_STATUS** BVD2 mask
- **PRR_BVD1_STATUS** BVD1 mask
- **PRR_WP_EVENT** WRITE PROTECT changed
- **PRR_READY_EVENT** READY changed
- **PRR_BVD2_EVENT** BVD2 changed
- **PRR_BVD1_EVENT** BVD1 changed

**ConfigIndex**
This field is the value written to the COR (Configuration Option Register) for the configuration index required by the PC Card. Only the least significant six bits of the ConfigIndex field are significant; the upper two (2) bits are ignored. The interrupt type in the COR is always set to level mode by Card Services.

**RETURN VALUES**

- **CS_SUCCESS**
  Successful operation.
- **CS_BAD_HANDLE**
  Client handle is invalid or csx_RequestConfiguration() not done.
- **CS_BAD_SOCKET**
  Error in getting or setting socket hardware parameters.
- **CS_BAD_VCC**
  Requested Vcc is not available on socket.
- **CS_BAD_VPP**
  Requested Vpp is not available on socket.
- **CS_NO_CARD**
  No PC Card in socket.
- **CS_BAD_TYPE**
  I/O and memory interface not supported on socket.
- **CS_CONFIGURATION_LOCKED**
  csx_RequestConfiguration() already done.

252 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
CS_UNSUPPORTED_FUNCTION
  No PCMCIA hardware installed.

CONTEXT
  This function may be called from user or kernel context.

SEE ALSO
  csx_AccessConfigurationRegister(9F), csx_GetStatus(9F),
  csx_RegisterClient(9F), csx_ReleaseConfiguration(9F),
  csx_RequestIO(9F), csx_RequestIRQ(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Kernel Functions for Drivers 253
csx_RequestIO(9F)

NAME

csx_RequestIO, csx_ReleaseIO – request or release I/O resources for the client

SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir);

int32_t csx_ReleaseIO(client_handle_t ch, io_req_t *ir);
```

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

| ch          | Client handle returned from csx_RegisterClient(9F). |
| ir          | Pointer to an io_req_t structure. |

DESCRIPTION

The functions csx_RequestIO() and csx_ReleaseIO() request or release, respectively, I/O resources for the client.

If a client requires I/O resources, csx_RequestIO() must be called to request I/O resources from Card Services; then csx_RequestConfiguration(9F) must be used to establish the configuration. csx_RequestIO() can be called multiple times until a successful set of I/O resources is found. csx_RequestConfiguration(9F) only uses the last configuration specified.

csx_RequestIO() fails if it has already been called without a corresponding csx_ReleaseIO().

csx_ReleaseIO() releases previously requested I/O resources. The Card Services window resource list is adjusted by this function. Depending on the adapter hardware, the I/O window might also be disabled.

STRUCTURE MEMBERS

The structure members of io_req_t are:

```
uint32_t Socket; /* socket number*/

uint32_t Baseport1.base; /* IO range base port address */
acc_handle_t Baseport1.handle; /* IO range base address */
   /* or port num */

uint32_t NumPorts1; /* first IO range number contiguous ports */

uint32_t Attributes1; /* first IO range attributes */

uint32_t Baseport2.base; /* IO range base port address */
acc_handle_t Baseport2.handle; /* IO range base or port num */

uint32_t NumPorts2; /* second IO range number contiguous ports */

uint32_t Attributes2; /* second IO range attributes */

uint32_t IOAddrLines; /* number of IO address lines decoded */
```

The fields are defined as follows:

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
Two I/O address ranges can be requested by \texttt{csx\_RequestIO()}. Each I/O address range is specified by the BasePort, NumPorts, and Attributes fields. If only a single I/O range is being requested, the NumPorts2 field must be reset to 0.

When calling \texttt{csx\_RequestIO()}, the BasePort.base field specifies the first port address requested. Upon successful return from \texttt{csx\_RequestIO()}, the BasePort.handle field contains an access handle, corresponding to the first byte of the allocated I/O window, which the client must use when accessing the PC Card’s I/O space via the common access functions. A client must not make any assumptions as to the format of the returned BasePort.handle field value.

If the BasePort.base field is set to 0, Card Services returns an I/O resource based on the available I/O resources and the number of contiguous ports requested. When BasePort.base is 0, Card Services aligns the returned resource in the host system’s I/O address space on a boundary that is a multiple of the number of contiguous ports requested, rounded up to the nearest power of two. For example, if a client requests two I/O ports, the resource returned will be a multiple of two. If a client requests five contiguous I/O ports, the resource returned will be a multiple of eight.

If multiple ranges are being requested, at least one of the BasePort.base fields must be non-zero.

\textbf{NumPorts}

This field is the number of contiguous ports being requested.

\textbf{Attributes}

This field is bit-mapped. The following bits are defined:

\begin{itemize}
  \item \texttt{IO\_DATA\_WIDTH\_8}
    \begin{itemize}
      \item I/O resource uses 8-bit data path.
    \end{itemize}
  \item \texttt{IO\_DATA\_WIDTH\_16}
    \begin{itemize}
      \item I/O resource uses 16-bit data path.
    \end{itemize}
  \item \texttt{WIN\_ACC\_NEVER\_SNAP}
    \begin{itemize}
      \item Host endian byte ordering.
    \end{itemize}
  \item \texttt{WIN\_ACC\_BIG\__ENDIAN}
    \begin{itemize}
      \item Big endian byte ordering.
    \end{itemize}
  \item \texttt{WIN\_ACC\_LITTLE\_Endian}
    \begin{itemize}
      \item Little endian byte ordering.
    \end{itemize}
  \item \texttt{WIN\_ACC\_STRICT\_ORDER}
    \begin{itemize}
      \item Program ordering references.
    \end{itemize}
  \item \texttt{WIN\_ACC\_UNORDERED\_OK}
    \begin{itemize}
      \item May re-order references.
    \end{itemize}
\end{itemize}
WIN_ACC_MERGING_OK
    Merge stores to consecutive locations.

WIN_ACC_LOADCACHING_OK
    May cache load operations.

WIN_ACC_STORECACHING_OK
    May cache store operations.

For some combinations of host system busses and adapter hardware, the width of an I/O resource can not be set via RequestIO(); on those systems, the host bus cycle access type determines the I/O resource data path width on a per-cycle basis.

WIN_ACC_BIG_ENDIAN and WIN_ACC_LITTLE_ENDIAN describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When WIN_ACC_BIG_ENDIAN or WIN_ACC_LITTLE_ENDIAN is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

When WIN_ACC_NEVER_SWAP is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following Attributes bits. Only one of the following bits may be specified:

WIN_ACC_STRICT_ORDER
    The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

WIN_ACC_UNORDERED_OK
    The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

WIN_ACC_MERGING_OK
    The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. IO_MERGING_OK_ACC also implies re-ordering.

WIN_ACC_LOADCACHING_OK
    The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. WIN_ACC_LOADCACHING_OK also implies merging and re-ordering.

WIN_ACC_STORECACHING_OK
    The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. WIN_ACC_STORECACHING_OK also implies load caching, merging, and
These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together. All other bits in the Attributes field must be set to 0.

**RETURN VALUES**

- **CS_SUCCESS**
  - Successful operation.

- **CS_BAD_ATTRIBUTE**
  - Invalid Attributes specified.

- **CS_BAD_BASE**
  - BasePort value is invalid.

- **CS_BAD_HANDLE**
  - Client handle is invalid.

- **CS_CONFIGURATION_LOCKED**
  - csx_RequestConfiguration(9F) has already been done.

- **CS_IN_USE**
  - csx_RequestIO() has already been done without a corresponding csx_ReleaseIO().

- **CS_NO_CARD**
  - No PC Card in socket.

- **CS_BAD_WINDOW**
  - Unable to allocate I/O resources.

- **CS_OUT_OF_RESOURCE**
  - Unable to allocate I/O resources.

- **CS_UNSUPPORTED_FUNCTION**
  - No PCMCIA hardware installed.

**CONTEXT**

These functions may be called from user or kernel context.

**SEE ALSO**

csx_RegisterClient(9F), csx_RequestConfiguration(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
It is important for clients to try to use the minimum amount of I/O resources necessary. One way to do this is for the client to parse the CIS of the PC Card and call `csx_RequestIO()` first with any `IOAddrLines` values that are 0 or that specify a minimum number of address lines necessary to decode the I/O space on the PC Card. Also, if no convenient minimum number of address lines can be used to decode the I/O space on the PC Card, it is important to try to avoid system conflicts with well-known architectural hardware features.
NAME

csx_RequestIRQ, csx_ReleaseIRQ – request or release IRQ resource

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_RequestIRQ(client_handle_t ch, irq_req_t *ir);

int32_t csx_ReleaseIRQ(client_handle_t ch, irq_req_t *ir);

INTERFACE

Solaris DDI Specific (Solaris DDI)

LEVEL

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

ir Pointer to an irq_req_t structure.

DESCRIPTION

The function csx_RequestIRQ() requests an IRQ resource and registers the client’s IRQ handler with Card Services.

If a client requires an IRQ, csx_RequestIRQ() must be called to request an IRQ resource as well as to register the client’s IRQ handler with Card Services. The client will not receive callbacks at the IRQ callback handler until csx_RequestConfiguration(9F) or csx_ModifyConfiguration(9F) has successfully returned when either of these functions are called with the CONF_ENABLE_IRQ_STEERING bit set.

The function csx_ReleaseIRQ() releases a previously requested IRQ resource.

The Card Services IRQ resource list is adjusted by csx_ReleaseIRQ(). Depending on the adapter hardware, the host bus IRQ connection might also be disabled. Client IRQ handlers always run above lock level and so should take care to perform only Solaris operations that are appropriate for an above-lock-level IRQ handler.

csx_RequestIRQ() fails if it has already been called without a corresponding csx_ReleaseIRQ().

STRUCTURE

The structure members of irq_req_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* IRQ attribute flags */
csfunction_t *irq_handler; /* IRQ handler */
void *irq_handler_arg; /* IRQ handler argument */

ddi_iblock_cookie_t *iblk_cookie; /* IRQ interrupt block cookie */

ddi_idevice_cookie_t *idev_cookie; /* IRQ interrupt device cookie */

The fields are defined as follows:

Socket
Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes
This field is bit-mapped. It specifies details about the type of IRQ desired by the client. The following bits are defined:
INTERRUPT REQUESTS

IRQ_TYPE_EXCLUSIVE
IRQ is exclusive to this socket. This bit must be set. It indicates that the system
IRQ is dedicated to this PC Card.

irq_handler
The client IRQ callback handler entry point is passed in the irq_handler field.

irq_handler_arg
The client can use the irq_handler_arg field to pass client-specific data to the
client IRQ callback handler.

iblk_cookie
dev_cookie
These fields must be used by the client to set up mutexes that are used in the
client’s IRQ callback handler.

For a specific csx_ReleaseIRQ() call, the values in the irq_req_t structure must
be the same as those returned from the previous csx_RequestIRQ() call; otherwise,
CS_BAD_ARGS is returned and no changes are made to Card Services resources or the
socket and adapter hardware.

RETURN VALUES

CS_SUCCESS
Successful operation.

CS_BAD_ARGS
IRQ description does not match allocation.

CS_BAD_ATTRIBUTE
IRQ_TYPE_EXCLUSIVE not set, or an unsupported or reserved bit is set.

CS_BAD_HANDLE
Client handle is invalid or csx_RequestConfiguration(9F) not done.

CS_BAD_IRQ
Unable to allocate IRQ resources.

CS_IN_USE
csx_RequestIRQ() already done or a previous csx_RequestIRQ() has not
been done for a corresponding csx_ReleaseIRQ().

CS_CONFIGURATION_LOCKED
csx_RequestConfiguration(9F) already done or
csx_ReleaseConfiguration(9F) has not been done.

CS_NO_CARD
No PC Card in socket.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

CONTEXT
These functions may be called from user or kernel context.

SEE ALSO
csx_ReleaseConfiguration(9F), csx_RequestConfiguration(9F)
csx_RequestSocketMask(9F)

NAME

csx_RequestSocketMask, csx_ReleaseSocketMask – set or clear the client’s client event mask

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_RequestSocketMask(client_handle_t ch, request_socket_mask_t *sm);

int32_t csx_ReleaseSocketMask(client_handle_t ch, release_socket_mask_t *rm);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

PARAMETERS

ch Client handle returned from csx_RegisterClient(9F).

sm Pointer to a request_socket_mask_t structure.

rm Pointer to a release_socket_mask_t structure.

DESCRIPTION

The function csx_RequestSocketMask() sets the client’s client event mask and enables the client to start receiving events at its event callback handler. Once this function returns successfully, the client can start receiving events at its event callback handler. Any pending events generated from the call to csx_RegisterClient(9F) will be delivered to the client after this call as well. This allows the client to set up the event handler mutexes before the event handler gets called.

csx_RequestSocketMask() must be used before calling csx_GetEventMask(9F) or csx_SetEventMask(9F) for the client event mask for this socket.

The function csx_ReleaseSocketMask() clears the client’s client event mask.

STRUCTURE MEMBERS

The structure members of request_socket_mask_t are:

uint32_t Socket; /* socket number */

uint32_t EventMask; /* event mask to set or return */

The structure members of release_socket_mask_t are:

uint32_t Socket; /* socket number */

The fields are defined as follows:

Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

EventMask This field is bit-mapped. Card Services performs event notification based on this field. See csx_event_handler(9E) for valid event definitions and for additional information about handling events.

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_IN_USE csx_ReleaseSocketMask() has not been done.

262 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
csx_RequestSocketMask(9F)

<table>
<thead>
<tr>
<th>CS_BAD_SOCKET</th>
<th>csx_RequestSocketMask() has not been done.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

CONTEXT

These functions may be called from user or kernel context.

SEE ALSO

csx_event_handler(9E), csx_GetEventMask(9F), csx_RegisterClient(9F), csx_SetEventMask(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
csx_RequestWindow(9F)

NAME

csx_RequestWindow, csx_ReleaseWindow – request or release window resources

SYNOPSIS

```
#include <sys/pccard.h>

int32_t csx_RequestWindow(client_handle_t ch, window_handle_t *wh, win_reg_t *wr);

int32_t csx_ReleaseWindow(window_handle_t wh);
```

INTERFACE

Solaris DDI Specific (Solaris DDI)

PARAMETERS

<table>
<thead>
<tr>
<th>ch</th>
<th>Client handle returned from csx_RegisterClient(9F).</th>
</tr>
</thead>
<tbody>
<tr>
<td>wh</td>
<td>Pointer to a window_handle_t structure.</td>
</tr>
<tr>
<td>wr</td>
<td>Pointer to a win_reg_t structure.</td>
</tr>
</tbody>
</table>

DESCRIPTION

The function \textit{csx\_RequestWindow()} requests a block of system address space be assigned to a PC Card in a socket.

The function \textit{csx\_ReleaseWindow()} releases window resources which were obtained by a call to \textit{csx\_RequestWindow()}. No adapter or socket hardware is modified by this function.

The \textit{csx\_MapMemPage(9F)} and \textit{csx\_ModifyWindow(9F)} functions use the window handle returned by \textit{csx\_RequestWindow()}. This window handle must be freed by calling \textit{csx\_ReleaseWindow()} when the client is done using this window.

The PC Card Attribute or Common Memory offset for this window is set by \textit{csx\_MapMemPage(9F)}.

STRUCTURE MEMBERS

The structure members of \textit{win_reg_t} are:

```
uint32_t Socket; /* socket number */
uint32_t Attributes; /* window flags */
uint32_t Base.base; /* requested window */
    /* base address */
acc_handle_t Base.handle; /* returned handle for */
    /* base of window */
uint32_t Size; /* window size granted */
    /* or granted */
uint32_t win_params.AccessSpeed; /* window access speed */
uint32_t win_params.IOAddrLines; /* IO address lines decoded */
uint32_t ReqOffset; /* required window offset */
```

The fields are defined as follows:

Socket

Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes

This field is bit-mapped. It is defined as follows:

---

264 man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Jul 1996
WIN_MEMORY_TYPE_IO  Window points to I/O space
WIN_MEMORY_TYPE_CM  Window points to Common Memory space
WIN_MEMORY_TYPE_AM  Window points to Attribute Memory space
WIN_ENABLE         Enable window
WIN_DATA_WIDTH_8   Set window to 8-bit data path
WIN_DATA_WIDTH_16  Set window to 16-bit data path
WIN_ACC_NEVER_SWAP Host endian byte ordering
WIN_ACC_BIG_ENDIAN Big endian byte ordering
WIN_ACC_LITTLE_ENDIAN Little endian byte ordering
WIN_ACC_STRICT_ORDER Program ordering references
WIN_ACC_UNORDERED_OK May re-order references
WIN_ACC_MERGING_OK  Merge stores to consecutive locations
WIN_ACC_LOADCACHING_OK May cache load operations
WIN_ACC_STORECACHING_OK May cache store operations

WIN_MEMORY_TYPE_IO
WIN_MEMORY_TYPE_CM
WIN_MEMORY_TYPE_AM

These bits select which type of window is being requested. One of these bits must be set.

WIN_ENABLE

The client must set this bit to enable the window.

WIN_ACC_BIG_ENDIAN

WIN_ACC_LITTLE_ENDIAN

These bits describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When either of these bits are set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

WIN_ACC_NEVER_SWAP

When this is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following Attributes bits, only one of which may be specified:

WIN_ACC_STRICT_ORDER

The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

WIN_ACC_UNORDERED_OK

The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

Kernel Functions for Drivers  265
The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. This bit also implies re-ordering.

The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. This bit also implies merging and re-ordering.

The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. This bit also implies load caching, merging, and re-ordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together.

All other bits in the Attributes field must be set to 0.

On successful return from `csx_RequestWindow()`, WIN_OFFSET_SIZE is set in the Attributes field when the client must specify card offsets to `csx_MapMemPage(9F)` that are a multiple of the window size.

```
Base.base
   This field must be set to 0 on calling csx_RequestWindow().
```

```
Base.handle
   On successful return from csx_RequestWindow(), the Base.handle field contains an access handle corresponding to the first byte of the allocated memory window which the client must use when accessing the PC Card's memory space via the common access functions. A client must not make any assumptions as to the format of the returned Base.handle field value.
```

```
Size
   On calling csx_RequestWindow(), the Size field is the size in bytes of the memory window requested. Size may be zero to indicate that Card Services should provide the smallest sized window available. On successful return from csx_RequestWindow(), the Size field contains the actual size of the window allocated.
```
**csx_RequestWindow(9F)**

**win_params.AccessSpeed**

This field specifies the access speed of the window if the client is requesting a memory window. The AccessSpeed field bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is 0 (noted as reserved in the *PC Card 95 Standard*), the lower bits are a binary code representing a speed from the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Reserved - do not use).</td>
</tr>
<tr>
<td>1</td>
<td>250 nsec</td>
</tr>
<tr>
<td>2</td>
<td>200 nsec</td>
</tr>
<tr>
<td>3</td>
<td>150 nsec</td>
</tr>
<tr>
<td>4</td>
<td>100 nsec</td>
</tr>
<tr>
<td>5-7</td>
<td>(Reserved—do not use.)</td>
</tr>
</tbody>
</table>

To request a window that supports the WAIT signal, OR-in the WIN_USE_WAIT bit to the AccessSpeed value before calling this function.

It is recommended that clients use the `csx_ConvertSpeed(9F)` function to generate the appropriate AccessSpeed values rather than manually perturbing the AccessSpeed field.

**win_params.IOAddrLines**

If the client is requesting an I/O window, the IOAddrLines field is the number of I/O address lines decoded by the PC Card in the specified socket. Access to the I/O window is not enabled until `csx_RequestConfiguration(9F)` has been invoked successfully.

**ReqOffset**

This field is a Solaris-specific extension that can be used by clients to generate optimum window offsets passed to `csx_MapMemPage(9F)`.

**RETURN VALUES**

- **CS_SUCCESS**
  - Successful operation.
- **CS_BAD_ATTRIBUTE**
  - Attributes are invalid.
- **CS_BAD_SPEED**
  - Speed is invalid.
- **CS_BAD_HANDLE**
  - Client handle is invalid.
- **CS_BAD_SIZE**
  - Window size is invalid.
CS_NO_CARD
   No PC Card in socket.

CS_OUT_OF_RESOURCE
   Unable to allocate window.

CS_UNSUPPORTED_FUNCTION
   No PCMCIA hardware installed.

CONTEXT These functions may be called from user or kernel context.

SEE ALSO csx_ConvertSpeed(9F), csx_MapMemPage(9F), csx_ModifyWindow(9F),
csx_RegisterClient(9F), csx_RequestConfiguration(9F)

PC Card 95 Standard, PCMCIA/JEIDA
csx_ResetFunction – reset a function on a PC card

#include <sys/pccard.h>

int32_t csx_ResetFunction(client_handle_t ch, reset_function_t *rf);

Solaris DDI Specific (Solaris DDI)

ch Client handle returned from csx_RegisterClient()

rf Pointer to a reset_function_t structure.

csx_ResetFunction() requests that the specified function on the PC card initiate a reset operation.

The structure members of reset_function_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* reset attributes */

The fields are defined as follows:

Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes Must be 0.

CS_SUCCESS Card Services has noted the reset request.
CS_IN_USE This Card Services implementation does not permit configured cards to be reset.
CS_BAD_HANDLE Client handle is invalid.
CS_NO_CARD No PC card in socket.
CS_BAD_SOCKET Specified socket or function number is invalid.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

This function may be called from user or kernel context.

csx_event_handler(), csx_RegisterClient()

PC Card 95 Standard, PCMCIA/JEIDA

CS_RESET_FUNCTION() has not been implemented in this release and always returns CS_IN_USE.
### NAME

csx_SetEventMask, csx_GetEventMask – set or return the client event mask for the client.

### SYNOPSIS

```c
#include <sys/pccard.h>

int32_t csx_SetEventMask(client_handle_t ch, sockevent_t *se);
int32_t csx_GetEventMask(client_handle_t ch, sockevent_t *se);
```

### INTERFACE LEVEL PARAMETERS

- `ch` Client handle returned from csx_RegisterClient(9F).
- `se` Pointer to a `sockevent_t` structure.

### DESCRIPTION

The function `csx_SetEventMask()` sets the client or global event mask for the client.

The function `csx_GetEventMask()` returns the client or global event mask for the client.

`csx_RequestSocketMask(9F)` must be called before calling `csx_SetEventMask()` for the client event mask for this socket.

### STRUCTURE MEMBERS

The structure members of `sockevent_t` are:

```c
uint32_t uint32_t /* attribute flags for call */
uint32_t EventMask; /* event mask to set or return */
uint32_t Socket; /* socket number if necessary */
```

The fields are defined as follows:

- **Attributes**
  This is a bit-mapped field that identifies the type of event mask to be returned. The field is defined as follows:

  - `CONF_EVENT_MASK_GLOBAL` Client’s global event mask. If set, the client’s global event mask is returned.
  - `CONF_EVENT_MASK_CLIENT` Client’s local event mask. If set, the client’s local event mask is returned.

- **EventMask**
  This field is bit-mapped. Card Services performs event notification based on this field. See `csx_event_handler(9E)` for valid event definitions and for additional information about handling events.

- **Socket**
  Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

### RETURN VALUES

- **CS_SUCCESS** Successful operation.
- **CS_BAD_HANDLE** Client handle is invalid.
csx_SetEventMask(9F)

<table>
<thead>
<tr>
<th>CS_BAD_SOCKET</th>
<th>csx_RequestSocketMask(9F) not called for CONF_EVENT_MASK_CLIENT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT** These functions may be called from user or kernel context.

**SEE ALSO** csx_event_handler(9E), csx_RegisterClient(9F),
csx_ReleaseSocketMask(9F), csx_RequestSocketMask(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
### NAME
csx_SetHandleOffset(9F)

### SYNOPSIS
```c
#include <sys/pccard.h>

int32_t csx_SetHandleOffset(acc_handle_t handle, uint32_t offset);
```

### INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

### PARAMETERS
- **handle**: Access handle returned by `csx_RequestIRQ(9F)` or `csx_RequestIO(9F)`.
- **offset**: New access handle offset.

### DESCRIPTION
This function sets the current offset for the access handle, `handle`, to `offset`.

### RETURN VALUES
- **CS_SUCCESS**: Successful operation.

### CONTEXT
This function may be called from user or kernel context.

### SEE ALSO
- `csx_GetHandleOffset(9F)`, `csx_RequestIO(9F)`, `csx_RequestIRQ(9F)`
- `PC Card 95 Standard`, PCMCIA/JEIDA
#include <sys/pccard.h>

int32_t csx_ValidateCIS(client_handle_t ch, cisinfo_t *ci);

This function validates the Card Information Structure (CIS) on the PC Card in the specified socket.

The structure members of cisinfo_t are:

- **Socket**
  Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

- **Chains**
  This field returns the number of valid tuple chains located in the CIS. If 0 is returned, the CIS is not valid.

- **Tuples**
  This field is a Solaris-specific extension and it returns the total number of tuples on all the chains in the PC Card’s CIS.

CS_SUCCESS
Successful operation.

CS_NO_CIS
No CIS on PC Card or CIS is invalid.

CS_NO_CARD
No PC Card in socket.

CS_UNSUPPORTED_FUNCTION
No PCMCIA hardware installed.

This function may be called from user or kernel context.

see also:
- csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_ParseTuple(9F),
- csx_RegisterClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cvbroadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

NAME
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
cvp A pointer to an abstract data type kcondvar_t.
mp A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)
type The constant CV_DRIVER.
arg A type-specific argument, drivers should pass arg as NULL.
timeout A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.

DESCRIPTION
Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with cv_signal() or
cv_broadcast(), and then release the mutex.

cv_wait() suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal() signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling cv_broadcast(). You must acquire the mutex
passed into cv_wait() before calling cv_signal() or cv_broadcast().

The function cv_wait() is similar to cv_wait() but returns 0 if a signal (for
example, by kill(2)) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function cv_wait() is similar to cv_wait(), except that it returns -1
without the condition being signaled after the timeout time has been reached.

The function cv_timedwait() is similar to cv_wait() and
cv_wait(), except that it returns -1 without the condition being signaled after
the timeout time has been reached, or 0 if a signal (for example, by kill(2)) is sent to
the thread.

For both cv_timedwait() and cv_timedwait(), time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
ddi_get_lbolt().

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>For cv_wait(), cv_wait() or cv_timedwait() indicates that the condition was met and the function returned because a call to cv_signal() or cv_broadcast() was pending.</td>
</tr>
<tr>
<td>-1</td>
<td>For cv_wait() or cv_timedwait() indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>For cv_wait(), cv_wait() or cv_timedwait() indicates that the condition was met and the function returned because a call to cv_signal() or cv_broadcast() was pending.</td>
</tr>
</tbody>
</table>

These functions can be called from user, kernel or interrupt context. In most cases,
however, cv_wait(), cv_timedwait(), cv_wait(), and
cv_timedwait() should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If cv_wait(), cv_timedwait(), cv_wait() or cv_timedwait() are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**EXAMPLE 1** Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**NOTES**

It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwaitSig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

**SEE ALSO**

`kill(2)`, `ddi_get_lbolt(9F)`, `mutex(9F)`, `mutex_init(9F)`

*Writing Device Drivers*
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, 
cv_timedwait, cv_timedwait_sig – condition variable routines

#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

NAME

condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, 
cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS

#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI).

cvp A pointer to an abstract data type kcondvar_t.

mp A pointer to a mutual exclusion lock (kmutex_t), initialized by 
mutex_init(9F) and held by the caller.

name Descriptive string. This is obsolete and should be NULL. 
(Non-NULL strings are legal, but they’re a waste of kernel memory.)

type The constant CV_DRIVER.

arg A type-specific argument, drivers should pass arg as NULL.

timeout A time, in absolute ticks since boot, when cv_timedwait() or 
cv_timedwait_sig() should return.

DESCRIPTION

Condition variables are a standard form of thread synchronization. They are designed 
to be used with mutual exclusion locks (mutexes). The associated mutex is used to 
sure that a condition can be checked atomically and that the thread can block on the 
associated condition variable without missing either a change to the condition or a 
signal that the condition has changed. Condition variables must be initialized by 
calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, 
data structure reference count, etc.) while holding a mutex which keeps other threads 
from changing the condition. If the condition is such that the thread should block, 
cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with cv_signal() or
cv_broadcast(), and then release the mutex.

cv_wait() suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal() signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling cv_broadcast(). You must acquire the mutex
passed into cv_wait() before calling cv_signal() or cv_broadcast().

The function cv_wait_sig() is similar to cv_wait() but returns 0 if a signal (for
example, by kill(2)) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function cv_timedwait() is similar to cv_wait(), except that it returns -1
without the condition being signaled after the timeout time has been reached.

The function cv_timedwait_sig() is similar to cv_timedwait() and
cv_wait_sig(), except that it returns -1 without the condition being signaled after
the timeout time has been reached, or 0 if a signal (for example, by kill(2)) is sent to
the thread.

For both cv_timedwait() and cv_timedwait_sig(), time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
ddi_get_lbolt(9F).

**RETURN VALUES**

0 For cv_wait_sig() and cv_timedwait_sig() indicates that
the condition was not necessarily signaled and the function
returned because a signal (as in kill(2)) was pending.

-1 For cv_timedwait() and cv_timedwait_sig() indicates that
the condition was not necessarily signaled and the function
returned because the timeout time was reached.

>0 For cv_wait_sig(), cv_timedwait() or cv_timedwait_sig
() indicates that the condition was met and the function returned
due to a call to cv_signal() or cv_broadcast(), or due to a
premature wakeup (see NOTES).

**CONTEXT**

These functions can be called from user, kernel or interrupt context. In most cases,
however, cv_wait(), cv_timedwait(), cv_wait_sig(), and
cv_timedwait_sig() should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If cv_wait(), cv_timedwait(), cv_wait_sig(), or cv_timedwait_sig() are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**EXAMPLE 1** Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**NOTES**

It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay()`. The `delay()` function calls `timeout()`, which can be subject to priority inversions.

**SEE ALSO** `kill(2)`, `ddi_get_lbolt()`, `mutex()`, `mutex_init()`, `cv_destroy()`
### NAME
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

### SYNOPSIS
```c
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
oclock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
oclock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
```

### INTERFACE LEVEL PARAMETERS
- **cvp**: A pointer to an abstract data type `kcondvar_t`.
- **mp**: A pointer to a mutual exclusion lock (`kmutex_t`), initialized by `mutex_init(9F)` and held by the caller.
- **name**: Descriptive string. This is obsolete and should be `NULL`. (Non-`NULL` strings are legal, but they’re a waste of kernel memory.)
- **type**: The constant `CV_DRIVER`.
- **arg**: A type-specific argument, drivers should pass `arg` as `NULL`.
- **timeout**: A time, in absolute ticks since boot, when `cv_timedwait()` or `cv_timedwait_sig()` should return.

### DESCRIPTION
Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling `cv_init()`, and must be deallocated by calling `cv_destroy()`.

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, `cv_wait()` is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with \texttt{cv\_signal()} or \texttt{cv\_broadcast()}, and then release the mutex.

\texttt{cv\_wait()} suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

\texttt{cv\_signal()} signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling \texttt{cv\_broadcast()}. You must acquire the mutex passed into \texttt{cv\_wait()} before calling \texttt{cv\_signal()} or \texttt{cv\_broadcast()}.

The function \texttt{cv\_wait\_sig()} is similar to \texttt{cv\_wait()} but returns 0 if a signal (for example, by \texttt{kill(2)}) is sent to the thread. In any case, the mutex is reacquired before returning.

The function \texttt{cv\_timedwait()} is similar to \texttt{cv\_wait()}, except that it returns -1 without the condition being signaled after the timeout time has been reached.

The function \texttt{cv\_timedwait\_sig()} is similar to \texttt{cv\_timedwait()} and \texttt{cv\_wait\_sig()}, except that it returns -1 without the condition being signaled after the timeout time has been reached, or 0 if a signal (for example, by \texttt{kill(2)}) is sent to the thread.

For both \texttt{cv\_timedwait()} and \texttt{cv\_timedwait\_sig()}, time is in absolute clock ticks since the last system reboot. The current time may be found by calling \texttt{ddi\_get\_lbolt(9F)}.

**RETURN VALUES**

0  
For \texttt{cv\_wait\_sig()} and \texttt{cv\_timedwait\_sig()} indicates that the condition was not necessarily signaled and the function returned because a signal (as in \texttt{kill(2)}) was pending.

-1  
For \texttt{cv\_timedwait()} and \texttt{cv\_timedwait\_sig()} indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.

>0  
For \texttt{cv\_wait\_sig()}, \texttt{cv\_timedwait()} or \texttt{cv\_timedwait\_sig()} indicates that the condition was met and the function returned due to a call to \texttt{cv\_signal()} or \texttt{cv\_broadcast()}, or due to a premature wakeup (see NOTES).

**CONTEXT**

These functions can be called from user, kernel or interrupt context. In most cases, however, \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, and \texttt{cv\_timedwait\_sig()} should not be called from interrupt context, and cannot be called from a high-level interrupt context.

If \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, or \texttt{cv\_timedwait\_sig()} are used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the wakeup becomes blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**EXAMPLE 1** Waiting for a Flag Value in a Driver's Unit

Here the condition being waited for is a flag value in a driver's unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**NOTES**

It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait Sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

**SEE ALSO**

`kill(2), ddi_get_lbolt(9F), mutex(9F), mutex_init(9F)`

*Writing Device Drivers*
NAME
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig,
cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
cvp A pointer to an abstract data type kcondvar_t.
mp A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)
type The constant CV_DRIVER.
arg A type-specific argument, drivers should pass arg as NULL.
timeout A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.

DESCRIPTION
Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with \texttt{cv\_signal()} or
\texttt{cv\_broadcast()}, and then release the mutex.

\texttt{cv\_wait()} suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

\texttt{cv\_signal()} signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling \texttt{cv\_broadcast()}. You must acquire the mutex
passed into \texttt{cv\_wait()} before calling \texttt{cv\_signal()} or \texttt{cv\_broadcast()}.

The function \texttt{cv\_wait\_sig()} is similar to \texttt{cv\_wait()} but returns \texttt{0} if a signal (for
example, by \texttt{kill(2)}) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function \texttt{cv\_timedwait()} is similar to \texttt{cv\_wait()}, except that it returns \texttt{-1}
without the condition being signaled after the timeout time has been reached.

The function \texttt{cv\_timedwait\_sig()} is similar to \texttt{cv\_timedwait()} and
\texttt{cv\_wait\_sig()}, except that it returns \texttt{-1} without the condition being signaled after
the timeout time has been reached, or \texttt{0} if a signal (for example, by \texttt{kill(2)}) is sent to
the thread.

For both \texttt{cv\_timedwait()} and \texttt{cv\_timedwait\_sig()}, time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
\texttt{ddi\_get\_lbolt(9F)}.

\textbf{RETURN VALUES}

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{0}</td>
<td>\texttt{cv_wait_sig()} and \texttt{cv_timedwait_sig()} indicates that the condition was not necessarily signaled and the function returned because a signal (as in \texttt{kill(2)}) was pending.</td>
</tr>
<tr>
<td>\texttt{-1}</td>
<td>\texttt{cv_timedwait()} and \texttt{cv_timedwait_sig()} indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.</td>
</tr>
<tr>
<td>&gt;\texttt{0}</td>
<td>\texttt{cv_wait_sig()}, \texttt{cv_timedwait()} or \texttt{cv_timedwait_sig()} indicates that the condition was met and the function returned due to a call to \texttt{cv_signal()} or \texttt{cv_broadcast()}, or due to a premature wakeup (see NOTES).</td>
</tr>
</tbody>
</table>

\textbf{CONTEXT}

These functions can be called from user, kernel or interrupt context. In most cases,
however, \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, and
\texttt{cv\_timedwait\_sig()} should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, or \texttt{cv\_timedwait\_sig()} are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**EXAMPLE 1** Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY) cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**NOTES**

It is possible for \texttt{cv\_wait()}, \texttt{cv\_wait\_sig()}, \texttt{cv\_timedwait()}, and \texttt{cv\_timedwait\_sig()} to return prematurely, that is, not due to a call to \texttt{cv\_signal()} or \texttt{cv\_broadcast()}. This occurs most commonly in the case of \texttt{cv\_wait\_sig()} and \texttt{cv\_timedwait\_sig()} when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for \texttt{cv\_wait()}. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use \texttt{cv\_timedwait()} rather than \texttt{delay()}(9F). The \texttt{delay()} function calls \texttt{timeout()}(9F), which can be subject to priority inversions.

**SEE ALSO** \texttt{kill(2)}, \texttt{ddi\_get\_lbolt(9F)}, \texttt{mutex(9F)}, \texttt{mutex\_init(9F)}

*Writing Device Drivers*
cv_timedwait(9F)

NAME | condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);

void cv_destroy(kcondvar_t *cvp);

void cv_wait(kcondvar_t *cvp, kmutex_t *mp);

void cv_signal(kcondvar_t *cvp);

void cv_broadcast(kcondvar_t *cvp);

int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);

clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS

cvp | A pointer to an abstract data type kcondvar_t.

mp | A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.

name | Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)

type | The constant CV_DRIVER.

arg | A type-specific argument, drivers should pass arg as NULL.

timeout | A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.

DESCRIPTION
Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, cv_wait() is called with a related condition variable and the mutex. At some later
At a point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with \texttt{cv\_signal()} or \texttt{cv\_broadcast()}, and then release the mutex.

\texttt{cv\_wait()} suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

\texttt{cv\_signal()} signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling \texttt{cv\_broadcast()}. You must acquire the mutex passed into \texttt{cv\_wait()} before calling \texttt{cv\_signal()} or \texttt{cv\_broadcast()}. The function \texttt{cv\_wait\_sig()} is similar to \texttt{cv\_wait()} but returns 0 if a signal (for example, by \texttt{kill(2)}) is sent to the thread. In any case, the mutex is reacquired before returning.

The function \texttt{cv\_timedwait()} is similar to \texttt{cv\_wait()}, except that it returns \(-1\) without the condition being signaled after the timeout time has been reached.

The function \texttt{cv\_timedwait\_sig()} is similar to \texttt{cv\_timedwait()} and \texttt{cv\_wait\_sig()}, except that it returns \(-1\) without the condition being signaled after the timeout time has been reached, or 0 if a signal (for example, by \texttt{kill(2)}) is sent to the thread.

For both \texttt{cv\_timedwait()} and \texttt{cv\_timedwait\_sig()}, time is in absolute clock ticks since the last system reboot. The current time may be found by calling \texttt{ddi\_get\_lbolt(9F)}.

**RETURN VALUES**

- 0: For \texttt{cv\_wait\_sig()} and \texttt{cv\_timedwait\_sig()} indicates that the condition was not necessarily signaled and the function returned because a signal (as in \texttt{kill(2)}) was pending.
- \(-1\): For \texttt{cv\_timedwait()} and \texttt{cv\_timedwait\_sig()} indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.
- \(>0\): For \texttt{cv\_wait\_sig()}, \texttt{cv\_timedwait()} or \texttt{cv\_timedwait\_sig()} indicates that the condition was met and the function returned due to a call to \texttt{cv\_signal()} or \texttt{cv\_broadcast()}, or due to a premature wakeup (see NOTES).

**CONTEXT**

These functions can be called from user, kernel or interrupt context. In most cases, however, \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, and \texttt{cv\_timedwait\_sig()} should not be called from interrupt context, and cannot be called from a high-level interrupt context.

If \texttt{cv\_wait()}, \texttt{cv\_timedwait()}, \texttt{cv\_wait\_sig()}, or \texttt{cv\_timedwait\_sig()} are used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the wakeup becomes blocked on anything that requires the lower-priority interrupt, the system will hang.
cv_timedwait(9F)

For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

EXAMPLES

EXAMPLE 1 Waiting for a Flag Value in a Driver's Unit

Here the condition being waited for is a flag value in a driver's unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

EXAMPLE 2 Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

NOTES

It is possible for cv_wait(), cv_wait_sig(), cv_timedwait(), and cv_timedwait_sig() to return prematurely, that is, not due to a call to cv_signal() or cv_broadcast(). This occurs most commonly in the case of cv_wait_sig() and cv_timedwait сиг() when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for cv_wait(). Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use cv_timedwait() rather than delay(9F). The delay() function calls timeout(9F), which can be subject to priority inversions.

SEE ALSO

kill(2), ddi_get_lbolt(9F), mutex(9F), mutex_init(9F)

Writing Device Drivers
## NAME
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig,
cv_timedwait, cv_timedwait_sig – condition variable routines

## SYNOPSIS

```
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
```

## INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

## PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvp</td>
<td>A pointer to an abstract data type kcondvar_t.</td>
</tr>
<tr>
<td>mp</td>
<td>A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.</td>
</tr>
<tr>
<td>name</td>
<td>Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)</td>
</tr>
<tr>
<td>type</td>
<td>The constant CV_DRIVER.</td>
</tr>
<tr>
<td>arg</td>
<td>A type-specific argument, drivers should pass arg as NULL.</td>
</tr>
<tr>
<td>timeout</td>
<td>A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.</td>
</tr>
</tbody>
</table>

## DESCRIPTION

Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with `cv_signal()` or `cv_broadcast()`, and then release the mutex.

`cv_wait()` suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

`cv_signal()` signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling `cv_broadcast()`. You must acquire the mutex passed into `cv_wait()` before calling `cv_signal()` or `cv_broadcast()`.

The function `cv_wait_sig()` is similar to `cv_wait()` but returns 0 if a signal (for example, by `kill(2)`) is sent to the thread. In any case, the mutex is reacquired before returning.

The function `cv_timedwait()` is similar to `cv_wait()`, except that it returns -1 without the condition being signaled after the timeout time has been reached.

The function `cv_timedwait_sig()` is similar to `cv_timedwait()` and `cv_wait_sig()`, except that it returns -1 without the condition being signaled after the timeout time has been reached, or 0 if a signal (for example, by `kill(2)`) is sent to the thread.

For both `cv_timedwait()` and `cv_timedwait_sig()`, time is in absolute clock ticks since the last system reboot. The current time may be found by calling `ddi_get_lbolt(9F)`.

**RETURN VALUES**

- 0 For `cv_wait_sig()` and `cv_timedwait_sig()` indicates that the condition was not necessarily signaled and the function returned because a signal (as in `kill(2)`) was pending.
- -1 For `cv_timedwait()` and `cv_timedwait_sig()` indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.
- >0 For `cv_wait_sig()`, `cv_timedwait()` or `cv_timedwait_sig()` indicates that the condition was met and the function returned due to a call to `cv_signal()` or `cv_broadcast()`, or due to a premature wakeup (see NOTES).

**CONTEXT**

These functions can be called from user, kernel or interrupt context. In most cases, however, `cv_wait()`, `cv_timedwait()`, `cv_wait_sig()`, and `cv_timedwait_sig()` should not be called from interrupt context, and cannot be called from a high-level interrupt context.

If `cv_wait()`, `cv_timedwait()`, `cv_wait_sig()`, or `cv_timedwait_sig()` are used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the wakeup becomes blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

EXAMPLE 1 Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

EXAMPLE 2 Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

NOTES It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

SEE ALSO `kill(2)`, `ddi_get_lbolt(9F)`, `mutex(9F)`, `mutex_init(9F)`

`Writing Device Drivers`
### Condition Variable Routines

This file describes the condition variable routines provided in Solaris DDI. These routines are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling `cv_init()`, and must be deallocated by calling `cv_destroy()`.

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, `cv_wait()` is called with a related condition variable and the mutex. At some later time, the thread may be woken up by a `cv_signal()` or `cv_broadcast()` call.

#### Interface Level

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| `cvp`     | A pointer to an abstract data type `kcondvar_t`.
| `mp`      | A pointer to a mutual exclusion lock (`kmutex_t`), initialized by `mutex_init(9F)` and held by the caller.
| `name`    | Descriptive string. This is obsolete and should be `NULL`. (Non-`NULL` strings are legal, but they’re a waste of kernel memory.)
| `type`    | The constant `CV_DRIVER`.
| `arg`     | A type-specific argument, drivers should pass `arg` as `NULL`.
| `timeout` | A time, in absolute ticks since boot, when `cv_timedwait()` or `cv_timedwait_sig()` should return.

#### Description

Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling `cv_init()`, and must be deallocated by calling `cv_destroy()`.

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, `cv_wait()` is called with a related condition variable and the mutex. At some later time, the thread may be woken up by a `cv_signal()` or `cv_broadcast()` call.

#### Synopsis

```c
#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
```
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with cv_signal() or
cv_broadcast(), and then release the mutex.

cv_wait() suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal() signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling cv_broadcast(). You must acquire the mutex
passed into cv_wait() before calling cv_signal() or cv_broadcast().

The function cv_wait_sig() is similar to cv_wait() but returns 0 if a signal (for
example, by kill(2)) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function cv_timedwait() is similar to cv_wait(), except that it returns −1
without the condition being signaled after the timeout time has been reached.

The function cv_timedwait_sig() is similar to cv_timedwait() and
cv_wait_sig(), except that it returns −1 without the condition being signaled after
the timeout time has been reached, or 0 if a signal (for example, by kill(2)) is sent to
the thread.

For both cv_timedwait() and cv_timedwait_sig(), time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
ddi_get_lbolt(9F).

RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>For cv_wait_sig() and cv_timedwait_sig() indicates that the condition was not necessarily signaled and the function returned because a signal (as in kill(2)) was pending.</td>
</tr>
<tr>
<td>−1</td>
<td>For cv_timedwait() and cv_timedwait_sig() indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>For cv_wait_sig(), cv_timedwait() or cv_timedwait_sig() indicates that the condition was met and the function returned due to a call to cv_signal() or cv_broadcast(), or due to a premature wakeup (see NOTES).</td>
</tr>
</tbody>
</table>

CONTEXT

These functions can be called from user, kernel or interrupt context. In most cases,
however, cv_wait(), cv_timedwait(), cv_wait_sig(), and
cv_timedwait_sig() should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If cv_wait(), cv_timedwait(), cv_wait_sig(), or cv_timedwait_sig() are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

EXAMPLES

EXAMPLE 1 Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

EXAMPLE 2 Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

NOTES

It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

SEE ALSO

`kill(2), ddi_get_lbolt(9F), mutex(9F), mutex_init(9F)`

Writing Device Drivers
condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig,
cv_timedwait, cv_timedwait_sig – condition variable routines

#include <sys/ksynch.h>

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
name A descriptive string. This is obsolete and should be NULL.
(Non-NULL strings are legal, but they’re a waste of kernel memory.)
type The constant CV_DRIVER.
arg A type-specific argument, drivers should pass arg as NULL.
timeout A time, in absolute ticks since boot, when cv_timedwait() or
 cv_timedwait_sig() should return.

DESCRIPTION
Condition variables are a standard form of thread synchronization. They are designed
to be used with mutual exclusion locks (mutexes). The associated mutex is used to
ensure that a condition can be checked atomically and that the thread can block on the
associated condition variable without missing either a change to the condition or a
signal that the condition has changed. Condition variables must be initialized by
calling cv_init(), and must be deallocated by calling cv_destroy().

The usual use of condition variables is to check a condition (for example, device state,
data structure reference count, etc.) while holding a mutex which keeps other threads
from changing the condition. If the condition is such that the thread should block,
cv_wait() is called with a related condition variable and the mutex. At some later
point in time, another thread would acquire the mutex, set the condition such that the
previous thread can be unblocked, unblock the previous thread with cv_signal() or
cv_broadcast(), and then release the mutex.

cv_wait() suspends the calling thread and exits the mutex atomically so that
another thread which holds the mutex cannot signal on the condition variable until
the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal() signals the condition and wakes one blocked thread. All blocked
threads can be unblocked by calling cv_broadcast(). You must acquire the mutex
passed into cv_wait() before calling cv_signal() or cv_broadcast().

The function cv_waitSig() is similar to cv_wait() but returns 0 if a signal (for
example, by kill(2)) is sent to the thread. In any case, the mutex is reacquired before
returning.

The function cv_timedwait() is similar to cv_wait(), except that it returns −1
without the condition being signaled after the timeout time has been reached.

The function cv_timedwaitSig() is similar to cv_timedwait() and
cv_waitSig(), except that it returns −1 without the condition being signaled after
the timeout time has been reached, or 0 if a signal (for example, by kill(2)) is sent to
the thread.

For both cv_timedwait() and cv_timedwaitSig(), time is in absolute clock
ticks since the last system reboot. The current time may be found by calling
ddi_get_lbolt(9F).

RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>For cv_waitSig() and cv_timedwaitSig() indicates that the condition was not necessarily signaled and the function returned because a signal (as in kill(2)) was pending.</td>
</tr>
<tr>
<td>−1</td>
<td>For cv_timedwait() and cv_timedwaitSig() indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.</td>
</tr>
<tr>
<td>&gt;0</td>
<td>For cv_waitSig(), cv_timedwait() or cv_timedwaitSig() indicates that the condition was met and the function returned due to a call to cv_signal() or cv_broadcast(), or due to a premature wakeup (see NOTES).</td>
</tr>
</tbody>
</table>

CONTEXT

These functions can be called from user, kernel or interrupt context. In most cases,
however, cv_wait(), cv_timedwait(), cv_waitSig(), and
cv_timedwaitSig() should not be called from interrupt context, and cannot be
called from a high-level interrupt context.

If cv_wait(), cv_timedwait(), cv_waitSig(), or cv_timedwaitSig() are
used from interrupt context, lower-priority interrupts will not be serviced during the
wait. This means that if the thread that will eventually perform the wakeup becomes
blocked on anything that requires the lower-priority interrupt, the system will hang.
For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**EXAMPLE 1** Waiting for a Flag Value in a Driver’s Unit

Here the condition being waited for is a flag value in a driver’s unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```c
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```c
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**NOTES** It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, and `cv_timedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

**SEE ALSO** `kill(2)`, `ddi_get_lbolt(9F)`, `mutex(9F)`, `mutex_init(9F)`

*Writing Device Drivers*
datamsg

NAME  datamsg – test whether a message is a data message

SYNOPSIS  
#include <sys/stream.h>  
#include <sys/ddi.h>  

int datamsg(unsigned char type);

INTERFACE LEVEL
PARAMETERS  
type  The type of message to be tested. The db_type field of the datab structure contains the message type. This field may be accessed through the message block using mp->b_datap->db_type.

DESCRIPTION  datamsg() tests the type of message to determine if it is a data message type (M_DATA, M_DELAY, M_PROTO, or M_PCPROTO).

RETURN VALUES  datamsg returns
1  if the message is a data message
0  otherwise.

CONTEXT  datamsg() can be called from user or interrupt context.

EXAMPLES  
EXAMPLE 1  The put(9E) routine enqueues all data messages for handling by the srv(9E) (service) routine. All non-data messages are handled in the put(9E) routine.

1  xxxput(q, mp)
2  queue_t *q;
3  mblk_t *mp;
4  {
5    if (datamsg(mp->b_datap->db_type)) {
6      putq(q, mp);
7      return;
8    }
9    switch (mp->b_datap->db_type) {
10      case M_FLUSH:
11      ...  
12    }

SEE ALSO  put(9E), srv(9E), allocb(9F), datab(9S), msgb(9S)

Writing Device Drivers

STREAMS Programming Guide
ddi_add_intr(9F)

NAME
ddi_add_intr, ddi_get_iblock_cookie, ddi_remove_intr – hardware interrupt handling routines

SYNOPSIS
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_iblock_cookie(dev_info_t *dip, uint_t inumber, ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_intr(dev_info_t *dip, uint_t inumber, ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *device_cookiep, uint_t (*int_handler)(caddr_t), caddr_t int_handler_arg);

void ddi_remove_intr(dev_info_t *dip, uint_t inumber, ddi_iblock_cookie_t iblock_cookie);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
For ddi_get_iblock_cookie():
dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookiep Pointer to an interrupt block cookie.

For ddi_add_intr():
dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookiep Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
device_cookiep Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.
int_handler Pointer to interrupt handler.
int_handler_arg Argument for interrupt handler.

For ddi_remove_intr():
dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookie Block cookie which identifies the interrupt handler to be removed.

ddi_get_iblock_cookie(): ddi_get_iblock_cookie() retrieves the interrupt block cookie associated with a particular interrupt specification. This routine should be called before ddi_add_intr() to retrieve the interrupt block cookie needed to initialize locks.
mutex(9F), rwlock(9F)) used by the interrupt routine. The interrupt number inumber determines for which interrupt specification to retrieve the cookie. inumber is associated with information provided either by the device (see sbus(4)) or the hardware configuration file (see sysbus(4), isa(4), eisa(4), and driver.conf(4)). If only one interrupt is associated with the device, inumber should be 0.

On a successful return, *iblock_cookie contains information needed for initializing locks associated with the interrupt specification corresponding to inumber (see mutex_init(9F) and rw_init(9F)). The driver can then initialize locks acquired by the interrupt routine before calling ddi_add_intr() which prevents a possible race condition where the driver’s interrupt handler is called immediately after the driver has called ddi_add_intr() but before the driver has initialized the locks. This may happen when an interrupt for a different device occurs on the same interrupt level. If the interrupt routine acquires the lock before the lock has been initialized, undefined behavior may result.

ddi_add_intr() adds an interrupt handler to the system. The interrupt number inumber determines which interrupt the handler will be associated with. (Refer to ddi_get_iblock_cookie() above.)

On a successful return, iblock_cookie contains information used for initializing locks associated with this interrupt specification (see mutex_init(9F) and rw_init(9F)). Note that the interrupt block cookie is usually obtained using ddi_get_iblock_cookie() to avoid the race conditions described above (refer to ddi_get_iblock_cookie() above). For this reason, iblock_cookie is no longer useful and should be set to NULL.

On a successful return, idevice_cookie contains a pointer to a ddi_idevice_cookie_t structure (see ddi_idevice_cookie(9S)) containing information useful for some devices that have programmable interrupts. If idevice_cookie is set to NULL, no value is returned.

The routine intr_handler, with its argument int_handler_arg, is called upon receipt of the appropriate interrupt. The interrupt handler should return DDI_INTR_CLAIMED if the interrupt was claimed, DDI_INTR_UNCLAIMED otherwise.

If successful, ddi_add_intr() will return DDI_SUCCESS. DDI_INTR_NOTFOUND is returned on i86pc and sun4n architectures if the interrupt information cannot be found. If the interrupt information cannot be found on the sun4u architecture, either DDI_INTR_NOTFOUND or DDI_FAILURE can be returned.

ddi_remove_intr() removes an interrupt handler from the system. Unloadable drivers should call this routine during their detach(9E) routine to remove their interrupt handler from the system.
The device interrupt routine for this instance of the device will not execute after `ddi_remove_intr()` returns. `ddi_remove_intr()` may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to `ddi_remove_intr()` or deadlock may result.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See `sysbus(4)`, `isa(4)`, `eisa(4)`, and `sbus(4)` for details.

**RETURN VALUES**

`ddi_add_intr()` and `ddi_get_iblock_cookie()` return:

- **DDI_SUCCESS** On success.
- **DDI_INTR_NOTFOUND** On failure to find the interrupt.
- **DDI_FAILURE** On failure. `DDI_FAILURE` can also be returned on failure to find interrupt (`sun4u`).

**CONTEXT**

`ddi_add_intr()`, `ddi_remove_intr()`, and `ddi_get_iblock_cookie()` can be called from user or kernel context.

**SEE ALSO**

`driver.conf(4)`, `eisa(4)`, `isa(4)`, `sbus(4)`, `sysbus(4)`, `attach(9E)`, `detach(9E)`, `ddi_intr_hilevel(9F)`, `mutex(9F)`, `mutex_init(9F)`, `rw_init(9F)`, `rwlock(9F)`, `ddi_idevice_cookie(9S)`

**Writing Device Drivers**

**NOTES**

`ddi_get_iblock_cookie()` must not be called after the driver adds an interrupt handler for the interrupt specification corresponding to `inumber`.

All consumers of these interfaces, checking return codes, should verify `return_code != DDI_SUCCESS`. Checking for specific failure codes can result in inconsistent behaviors among platforms.

**BUGS**

The `idevice_cookiep` should really point to a data structure that is specific to the bus architecture that the device operates on. Currently the SBus and PCI buses are supported and a single data structure is used to describe both.
### NAME

ddi_add_softintr, ddi_get_soft_iblock_cookie, ddi_remove_softintr, ddi_trigger_softintr – software interrupt handling routines

### SYNOPSIS

```c
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t *dip, int preference, ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t *idp, ddi_iblock_cookie_t *iblock_cookie, ddi_idevice_cookie_t *idevice_cookie, uint_t (*int_handler)(caddr_t int_handler_arg), caddr_t int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);

void ddi_trigger_softintr(ddi_softintr_t id);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

### PARAMETERS

**ddi_get_soft_iblock_cookie()**

- `dip` Pointer to a dev_info structure.
- `preference` The type of soft interrupt to retrieve the cookie for.
- `iblock_cookiep` Pointer to a location to store the interrupt block cookie.

**ddi_add_softintr()**

- `dip` Pointer to dev_info structure.
- `preference` A hint value describing the type of soft interrupt to generate.
- `idp` Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.
- `iblock_cookie` Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
- `idevice_cookie` Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).
- `int_handler` Pointer to interrupt handler.
- `int_handler_arg` Argument for interrupt handler.

**ddi_remove_softintr()**

- `id` The identifier specifying which soft interrupt handler to remove.

**ddi_trigger_softintr()**

- `id` The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.
For `ddi_get_soft_iblock_cookie()`:

`ddi_get_soft_iblock_cookie()` retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before `ddi_add_softintr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the software interrupt routine. `preference` determines which type of soft interrupt to retrieve the cookie for. The possible values for `preference` are:

- **DDI_SOFTINT_LOW**
  - Low priority soft interrupt.
- **DDI_SOFTINT_MED**
  - Medium priority soft interrupt.
- **DDI_SOFTINT_HIGH**
  - High priority soft interrupt.

On a successful return, `iblock_cookiep` contains information needed for initializing locks associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize mutexes acquired by the interrupt routine before calling `ddi_add_softintr()` which prevents a possible race condition where the driver's soft interrupt handler is called immediately after the driver has called `ddi_add_softintr()` but before the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

For `ddi_add_softintr()`:

`ddi_add_softintr()` adds a soft interrupt to the system. The user specified hint `preference` identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for `preference` should be the same as that used in the corresponding call to `ddi_get_soft_iblock_cookie()`. Refer to the description of `ddi_get_soft_iblock_cookie()` above.

The value returned in the location pointed at by `idp` is the soft interrupt identifier. This value is used in later calls to `ddi_remove_softintr()` and `ddi_trigger_softintr()` to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by `iblock_cookiep` is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). Note that the interrupt block cookie is normally obtained using `ddi_get_soft_iblock_cookie()` to avoid the race conditions described above (refer to the description of `ddi_get_soft_iblock_cookie()` above). For this reason, `iblock_cookiep` is no longer useful and should be set to `NULL`.

`idevice_cookiep` is not used and should be set to `NULL`. 
The routine `int_handler`, with its argument `int_handler_arg`, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine `int_handler` checks this flag, reachable through `int_handler_arg`, to determine if it should claim the interrupt and do its work.

The interrupt handler must return `DDI_INTR_CLAIMED` if the interrupt was claimed, `DDI_INTR_UNCLAIMED` otherwise.

If successful, `ddi_add_softintr()` will return `DDI_SUCCESS`; if the interrupt information cannot be found, it will return `DDI_FAILURE`.

For `ddi_remove_softintr()`:

`ddi_remove_softintr()` removes a soft interrupt from the system. The soft interrupt identifier `id`, which was returned from a call to `ddi_add_softintr()`, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For `ddi_trigger_softintr()`:

`ddi_trigger_softintr()` triggers a soft interrupt. The soft interrupt identifier `id` is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using `ddi_add_softintr()`.

RETURN VALUES:

`ddi_add_softintr()` and `ddi_get_soft_iblock_cookie()` return:

- `DDI_SUCCESS` on success
- `DDI_FAILURE` on failure

CONTEXT:

These functions can be called from user or kernel context. `ddi_trigger_softintr()` may be called from high-level interrupt context as well.

EXAMPLES:

EXAMPLE 1 device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See `ddi_intr_hilevel(9F)`. 
EXAMPLE 1 device using high-level interrupts  (Continued)

In the example, the high-level interrupt routine minimally services the device, and
enqueues the data for later processing by the soft interrupt handler. If the soft
interrupt handler is not currently running, the high-level interrupt routine triggers a
soft interrupt so the soft interrupt handler can process the data. Once running, the soft
interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data
shared between the high-level interrupt handler and the soft interrupt handler. The
low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```c
struct xxstate {
    ddi_softintr_t id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t low_mutex;
    int softint_running;
};
```

EXAMPLE 2 sample attach() routine

The following code fragment would usually appear in the driver’s `attach()` routine. `ddi_add_intr()` is used to add the high-level interrupt handler and
`ddi_add_softintr()` is used to add the low-level interrupt routine.

```c
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    ...
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber,
                        &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
               (void *)&xsp->high_iblock_cookie);

    /* add high-level routine - xxhighintr() */
    if (ddi_add_intr(dip, inumber, NULL, NULL,
                     xxhighintr, (caddr_t)xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }
    ...
}
```
EXAMPLE 2 Sample attach() routine

(Continued)

```c
} /* get soft iblock cookie */
if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
    &xsp->low_iblock_cookie) != DDI_SUCCESS) {
    /* clean up */
    return (DDI_FAILURE); /* fail attach */
}
/* initialize low-level mutex */
mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
    (void *)&xsp->low_iblock_cookie);
/* add low level routine - xxsoftintr() */
if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
    NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
    /* cleanup */
    return (DDI_FAILURE); /* fail attach */
}
```

EXAMPLE 3 High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, ddi_trigger_softintr() is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```c
static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    ...
    mutex_enter(&xsp->high_mutex);
    /*
    * Verify this device generated the interrupt
    * and disable the device interrupt.
    * Enqueue data for xxsoftintr() processing.
    */
    /* is xxsoftintr() already running ? */
    if (xsp->softint_running)
        need_softint = 0;
    else
        need_softint = 1;
    mutex_exit(&xsp->high_mutex);
    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
```
EXAMPLE 3 High-level interrupt routine  
(Continued)

    ddi_trigger_softintr(xsp->id);
    ...
    return (DDI_INTR_CLAIMED);
}

static uint_t
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    ...
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);
    /* verify there is work to do */
    if (work queue empty || xsp->softint_running ) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }
    xsp->softint_running = 1;
    while ( data on queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));
        /* de-queue data */
        mutex_exit(&xsp->high_mutex);
        /* Process data on queue */
        mutex_enter(&xsp->high_mutex);
    }
    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);
    return (DDI_INTR_CLAIMED);
}

SEE ALSO  ddi_add_intr(9F), ddi_in_panic(9F), ddi_intr_hilevel(9F),
          ddi_remove_intr(9F), mutex_init(9F)

Writing Device Drivers

NOTES  ddi_add_softintr() may not be used to add the same software interrupt handler
more than once. This is true even if a different value is used for int_handler_arg in each
of the calls to ddi_add_softintr(). Instead, the argument passed to the interrupt
handler should indicate what service(s) the interrupt handler should perform. For
example, the argument could be a pointer to the device’s soft state structure, which could contain a `which_service` field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_trigger_softintr()`.
ddi_binding_name(9F)

NAME

ddi_binding_name, ddi_get_name – return driver binding name

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

char *ddi_binding_name(dev_info_t *dip);
char *ddi_get_name(dev_info_t *dip);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dip A pointer to the device’s dev_info structure.

DESCRIPTION

ddi_binding_name() and ddi_get_name() return the driver binding name. This is the name used to select a driver for the device. This name is typically derived from the device name property or the device compatible property. The name returned may be a driver alias or the driver name.

RETURN VALUES

ddi_binding_name() and ddi_get_name() return the name used to bind a driver to a device.

CONTEXT

ddi_binding_name() and ddi_get_name() can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_node_name(9F)

Writing Device Drivers

WARNINGS

The name returned by ddi_binding_name() and ddi_get_name() is read-only.
ddi_btop(9F)

NAME  ddi_btop, ddi_btopr, ddi_ptob – page size conversions

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned long ddi_btop(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_btopr(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_ptob(dev_info_t *dip, unsigned long pages);

INTERFACE  Solaris DDI specific (Solaris DDI).

DESCRIPTION  This set of routines use the parent nexus driver to perform conversions in page size units.

  ddi_btop() converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.

  ddi_btopr() converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.

  ddi_ptob() converts the given number of pages to the number of bytes that it corresponds to.

Because bus nexus may possess their own hardware address translation facilities, these routines should be used in preference to the corresponding DDI/DKI routines btop(9F), btopr(9F), and ptob(9F), which only deal in terms of the pagesize of the main system MMU.

RETURN VALUES  ddi_btop() and ddi_btopr() return the number of corresponding pages.
  ddi_ptob() returns the corresponding number of bytes. There are no error return values.

CONTEXT  This function can be called from user or interrupt context.

EXAMPLES  EXAMPLE 1 Find the size (in bytes) of one page
  pagesize = ddi_ptob(dip, 1L);

SEE ALSO  btop(9F), btopr(9F), ptob(9F)

Writing Device Drivers
ddi_btop, ddi_btopr, ddi_ptob – page size conversions

#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned long ddi_btop(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_btopr(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_ptob(dev_info_t *dip, unsigned long pages);

Solaris DDI specific (Solaris DDI).

This set of routines use the parent nexus driver to perform conversions in page size units.

ddi_btop() converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.

ddi_btopr() converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.

ddi_ptob() converts the given number of pages to the number of bytes that it corresponds to.

Because bus nexus may possess their own hardware address translation facilities, these routines should be used in preference to the corresponding DDI/DKI routines btop(9F), btopr(9F), and ptob(9F), which only deal in terms of the pagesize of the main system MMU.

ddi_btop() and ddi_btopr() return the number of corresponding pages.

ddi_ptob() returns the corresponding number of bytes. There are no error return values.

This function can be called from user or interrupt context.

EXAMPLE 1 Find the size (in bytes) of one page

```
pagesize = ddi_ptob(dip, 1L);
```

SEE ALSO btop(9F), btopr(9F), ptob(9F)

Writing Device Drivers
ddi_check_acc_handle(9F)

NAME
ddi_check_acc_handle, ddi_check_dma_handle – Check data access and DMA handles

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_check_acc_handle(ddi_acc_handle_t acc_handle);
int ddi_check_dma_handle(ddi_dma_handle_t dma_handle);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL
PARAMETERS
acc_handle Data access handle obtained from a previous call to
ddi_regs_map_setup(9F), ddi_dma_mem_alloc(9F), or similar
function.
dma_handle DMA handle obtained from a previous call to
ddi_dma_setup(9F) or one of its derivatives.

DESCRIPTION
The ddi_check_acc_handle() and ddi_check_dma_handle() functions check
for faults that can interfere with communication between a driver and the device it
controls. Each function checks a single handle of a specific type and returns a status
value indicating whether faults affecting the resource mapped by the supplied handle
have been detected.

If a fault is indicated when checking a data access handle, this implies that the driver
is no longer able to access the mapped registers or memory using programmed I/O
through that handle. Typically, this might occur after the device has failed to respond
to an I/O access (for example, has incurred a bus error or timed out). The effect of
programmed I/O accesses made after this happens is undefined; for example, read
accesses (for example, ddi_get8(9F)) may return random values, and write accesses
(for example, ddi_put8(9F)) may or may not have any effect. This type of fault is
normally fatal to the operation of the device, and the driver should report it via
ddi_dev_report_fault(9F) specifying DDI_SERVICE_LOST for the impact, and
DDI_DATAPATH_FAULT for the location.

If a fault is indicated when checking a DMA handle, it implies that a fault has been
detected that has (or will) affect DMA transactions between the device and the
memory currently bound to the handle (or most recently bound, if the handle is
currently unbound). Possible causes include the failure of a component in the DMA
data path, or an attempt by the device to make an invalid DMA access. The driver
may be able to continue by falling back to a non-DMA mode of operation, but in
general, DMA faults are non-recoverable. The contents of the memory currently (or
previously) bound to the handle should be regarded as indeterminate. The fault
indication associated with the current transaction is lost once the handle is (re-)bound,
but because the fault may persist, future DMA operations may not succeed.

Note – Some implementations cannot detect all types of failure. If a fault is not
indicated, this does not constitute a guarantee that communication is possible.
However, if a check fails, this is a positive indication that a problem does exist with
respect to communication using that handle.
The `ddi_check_acc_handle()` and `ddi_check_dma_handle()` functions return `DDI_SUCCESS` if no faults affecting the supplied handle are detected and `DDI_FAILURE` if any fault affecting the supplied handle is detected.

```c
static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /* This driver uses only a single register-access handle */
    status = ddi_regs_map_setup(dip, REGSET_ZERO, &regaddr,
        0, 0, &acc_attrs, &acc_hdl);
    if (status != DDI_SUCCESS)
        return (DDI_FAILURE);
}

static int xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    if (ddi_check_acc_handle(acc_hdl) != DDI_SUCCESS) {
        ddi_dev_report_fault(dip, DDI_SERVICE_LOST,
            DDI_DATAPATH_FAULT, "register access fault during read");
        return (EIO);
    }
}
```

The `ddi_check_acc_handle()` and `ddi_check_dma_handle()` functions may be called from user, kernel, or interrupt context.

**SEE ALSO**
- `ddi_regs_map_setup(9F)`, `ddi_dma_setup(9F)`, `ddi_dev_report_fault(9F)`, `ddi_get8(9F)`, `ddi_put8(9F)`
ddi_check_dma_handle(9F)

NAME  

ddi_check_acc_handle, ddi_check_dma_handle – Check data access and DMA handles

SYNOPSIS  

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_check_acc_handle(ddi_acc_handle_t acc_handle);
int ddi_check_dma_handle(ddi_dma_handle_t dma_handle);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

PARAMETERS

acc_handle  
Data access handle obtained from a previous call to
  ddi_regs_map_setup(9F), ddi_dma_mem_alloc(9F), or similar
  function.

dma_handle  
DMA handle obtained from a previous call to
  ddi_dma_setup(9F) or one of its derivatives.

DESCRIPTION

The ddi_check_acc_handle() and ddi_check_dma_handle() functions check for faults that can interfere with communication between a driver and the device it controls. Each function checks a single handle of a specific type and returns a status value indicating whether faults affecting the resource mapped by the supplied handle have been detected.

If a fault is indicated when checking a data access handle, this implies that the driver is no longer able to access the mapped registers or memory using programmed I/O through that handle. Typically, this might occur after the device has failed to respond to an I/O access (for example, has incurred a bus error or timed out). The effect of programmed I/O accesses made after this happens is undefined; for example, read accesses (for example, ddi_get8(9F)) may return random values, and write accesses (for example, ddi_put8(9F)) may or may not have any effect. This type of fault is normally fatal to the operation of the device, and the driver should report it via ddi_dev_report_fault(9F) specifying DDI_SERVICE_LOST for the impact, and DDI_DATAPATH_FAULT for the location.

If a fault is indicated when checking a DMA handle, it implies that a fault has been detected that has (or will) affect DMA transactions between the device and the memory currently bound to the handle (or most recently bound, if the handle is currently unbound). Possible causes include the failure of a component in the DMA data path, or an attempt by the device to make an invalid DMA access. The driver may be able to continue by falling back to a non-DMA mode of operation, but in general, DMA faults are non-recoverable. The contents of the memory currently (or previously) bound to the handle should be regarded as indeterminate. The fault indication associated with the current transaction is lost once the handle is (re-)bound, but because the fault may persist, future DMA operations may not succeed.

Note – Some implementations cannot detect all types of failure. If a fault is not indicated, this does not constitute a guarantee that communication is possible. However, if a check fails, this is a positive indication that a problem does exist with respect to communication using that handle.
The ddi_check_acc_handle() and ddi_check_dma_handle() functions return DDI_SUCCESS if no faults affecting the supplied handle are detected and DDI_FAILURE if any fault affecting the supplied handle is detected.

**RETURN VALUES**

The ddi_check_acc_handle() and ddi_check_dma_handle() functions may be called from user, kernel, or interrupt context.

**EXAMPLES**

```c
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /* This driver uses only a single register-access handle */
    status = ddi_regs_map_setup(dip, REGSET_ZERO, &regaddr,
                                0, 0, &acc_attrs, &acc_hdl);
    if (status != DDI_SUCCESS)
        return (DDI_FAILURE);
    \
static int
xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    if (ddi_check_acc_handle(acc_hdl) != DDI_SUCCESS) {
        ddi_dev_report_fault(dip, DDI_SERVICE_LOST,
                              DDI_DATAPATH_FAULT, "register access fault during read");
        return (EIO);
    }
    \
\n\}
\n\}
\n\}
\n\}
```

**KERNEL FUNCTIONS FOR DRIVERS**

See also:

- ddi_regs_map_setup(9F)
- ddi_dma_setup(9F)
- ddi_dev_report_fault(9F)
- ddi_get8(9F)
- ddi_put8(9F)
ddi_copyin(9F)

NAME
ddi_copyin – copy data to a driver buffer

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_copyin(const void *buf, void *driverbuf, size_t cn, int flags);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
buf Source address from which data is transferred.

driverbuf Driver destination address to which data is transferred.

cn Number of bytes transferred.

flags Set of flag bits that provide address space information about buf.

DESCRIPTION
This routine is designed for use in driver ioctl(9E) routines for drivers that support layered iocts. ddi_copyin() copies data from a source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.

The flags argument determines the address space information about buf. If the FKIOCTL flag is set, this indicates that buf is a kernel address, and ddi_copyin() behaves like bcopy(9F). Otherwise, buf is interpreted as a user buffer address, and ddi_copyin() behaves like copyin(9F).

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move according to address alignment.

RETURN VALUES
ddi_copyin() returns 0, indicating a successful copy. It returns −1 if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory.
  For example, an uncorrectable parity or ECC error occurred.

If −1 is returned to the caller, driver entry point routines should return EFAULT.

CONTEXT
ddi_copyin() can be called from user or kernel context only.
A driver ioctl(9E) routine (line 12) can be used to get or set device attributes or registers. For the XX_SETREGS condition (line 25), the driver copies the user data in *arg to the device registers. If the specified argument contains an invalid address, an error code is returned.

```c
1 struct device { /* layout of physical device registers */
2     int     control; /* physical device control word */
3     int     status; /* physical device status word */
4     short   recv_char; /* receive character from device */
5     short   xmit_char; /* transmit character to device */
6 };
7 struct device_state {
8     volatile struct device *regsp; /* pointer to device registers */
9     kmutex_t reg_mutex; /* protect device registers */
   ...
10 };
11 static void *statep; /* for soft state routines */
12 xxioctl(dev_t dev, int cmd, int arg, int mode,
13          cred_t *cred_p, int *rval_p)
14 {
15     struct device_state *sp;
16     volatile struct device *rp;
17     struct device reg_buf; /* temporary buffer for registers */
18     int instance;
19     instance = getminor(dev);
20     sp = ddi_get_soft_state(statep, instance);
21     if (sp == NULL)
22         return (ENXIO);
23     rp = sp->regsp;
   ...
24     switch (cmd) {
25         case XX_GETREGS: /* copy data to temp. regs. buf */
26             if (ddi_copyin(arg, &reg_buf,
27                    sizeof (struct device), mode) != 0) {
28                 return (EFAULT);
29             }
30             mutex_enter(&sp->reg_mutex);
31             /*
32                 * Copy data from temporary device register
33                 * buffer to device registers.
34                 * e.g. rp->control = reg_buf.control;
35             */
36             mutex_exit(&sp->reg_mutex);
37             break;
38         }
39     }
```

**EXAMPLE 1 ddi_copyin() example**

...
### ddi_copyin(9F)

**SEE ALSO**
- ioctl(9E), bcopy(9F), copyin(9F), copyout(9F), ddi_copyout(9F), uimorev(9F)

**NOTES**
- The value of the `flags` argument to `ddi_copyin()` should be passed through directly from the `mode` argument of `ioctl()` untranslated.
- Driver defined locks should not be held across calls to this function.
- `ddi_copyin()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*. 

---

**EXAMPLE 1**

```c
/* ddi_example() example (Continued) */

int

```
### NAME
ddi_copyout – copy data from a driver

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_copyout(const void *driverbuf, void *buf, size_t cn, int flags);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **driverbuf**: Source address in the driver from which the data is transferred.
- **buf**: Destination address to which the data is transferred.
- **cn**: Number of bytes to copy.
- **flags**: Set of flag bits that provide address space information about `buf`.

### DESCRIPTION
This routine is designed for use in driver ioctl(9E) routines for drivers that support layered ioctls. ddi_copyout() copies data from a driver buffer to a destination address, `buf`.

The `flags` argument determines the address space information about `buf`. If the FKIOCTL flag is set, this indicates that `buf` is a kernel address, and `ddi_copyout()` behaves like `bcopy(9F)`. Otherwise, `buf` is interpreted as a user buffer address, and `ddi_copyout()` behaves like `copyout(9F).

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

### RETURN VALUES
Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, −1 is returned if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.

If −1 is returned to the caller, driver entry point routines should return EFAULT.

### CONTEXT
`ddi_copyout()` can be called from user or kernel context only.
ddi_copyout(9F)

EXAMPLE 1 ddi_copyout() example

A driver ioctl(9E) routine (line 12) can be used to get or set device attributes or registers. In the XX_GETREGS condition (line 25), the driver copies the current device register values to another data area. If the specified argument contains an invalid address, an error code is returned.

```c
struct device { /* layout of physical device registers */
    int control; /* physical device control word */
    int status; /* physical device status word */
    short recv_char; /* receive character from device */
    short xmit_char; /* transmit character to device */
};

struct device_state {
    volatile struct device *regsp; /* pointer to device registers */
    kmutex_t reg_mutex; /* protect device registers */
    ...
};

static void *statep; /* for soft state routines */

xxioctl(dev_t dev, int cmd, int arg, int mode,
    cred_t *cred_p, int *rval_p)
{
    struct device_state *sp;
    volatile struct device *rp;
    struct device reg_buf; /* temporary buffer for registers */
    int instance;

    instance = getminor(dev);
    sp = ddi_get_soft_state(statep, instance);
    if (sp == NULL)
        return (ENXIO);
    rp = sp->regsp;
    ...

    switch (cmd) {
    case XX_GETREGS: /* copy registers to arg */
        mutex_enter(&sp->reg_mutex);
        /*
         * Copy data from device registers to
         * temporary device register buffer
         * e.g. reg_buf.control = rp->control;
         */
        mutex_exit(&sp->reg_mutex);
        if (ddi_copyout(&reg_buf, arg,
            sizeof (struct device), mode) != 0) {
            return (EFAULT);
        }
        break;
    }
}
```

man pages section 9: DDI and DKI Kernel Functions • Last Revised 19 Apr 2000
EXAMPLE 1 ddi_copyout() example (Continued)

SEE ALSO

ioctl(9E), bcopy(9F), copyin(9F), copyout(9F), ddi_copyin(9F), uiomove(9F)

Writing Device Drivers

NOTES

The value of the flags argument to ddi_copyout() should be passed through directly from the mode argument of ioctl() untranslated.

Driver defined locks should not be held across calls to this function.

ddi_copyout() should not be used from a streams driver. See M_COPYIN and M_COPYOUT in STREAMS Programming Guide.
ddi_create_minor_node(9F)

NAME

ddi_create_minor_node – create a minor node for this device

SYNOPSIS

#include <sys/stat.h>
#include <sys/sunddi.h>

int ddi_create_minor_node(dev_info_t *dip, char *name, int spec_type, 
minor_t minor_num, char *node_type, int flag);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

dip A pointer to the device’s dev_info structure.

name The name of this particular minor device.

spec_type S_IFCHR or S_IFBLK for character or block minor devices respectively.

minor_num The minor number for this particular minor device.

node_type Any string that uniquely identifies the type of node. The following predefined node types are provided with this release:

DDI_NT_SERIAL For serial ports
DDI_NT_SERIAL_MB For on board serial ports
DDI_NT_SERIAL_DO For dial out ports
DDI_NT_SERIAL_MB_DO For on board dial out ports
DDI_NT_BLOCK For hard disks
DDI_NT_BLOCK_CHAN For hard disks with channel or target numbers
DDI_NT_CD For CDROM drives
DDI_NT_CD_CHAN For CDROM drives with channel or target numbers
DDI_NT_PD For floppy disks
DDI_NT_TAPE For tape drives
DDI_NT_NET For DLPI style 1 or style 2 network devices
DDI_NT_DISPLAY For display devices
DDI_PSEUDO For pseudo devices

flag If the device is a clone device then this flag is set to CLONE_DEV else it is set to 0. The device node class can also be specified using this flag. The device classes do not have an effect in the creation of the device node in a non-clustered environment; but for device drivers intended for use in a clustered environment, one of the following needs to be specified. If the device class is not indicated

322 man pages section 9: DDI and DKI Kernel Functions • Last Revised 7 Jun 2001
the default class for pseudo devices will be NODESPECIFIC_DEV and for physical devices will be ENUMERATED_DEV.

GLOBAL_DEV The device is a node invariant device and can be opened from any node in the cluster.

NODEBOUND_DEV The device is node invariant but it has cluster wide state associated with it so that all subsequent opens must be directed there.

NODESPECIFIC_DEV The device node provides node specific information and must be opened co-located with the process.

ENUMERATED_DEV Unique cluster wide device nodes. The i/o must take place at the host where the device node was created.

ddi_create_minor_node() provides the necessary information to enable the system to create the /dev and /devices hierarchies. The name is used to create the minor name of the block or character special file under the /devices hierarchy. At-sign (@), slash (/), and space are not allowed. The spec_type specifies whether this is a block or character device. The minor_num is the minor number for the device. The node_type is used to create the names in the /dev hierarchy that refers to the names in the /devices hierarchy. See disks(1M), ports(1M), tapes(1M), devlinks(1M). Finally flag determines if this is a clone device or not, and what device class the node belongs to.

ddi_create_minor_node() returns:
- DDI_SUCCESS Was able to allocate memory, create the minor data structure, and place it into the linked list of minor devices for this driver.
- DDI_FAILURE Minor node creation failed.

EXAMPLE 1 Create Data Structure Describing Minor Device with Minor Number of 0

The following example creates a data structure describing a minor device called foo which has a minor number of 0. It is of type DDI_NT_BLOCK (a block device) and it is not a clone device.

```c
ddi_create_minor_node(dip, "foo", S_IFBLK, 0, DDI_NT_BLOCK, 0);
```

SEE ALSO add_dr(1M), devlinks(1M), disks(1M), drvconfig(1M), ports(1M), tapes(1M), attach(9E), ddi_remove_minor_node(9F)

Writing Device Drivers
If the driver is for a network device (`node_type DDI_NT_NET`), note that the driver name will undergo the driver name constraints identified in the NOTES section of `dlpi(7P)`. Additionally, the minor name must match the driver name for a DLPI style 2 provider. If the driver is a DLPI style 1 provider, the minor name must also match the driver name with the exception that the ppa is appended to the minor name.
ddi_device_copy(9F)

NAME

ddi_device_copy – copy data from one device register to another device register

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_device_copy(ddi_acc_handle_t src_handle, caddr_t src_addr,
                     ssize_t src_advcnt, ddi_acc_handle_t dest_handle, caddr_t dest_addr,
                     ssize_t dest_advcnt, size_t bytecount, uint_t dev_datasz);

DESCRIPTION

ddi_device_copy() copies bytecount bytes from the source address, src_addr, to the destination address, dest_addr. The attributes encoded in the access handles, src_handle and dest_handle, govern how data is actually copied from the source to the destination. Only matching data sizes between the source and destination are supported.

Data will automatically be translated to maintain a consistent view between the source and the destination. The translation may involve byte-swapping if the source and the destination devices have incompatible endian characteristics.

The src_advcnt and dest_advcnt arguments specifies the number of dev_datasz units to advance with each access to the device addresses. A value of 0 will use the same source and destination device address on every access. A positive value increments the corresponding device address by certain number of data size units in the next access. On the other hand, a negative value decrements the device address.

The dev_datasz argument determines the size of the data word on each access. The data size must be the same between the source and destination.

RETURN VALUES

ddi_device_copy() returns:

DDI_SUCCESS Successfully transferred the data.

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

src_handle The data access handle of the source device.
src_addr Base data source address.
src_advcnt Number of dev_datasz units to advance on every access.
dest_handle The data access handle of the destination device.
dest_addr Base data destination address.
dest_advcnt Number of dev_datasz units to advance on every access.
bytecount Number of bytes to transfer.
dev_datasz The size of each data word. Possible values are defined as:

- DDI_DATA_SZ01_ACC 1 byte data size
- DDI_DATA_SZ02_ACC 2 bytes data size
- DDI_DATA_SZ04_ACC 4 bytes data size
- DDI_DATA_SZ08_ACC 8 bytes data size

Kernel Functions for Drivers 325
<table>
<thead>
<tr>
<th>DDI FAILURE</th>
<th>The byte count is not a multiple <em>dev_datasz</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT</td>
<td><code>ddi_device_copy()</code> can be called from user, kernel, or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>ddi_regs_map_free(9F), ddi_regs_map_setup(9F)</code></td>
</tr>
</tbody>
</table>

*Writing Device Drivers*
ddi_device_zero(9F)

NAME

ddi_device_zero — zero fill the device

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_device_zero(ddi_acc_handle_t handle, caddr_t dev_addr,
size_t bytecount, ssize_t dev_advcnt, uint_t dev_datasz);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</td>
</tr>
<tr>
<td>dev_addr</td>
<td>Beginning of the device address.</td>
</tr>
<tr>
<td>bytecount</td>
<td>Number of bytes to zero.</td>
</tr>
<tr>
<td>dev_advcnt</td>
<td>Number of dev_datasz units to advance on every access.</td>
</tr>
<tr>
<td>dev_datasz</td>
<td>The size of each data word. Possible values are defined as:</td>
</tr>
<tr>
<td></td>
<td>DDI_DATA_SZ01_ACC 1 byte data size</td>
</tr>
<tr>
<td></td>
<td>DDI_DATA_SZ02_ACC 2 bytes data size</td>
</tr>
<tr>
<td></td>
<td>DDI_DATA_SZ04_ACC 4 bytes data size</td>
</tr>
<tr>
<td></td>
<td>DDI_DATA_SZ08_ACC 8 bytes data size</td>
</tr>
</tbody>
</table>

DESCRIPTION

ddi_device_zero() function fills the given, bytecount, number of byte of zeroes to the device register or memory.

The dev_advcnt argument determines the value of the device address, dev_addr, on each access. A value of 0 will use the same device address, dev_addr, on every access. A positive value increments the device address in the next access while a negative value decrements the address. The device address is incremented and decremented in dev_datasz units.

The dev_datasz argument determines the size of data word on each access.

RETURN VALUES

ddi_device_zero() returns:

DDI_SUCCESS Successfully zeroed the data.

DDI_FAILURE The byte count is not a multiple of dev_datasz.

CONTEXT

ddi_device_zero() can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_regs_map_free(9F), ddi_regs_map_setup(9F)

Writing Device Drivers
The device id address.

devidstr The devid and minor_name represented as a string.
devid1 The first of two device id addresses to be compared calling
ddi_devid_compare().
devid2 The second of two device id addresses to be compared calling
ddi_devid_compare().
dip A dev_info pointer, which identifies the device.
devid_type The following device id types may be accepted by the
ddi_devid_init() function:

DEVID_SCSI3_WWN World Wide Name associated with
SCSI-3 devices.

DEVID_SCSI_SERIAL Vendor ID and serial number
associated with a SCSI device.
Note: This may only be used if
known to be unique; otherwise a
fabricated device id must be used.

DEVID_ENCAP Device ID of another device. This is
for layered device driver usage.

DEVID_FAB Fabricated device ID.

minor_name The minor name to be encoded.
Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

- **ddi_devid_compare()**: compares two device IDs byte-by-byte and determines both equality and sort order.
- **ddi_devid_sizeof()**: returns the number of bytes allocated for the passed in device ID (`devid`).
- **ddi_devid_init()**: allocates memory and initializes the opaque device ID structure. This function does not store the `devid`. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the `devid` on some reliable store. When a `devid_type` of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (`id`) must be passed in (`nbytes`).

When the `devid_type` `DEVID_FAB` is used, the array of bytes (`id`) must be NULL and the length (`nbytes`) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

- **ddi_devid_free()**: frees the memory allocated for the returned `devid` by the `ddi_devid_init()` and `devid_str_decode()` functions.

- **ddi_devid_register()**: registers the device ID address (`devid`) with the DDI framework, associating it with the `dev_info` passed in (`dip`). The drivers must register device IDs at attach time. See `attach(9E)`.

- **ddi_devid_unregister()**: removes the device ID address from the `dev_info` framework, associating it with the `dev_info` passed in (`dip`). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_devid_free()` function. See `detach(9E)`.

- **ddi_devid_valid()**: validates the device ID (`devid`) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.
The `ddi_devid_str_encode()` function encodes a `devid` and `minor_name` into a null-terminated ASCII string, returning a pointer to that string. If both a `devid` and a `minor_name` are non-null, then a slash (/) is used to separate the `devid` from the `minor_name` in the encoded string. If `minor_name` is null, then only the `devid` is encoded. If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` or `ddi_devid_str_encode()` function and decodes the contained device ID and `minor_name`, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr` id0 was specified then the returned device ID and `minor_name` will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned `minor_name` must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- `DDI_SUCCESS` Valid device ID.
- `DDI_FAILURE` Invalid device ID.

`ddi_devid_register()` returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- `DDI_SUCCESS` Valid device ID.
- `DDI_FAILURE` Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1` The first device ID is less than the second device ID.
- `0` The first device ID is equal to the second device ID.
- `1` The first device ID is greater than the second device ID.
ddi_devid_sizeof() returns the size of the devid in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.

ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

DDI_SUCCESS
Success.

DDI_FAILURE
Failure; the devidstr string was not valid.

CONTEXT These functions can be called from a user or kernel context.

SEE ALSO ddevid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
NAME  ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS

    int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
    size_t ddi_devid_sizeof(ddi_devid_t devid);
    int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
        nbytes, void *id, ddi_devid_t *retdevid);
    void ddi_devid_free(ddi_devid_t devid);
    int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
    int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
        **retminor_name);
    int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
    int ddi_devid_str_free(char *devidstr);
    void ddi_devid_unregister(dev_info_t *dip);
    int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS

    devid     The device id address.
    devidstr  The devid and minor_name represented as a string.
    devid1    The first of two device id addresses to be compared calling
              ddi_devid_compare().
    devid2    The second of two device id addresses to be compared calling
              ddi_devid_compare().
    dip       A dev_info pointer, which identifies the device.
    devid_type The following device id types may be accepted by the
              ddi_devid_init() function:
              DEVID_SCSI3_WWN      World Wide Name associated with
                                 SCSI-3 devices.
              DEVID_SCSI_SERIAL    Vendor ID and serial number
                                 associated with a SCSI device.
                                 Note: This may only be used if
                                 known to be unique; otherwise a
                                 fabricated device id must be used.
              DEVID_ENCAP          Device ID of another device. This is
                                 for layered device driver usage.
              DEVID_FAB            Fabricated device ID.
    minor_name The minor name to be encoded.
    nbytes     The length in bytes of device ID.
Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

- `ddi_devid_compare()` compares two device IDs byte-by-byte and determines both equality and sort order.
- `ddi_devid_sizeof()` returns the number of bytes allocated for the passed in device ID (`devid`).
- `ddi_devid_init()` allocates memory and initializes the opaque device ID structure. This function does not store the `devid`. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the `devid` on some reliable store. When a `devid_type` of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (`id`) must be passed in (`nbytes`).

When the `devid_type` `DEVID_FAB` is used, the array of bytes (`id`) must be NULL and the length (`nbytes`) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

- `ddi_devid_free()` frees the memory allocated for the returned `devid` by the `ddi_devid_init()` and `devid_str_decode()` functions.

- `ddi_devid_register()` registers the device ID address (`devid`) with the DDI framework, associating it with the `dev_info` passed in (`dip`). The drivers must register device IDs at attach time. See `attach(9E)`.

- `ddi_devid_unregister()` removes the device ID address from the `dev_info` passed in (`dip`). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_devid_free()` function. See `detach(9E)`.

- `ddi_devid_valid()` validates the device ID (`devid`) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The `ddi_devid_str_encode()` function encodes a `devid` and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a `devid` and a `minor_name` are non-null, then a slash (`/`) is used to separate the `devid` from the `minor_name` in the encoded string. If `minor_name` is null, then only the `devid` is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr_id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- **DDI_SUCCESS**: Success.
- **DDI_FAILURE**: Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS**: Valid device ID.
- **DDI_FAILURE**: Invalid device ID.

`ddi_devid_register()` returns the following values:

- **DDI_SUCCESS**: Success.
- **DDI_FAILURE**: Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS**: Valid device ID.
- **DDI_FAILURE**: Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1`: The first device ID is less than the second device ID.
- `0`: The first device ID is equal to the second device ID.
- `1`: The first device ID is greater than the second device ID.

`ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

- **DDI_SUCCESS**
  - Success.
- **DDI_FAILURE**
  - Failure; the devidstr string was not valid.

**CONTEXT**

These functions can be called from a user or kernel context.

**SEE ALSO**

devid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

*Writing Device Drivers*
NAME

ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
 ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
 ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS

int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
 nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
 **retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

devid The device id address.
devidstr The devid and minor_name represented as a string.
devid1 The first of two device id addresses to be compared calling
ddi_devid_compare().
devid2 The second of two device id addresses to be compared calling
ddi_devid_compare().
dip A dev_info pointer, which identifies the device.
devid_type The following device id types may be accepted by the
 ddi_devid_init() function:

DEVID_SCSI3_WWN World Wide Name associated with
SCSI-3 devices.

DEVID_SCSI_SERIAL Vendor ID and serial number
associated with a SCSI device.
Note: This may only be used if
known to be unique; otherwise a
fabricated device id must be used.

DEVID_ENCAP Device ID of another device. This is
for layered device driver usage.

DEVID_FAB Fabricated device ID.

minor_name The minor name to be encoded.
nbytes The length in bytes of device ID.
The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

`ddi_devid_compare()` compares two device IDs byte-by-byte and determines both equality and sort order.

`ddi_devid_sizeof()` returns the number of bytes allocated for the passed in device ID (`devid`).

`ddi_devid_init()` allocates memory and initializes the opaque device ID structure. This function does not store the `devid`. If the device id is not derived from the device's firmware, it is the driver's responsibility to store the `devid` on some reliable store. When a `devid_type` of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (`id`) must be passed in (`nbytes`).

When the `devid_type` `DEVID_FAB` is used, the array of bytes (`id`) must be NULL and the length (`nbytes`) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine's host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

`ddi_devid_free()` frees the memory allocated for the returned `devid` by the `ddi_devid_init()` and `devid_str_decode()` functions.

`ddi_devid_register()` registers the device ID address (`devid`) with the DDI framework, associating it with the `dev_info` passed in (`dip`). The drivers must register device IDs at attach time. See `attach(9E)`.

`ddi_devid_unregister()` removes the device ID address from the `dev_info` passed in (`dip`). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_devid_free()` function. See `detach(9E)`.

`ddi_devid_valid()` validates the device ID (`devid`) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The `ddi_devid_str_encode()` function encodes a `devid` and `minor_name` into a null-terminated ASCII string, returning a pointer to that string. If both a `devid` and a `minor_name` are non-null, then a slash (/) is used to separate the `devid` from the `minor_name` in the encoded string. If `minor_name` is null, then only the `devid` is encoded.
If the devid is null, then the special string id0 is returned. Note that you cannot compare the returned string against another string with strcmp() to determine devid equality. The returned string must be freed by calling devid_str_free().

The ddi_devid_str_decode() function takes a string previously produced by the devid_str_encode(3DEVID) or ddi_devid_str_encode() function and decodes the contained device ID and minor_name, allocating and returning pointers to the extracted parts through the retdevid and retminor_name arguments. If the special devidstr id0 was specified then the returned device ID and minor name will both be null. A non-null returned devid must be freed by the caller through the ddi_devid_free() function. A non-null returned minor name must be freed by calling ddi_devid_str_free().

The ddi_devid_str_free() function is used to free all strings returned by the ddi_devid functions (the ddi_devid_str_encode() function return value and the returned retminor_name argument).

**RETURN VALUES**

**ddi_devid_init()** returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Out of memory. An invalid devid_type was passed in.

**ddi_devid_valid()** returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

**ddi_devid_register()** returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Failure. The device ID is already registered or the device ID is invalid.

**ddi_devid_valid()** returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

**ddi_devid_compare()** returns the following values:

- **-1** The first device ID is less than the second device ID.
- **0** The first device ID is equal to the second device ID.
- **1** The first device ID is greater than the second device ID.

**ddi_devid_sizeof()** returns the size of the devid in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_SUCCESS</td>
<td>Success.</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
<td>Failure; the devidstr string was not valid.</td>
</tr>
</tbody>
</table>

These functions can be called from a user or kernel context.

SEE ALSO devid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
NAME  ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
      ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
      ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS  int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
      nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid,
      char **retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS
  devid   The device id address.
  devidstr The devid and minor_name represented as a string.
  devid1   The first of two device id addresses to be compared calling
            ddi_devid_compare().
  devid2   The second of two device id addresses to be compared calling
            ddi_devid_compare().
  dip      A dev_info pointer, which identifies the device.
  devid_type The following device id types may be accepted by the
              ddi_devid_init() function:

  DEVID_SCSI3_WWN   World Wide Name associated with
                    SCSI-3 devices.

  DEVID_SCSI_SERIAL Vendor ID and serial number
                    associated with a SCSI device.
                    Note: This may only be used if
                    known to be unique; otherwise a
                    fabricated device id must be used.

  DEVID_ENCAP      Device ID of another device. This is
                    for layered device driver usage.

  DEVID_FAB        Fabricated device ID.

  minor_name     The minor name to be encoded.
  nbytes         The length in bytes of device ID.
ddi_devid_register(9F)

INTERFACE
LEVEL
DESCRIPTION

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

ddi_devid_compare() compares two device IDs byte-by-byte and determines both equality and sort order.

ddi_devid_sizeof() returns the number of bytes allocated for the passed in device ID (devid).

ddi_devid_init() allocates memory and initializes the opaque device ID structure. This function does not store the devid. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the devid on some reliable store. When a devid_type of either DEVID_SCSI3_WWN, DEVID_SCSI_SERIAL, or DEVID_ENCAP is accepted, an array of bytes (id) must be passed in (nbytes).

When the devid_type DEVID_FAB is used, the array of bytes (id) must be NULL and the length (nbytes) must be zero. The fabricated device ids, DEVID_FAB will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the ddi_devid_free() function.

ddi_devid_free() frees the memory allocated for the returned devid by the ddi_devid_init() and devid_str_decode() functions.

ddi_devid_register() registers the device ID address (devid) with the DDI framework, associating it with the dev_info passed in (dip). The drivers must register device IDs at attach time. See attach(9E).

ddi_devid_unregister() removes the device ID address from the dev_info passed in (dip). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the ddi_devid_free() function. See detach(9E).

ddi_devid_valid() validates the device ID (devid) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The ddi_devid_str_encode() function encodes a devid and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a devid and a minor_name are non-null, then a slash (/) is used to separate the devid from the minor_name in the encoded string. If minor_name is null, then only the devid is encoded.

retdevid The return address of the device ID.

retminor_name The return address of a minor name. Free string with ddi_devid_str_free().
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode(3DEVID)` or `ddi_devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- `DDI_SUCCESS` Valid device ID.
- `DDI_FAILURE` Invalid device ID.

`ddi_devid_register()` returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- `DDI_SUCCESS` Valid device ID.
- `DDI_FAILURE` Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1` The first device ID is less than the second device ID.
- `0` The first device ID is equal to the second device ID.
- `1` The first device ID is greater than the second device ID.

`ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
**ddi_devid_str_encode()** returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid `devid`. If the return value is non-null then the caller must free the returned string by using the `devid_str_free()` function.

**ddi_devid_str_decode()** returns the following values:

- **DDI_SUCCESS**
  - Success.
- **DDI_FAILURE**
  - Failure; the `devidstr` string was not valid.

**CONTEXT**
These functions can be called from a user or kernel context.

**SEE ALSO**
- `devid_get(3DEVID)`, `libdevid(3LIB)`, `attributes(5)`, `attach(9E)`, `detach(9E)`, `kmem_free(9F)`

*Writing Device Drivers*
NAME
ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS

int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
**retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS
devid The device id address.
devidstr The devid and minor_name represented as a string.
devid1 The first of two device id addresses to be compared calling
ddi_devid_compare().
devid2 The second of two device id addresses to be compared calling
ddi_devid_compare().
dip A dev_info pointer, which identifies the device.
devid_type The following device id types may be accepted by the
ddi_devid_init() function:

DEVID_SCSI3_WWN World Wide Name associated with
SCSI-3 devices.

DEVID_SCSI_SERIAL Vendor IDand serial number
associated with a SCSI device.
Note: This may only be used if
known to be unique; otherwise a
fabricated device id must be used.

DEVID_ENCAP Device ID of another device. This is
for layered device driver usage.

DEVID_FAB Fabricated device ID.

minor_name The minor name to be encoded.
nbytes The length in bytes of device ID.

344 man pages section 9: DDI and DKI Kernel Functions • Last Revised 9 Nov 2000
The return address of the device ID.

The return address of a minor name. Free string with ddi_devid_str_free().

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

\`ddi_devid_compare()\` compares two device IDs byte-by-byte and determines both equality and sort order.

\`ddi_devid_sizeof()\` returns the number of bytes allocated for the passed in device ID (\`devid\`).

\`ddi_devid_init()\` allocates memory and initializes the opaque device ID structure. This function does not store the \`devid\`. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the \`devid\` on some reliable store. When a \`devid_type\` of either \`DEVID_SCSI3_WWN\`, \`DEVID_SCSI_SERIAL\`, or \`DEVID_ENCAP\` is accepted, an array of bytes (\`id\`) must be passed in (\`nbytes\`).

When the \`devid_type\` \`DEVID_FAB\` is used, the array of bytes (\`id\`) must be NULL and the length (\`nbytes\`) must be zero. The fabricated device ids, \`DEVID_FAB\` will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the \`ddi_devid_free()\` function.

\`ddi_devid_free()\` frees the memory allocated for the returned \`devid\` by the \`ddi_devid_init()\` and \`devid_str_decode()\` functions.

\`ddi_devid_register()\` registers the device ID address (\`devid\`) with the DDI framework, associating it with the dev_info passed in (\`dip\`). The drivers must register device IDs at attach time. See attach(9E).

\`ddi_devid_unregister()\` removes the device ID address from the dev_info passed in (\`dip\`). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the \`ddi_devid_free()\` function. See detach(9E).

\`ddi_devid_valid()\` validates the device ID (\`devid\`) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The \`ddi_devid_str_encode()\` function encodes a \`devid\` and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a \`devid\` and a \`minor_name\` are non-null, then a slash (/) is used to separate the \`devid\` from the \`minor_name\` in the encoded string. If \`minor_name\` is null, then only the \`devid\` is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` or `ddi_devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `reminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `reminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- **DDI_SUCCESS**: Success.
- **DDI_FAILURE**: Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS**: Valid device ID.
- **DDI_FAILURE**: Invalid device ID.

`ddi_devid_register()` returns the following values:

- **DDI_SUCCESS**: Success.
- **DDI_FAILURE**: Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS**: Valid device ID.
- **DDI_FAILURE**: Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1`: The first device ID is less than the second device ID.
- `0`: The first device ID is equal to the second device ID.
- `1`: The first device ID is greater than the second device ID.

`ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

DDI_SUCCESS
Success.

DDI_FAILURE
Failure; the devidstr string was not valid.

CONTEXT These functions can be called from a user or kernel context.

SEE ALSO devid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
NAME
  ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
  ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
  ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS
  int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
  size_t ddi_devid_sizeof(ddi_devid_t devid);
  int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
    nbytes, void *id, ddi_devid_t *retdevid);
  void ddi_devid_free(ddi_devid_t devid);
  int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
  int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
    **retminor_name);
  int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
  int ddi_devid_str_free(char *devidstr);
  void ddi_devid_unregister(dev_info_t *dip);
  int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS
  devid
    The device id address.
  devidstr
    The devid and minor_name represented as a string.
  devid1
    The first of two device id addresses to be compared calling
      ddi_devid_compare().
  devid2
    The second of two device id addresses to be compared calling
      ddi_devid_compare().
  dip
    A dev_info pointer, which identifies the device.
  devid_type
    The following device id types may be accepted by the
      ddi_devid_init() function:
      DEVID_SCSI3_WWN     World Wide Name associated with
                           SCSI-3 devices.
      DEVID_SCSI_SERIAL   Vendor ID and serial number
                           associated with a SCSI device.
                           Note: This may only be used if
                           known to be unique; otherwise a
                           fabricated device id must be used.
      DEVID_ENCAP        Device ID of another device. This is
                           for layered device driver usage.
      DEVID_FAB          Fabricated device ID.
  minor_name
    The minor name to be encoded.
  nbytes
    The length in bytes of device ID.
The return address of the device ID.

The return address of a minor name. Free string with 
ddi_devid_str_free().

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

ddi_devid_compare() compares two device IDs byte-by-byte and determines both equality and sort order.

ddi_devid_sizeof() returns the number of bytes allocated for the passed in device ID (devid).

ddi_devid_init() allocates memory and initializes the opaque device ID structure. This function does not store the devid. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the devid on some reliable store. When a devid_type of either DEVID_SCSI3_WWN, DEVID_SCSI_SERIAL, or DEVID_ENCAP is accepted, an array of bytes (id) must be passed in (nbytes).

When the devid_type DEVID_FAB is used, the array of bytes (id) must be NULL and the length (nbytes) must be zero. The fabricated device ids, DEVID_FAB will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the 
ddi_devid_free() function.

ddi_devid_free() frees the memory allocated for the returned devid by the 
ddi_devid_init() and devid_str_decode() functions.

ddi_devid_register() registers the device ID address (devid) with the DDI framework, associating it with the dev_info passed in (dip). The drivers must register device IDs at attach time. See attach(9E).

ddi_devid_unregister() removes the device ID address from the dev_info passed in (dip). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the 
ddi_devid_free() function. See detach(9E).

ddi_devid_valid() validates the device ID (devid) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The ddi_devid_str_encode() function encodes a devid and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a devid and a 
minor_name are non-null, then a slash (/) is used to separate the devid from the 
minor_name in the encoded string. If minor_name is null, then only the devid is encoded.
If the devid is null, then the special string id0 is returned. Note that you cannot compare the returned string against another string with strcmp() to determine devid equality. The returned string must be freed by calling devid_str_free().

The ddi_devid_str_decode() function takes a string previously produced by the devid_str_encode(3DEVID) or ddi_devid_str_encode() function and decodes the contained device ID and minor_name, allocating and returning pointers to the extracted parts through the retdevid and retminor_name arguments. If the special devidstr id0 was specified then the returned device ID and minor name will both be null. A non-null returned devid must be freed by the caller through the ddi_devid_free() function. A non-null returned minor name must be freed by calling ddi_devid_str_free().

The ddi_devid_str_free() function is used to free all strings returned by the ddi_devid functions (the ddi_devid_str_encode() function return value and the returned retminor_name argument).

**RETURN VALUES**

**ddi_devid_init()** returns the following values:

- **DDI_SUCCESS** — Success.
- **DDI_FAILURE** — Out of memory. An invalid devid_type was passed in.

**ddi_devid_valid()** returns the following values:

- **DDI_SUCCESS** — Valid device ID.
- **DDI_FAILURE** — Invalid device ID.

**ddi_devid_register()** returns the following values:

- **DDI_SUCCESS** — Success.
- **DDI_FAILURE** — Failure. The device ID is already registered or the device ID is invalid.

**ddi_devid_valid()** returns the following values:

- **DDI_SUCCESS** — Valid device ID.
- **DDI_FAILURE** — Invalid device ID.

**ddi_devid_compare()** returns the following values:

- **−1** — The first device ID is less than the second device ID.
- **0** — The first device ID is equal to the second device ID.
- **1** — The first device ID is greater than the second device ID.

**ddi_devid_sizeof()** returns the size of the devid in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

DDI_SUCCESS
Success.

DDI_FAILURE
Failure; the devidstr string was not valid.

CONTEXT
These functions can be called from a user or kernel context.

SEE ALSO
devid_get(3DEVID),, libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
NAME

ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS

int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *rettdevid, char **retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

devid

The device id address.

devidstr

The devid and minor_name represented as a string.

devid1

The first of two device id addresses to be compared calling

  ddi_devid_compare().

devid2

The second of two device id addresses to be compared calling

  ddi_devid_compare().

dip

A dev_info pointer, which identifies the device.

devid_type

The following device id types may be accepted by the

  ddi_devid_init() function:

  DEVID_SCSI3_WWN World Wide Name associated with
  SCSI-3 devices.

  DEVID_SCSI_SERIAL Vendor ID and serial number
  associated with a SCSI device.
  Note: This may only be used if
  known to be unique; otherwise a
  fabricated device id must be used.

  DEVID_ENCAP Device ID of another device. This is
  for layered device driver usage.

  DEVID_FAB Fabricated device ID.

minor_name

The minor name to be encoded.

nbytes

The length in bytes of device ID.
The return address of the device ID.

The return address of a minor name. Free string with ddi_devid_str_free().

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

ddi_devid_compare() compares two device IDs byte-by-byte and determines both equality and sort order.

ddi_devid_sizeof() returns the number of bytes allocated for the passed in device ID (devid).

ddi_devid_init() allocates memory and initializes the opaque device ID structure. This function does not store the devid. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the devid on some reliable store. When a devid_type of either DEVID_SCSI3_WWN, DEVID_SCSI_SERIAL, or DEVID_ENCAP is accepted, an array of bytes (id) must be passed in (nbytes).

When the devid_type DEVID_FAB is used, the array of bytes (id) must be NULL and the length (nbytes) must be zero. The fabricated device ids, DEVID_FAB will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the ddi_devid_free() function.

ddi_devid_free() frees the memory allocated for the returned devid by the ddi_devid_init() and devid_str_decode() functions.

ddi_devid_register() registers the device ID address (devid) with the DDI framework, associating it with the dev_info passed in (dip). The drivers must register device IDs at attach time. See attach(9E).

ddi_devid_unregister() removes the device ID address from the dev_info passed in (dip). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the ddi_devid_free() function. See detach(9E).

ddi_devid_valid() validates the device ID (devid) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The ddi_devid_str_encode() function encodes a devid and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a devid and a minor_name are non-null, then a slash (/) is used to separate the devid from the minor_name in the encoded string. If minor_name is null, then only the devid is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` function and decodes the contained device ID and `minor_name`, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

- **ddi_devid_init()** returns the following values:
  - `DDI_SUCCESS` Success.
  - `DDI_FAILURE` Out of memory. An invalid `devid_type` was passed in.

- **ddi_devid_valid()** returns the following values:
  - `DDI_SUCCESS` Valid device ID.
  - `DDI_FAILURE` Invalid device ID.

- **ddi_devid_register()** returns the following values:
  - `DDI_SUCCESS` Success.
  - `DDI_FAILURE` Failure. The device ID is already registered or the device ID is invalid.

- **ddi_devid_valid()** returns the following values:
  - `DDI_SUCCESS` Valid device ID.
  - `DDI_FAILURE` Invalid device ID.

- **ddi_devid_compare()** returns the following values:
  - `-1` The first device ID is less than the second device ID.
  - `0` The first device ID is equal to the second device ID.
  - `1` The first device ID is greater than the second device ID.

- **ddi_devid_sizeof()** returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode(9F)

ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

DDI_SUCCESS
Success.

DDI_FAILURE
Failure; the devidstr string was not valid.

CONTEXT These functions can be called from a user or kernel context.

SEE ALSO ddevid_get(3DEVID),, libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
ddi_devid_str_free(9F)

NAME
ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS

int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
**retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS

deid
The device id address.
devidstr
The devid and minor_name represented as a string.
devid1
The first of two device id addresses to be compared calling
ddi_devid_compare().
devid2
The second of two device id addresses to be compared calling
ddi_devid_compare().
dip
A dev_info pointer, which identifies the device.
devid_type
The following device id types may be accepted by the
ddi_devid_init() function:

DEVID_SCSI3_WWN
World Wide Name associated with
SCSI-3 devices.

DEVID_SCSI_SERIAL
Vendor ID and serial number
associated with a SCSI device.
Note: This may only be used if
known to be unique; otherwise a
fabricated device id must be used.

DEVID_ENCAP
Device ID of another device. This is
for layered device driver usage.

DEVID_FAB
Fabricated device ID.

minor_name
The minor name to be encoded.
nbytes
The length in bytes of device ID.
The return address of the device ID.

retminor_name  The return address of a minor name. Free string with ddi_devid_str_free().

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

ddi_devid_compare() compares two device IDs byte-by-byte and determines both equality and sort order.

ddi_devid_sizeof() returns the number of bytes allocated for the passed in device ID (devid).

ddi_devid_init() allocates memory and initializes the opaque device ID structure. This function does not store the devid. If the device id is not derived from the device’s firmware, it is the driver’s responsibility to store the devid on some reliable store. When a devid_type of either DEVID_SCSI3_WWN, DEVID_SCSI_SERIAL, or DEVID_ENCAP is accepted, an array of bytes (id) must be passed in (nbytes).

When the devid_type DEVID_FAB is used, the array of bytes (id) must be NULL and the length (nbytes) must be zero. The fabricated device ids, DEVID_FAB will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the ddi_devid_free() function.

ddi_devid_free() frees the memory allocated for the returned devid by the ddi_devid_init() and devid_str_decode() functions.

ddi_devid_register() registers the device ID address (devid) with the DDI framework, associating it with the dev_info passed in (dip). The drivers must register device IDs at attach time. See attach(9E).

ddi_devid_unregister() removes the device ID address from the dev_info passed in (dip). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the ddi_devid_free() function. See detach(9E).

ddi_devid_valid() validates the device ID (devid) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The ddi_devid_str_encode() function encodes a devid and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a devid and a minor_name are non-null, then a slash (/) is used to separate the devid from the minor_name in the encoded string. If minor_name is null, then only the devid is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

`ddi_devid_register()` returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1` The first device ID is less than the second device ID.
- `0` The first device ID is equal to the second device ID.
- `1` The first device ID is greater than the second device ID.

`ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

DDI_SUCCESS
Success.

DDI_FAILURE
Failure; the devidstr string was not valid.

CONTEXT
These functions can be called from a user or kernel context.

SEE ALSO
devic_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
ddi_devid_unregister(9F)

NAME  ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free,
ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS  int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t
nbytes, void *id, ddi_devid_t *retdevid);
void ddi_devid_free(ddi_devid_t devid);
int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char
**retminor_name);
int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
int ddi_devid_str_free(char *devidstr);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS

devid  The device id address.
devidstr  The devid and minor_name represented as a string.
devid1  The first of two device id addresses to be compared calling
ddi_devid_compare().
devid2  The second of two device id addresses to be compared calling
ddi_devid_compare().
dip  A dev_info pointer, which identifies the device.
devid_type  The following device id types may be accepted by the
ddi_devid_init() function:

DEVID_SCSI3_WWN  World Wide Name associated with
SCSI-3 devices.

DEVID_SCSI_SERIAL  Vendor ID and serial number
associated with a SCSI device.
Note: This may only be used if
known to be unique; otherwise a
fabricated device id must be used.

DEVID_ENCAP  Device ID of another device. This is
for layered device driver usage.

DEVID_FAB  Fabricated device ID.

minor_name  The minor name to be encoded.
nbytes  The length in bytes of device ID.
The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device's physical connection or its logical device name or number.

- **ddi_devid_compare()** compares two device IDs byte-by-byte and determines both equality and sort order.

- **ddi_devid_sizeof()** returns the number of bytes allocated for the passed in device ID (devid).

- **ddi_devid_init()** allocates memory and initializes the opaque device ID structure. This function does not store the devid. If the device id is not derived from the device's firmware, it is the driver's responsibility to store the devid on some reliable store. When a devid_type of either DEVID_SCSI3_WWN, DEVID_SCSI_SERIAL, or DEVID_ENCAP is accepted, an array of bytes (id) must be passed in (nbytes).

When the devid_type DEVID_FAB is used, the array of bytes (id) must be NULL and the length (nbytes) must be zero. The fabricated device ids, DEVID_FAB will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the ddi_devid_free() function.

- **ddi_devid_free()** frees the memory allocated for the returned devid by the ddi_devid_init() and devid_str_decode() functions.

- **ddi_devid_register()** registers the device ID address (devid) with the DDI framework, associating it with the dev_info passed in (dip). The drivers must register device IDs at attach time. See attach(9E).

- **ddi_devid_unregister()** removes the device ID address from the dev_info passed in (dip). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the ddi_devid_free() function. See detach(9E).

- **ddi_devid_valid()** validates the device ID (devid) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The ddi_devid_str_encode() function encodes a devid and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a devid and a minor_name are non-null, then a slash (/) is used to separate the devid from the minor_name in the encoded string. If minor_name is null, then only the devid is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode(3DEVID)` or `ddi_devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

- `ddi_devid_init()` returns the following values:
  - `DDI_SUCCESS` Success.
  - `DDI_FAILURE` Out of memory. An invalid `devid_type` was passed in.

- `ddi_devid_valid()` returns the following values:
  - `DDI_SUCCESS` Valid device ID.
  - `DDI_FAILURE` Invalid device ID.

- `ddi_devid_register()` returns the following values:
  - `DDI_SUCCESS` Success.
  - `DDI_FAILURE` Failure. The device ID is already registered or the device ID is invalid.

- `ddi_devid_unregister()` returns the following values:
  - `DDI_SUCCESS` Valid device ID.
  - `DDI_FAILURE` Invalid device ID.

- `ddi_devid_compare()` returns the following values:
  - `-1` The first device ID is less than the second device ID.
  - `0` The first device ID is equal to the second device ID.
  - `1` The first device ID is greater than the second device ID.

- `ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

- **DDI_SUCCESS**
  - Success.

- **DDI_FAILURE**
  - Failure; the devidstr string was not valid.

**CONTEXT**

These functions can be called from a user or kernel context.

**SEE ALSO**

devid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

Writing Device Drivers
ddi_devid_valid(9F)

NAME  ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register, ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free, ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids

SYNOPSIS  int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
            size_t ddi_devid_sizeof(ddi_devid_t devid);
            int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t nbytes, void *id, ddi_devid_t *retdevid);
            void ddi_devid_free(ddi_devid_t devid);
            int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);
            int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char **retminor_name);
            int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);
            int ddi_devid_str_free(char *devidstr);
            void ddi_devid_unregister(dev_info_t *dip);
            int ddi_devid_valid(ddi_devid_t devid);

PARAMETERS

devid  The device id address.
devidstr  The devid and minor_name represented as a string.
devid1  The first of two device id addresses to be compared calling ddi_devid_compare().
devid2  The second of two device id addresses to be compared calling ddi_devid_compare().
dip  A dev_info pointer, which identifies the device.
devid_type  The following device id types may be accepted by the ddi_devid_init() function:
               DEVID_SCSI3_WWN  World Wide Name associated with SCSI-3 devices.
               DEVID_SCSI_SERIAL  Vendor ID and serial number associated with a SCSI device.
                                 Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.
               DEVID_ENCAP  Device ID of another device. This is for layered device driver usage.
               DEVID_FAB  Fabricated device ID.
minor_name  The minor name to be encoded.
nbytes  The length in bytes of device ID.
The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device’s physical connection or its logical device name or number.

- `ddi_devid_compare()` compares two device IDs byte-by-byte and determines both equality and sort order.
- `ddi_devid_sizeof()` returns the number of bytes allocated for the passed in device ID (`devid`).
- `ddi_devid_init()` allocates memory and initializes the opaque device ID structure. This function does not store the `devid`. If the device id is not derived from the device’s firmware, it is the driver's responsibility to store the `devid` on some reliable store. When a `devid_type` of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (`id`) must be passed in (`nbytes`).

When the `devid_type` `DEVID_FAB` is used, the array of bytes (`id`) must be NULL and the length (`nbytes`) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine’s host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

- `ddi_devid_free()` frees the memory allocated for the returned `devid` by the `ddi_devid_init()` and `devid_str_decode()` functions.
- `ddi_devid_register()` registers the device ID address (`devid`) with the DDI framework, associating it with the `dev_info` passed in (`dip`). The drivers must register device IDs at attach time. See `attach(9E)`.
- `ddi_devid_unregister()` removes the device ID address from the `dev_info` passed in (`dip`). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_devid_free()` function. See `detach(9E)`.
- `ddi_devid_valid()` validates the device ID (`devid`) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The `ddi_devid_str_encode()` function encodes a `devid` and minor_name into a null-terminated ASCII string, returning a pointer to that string. If both a `devid` and a `minor_name` are non-null, then a slash (`/`) is used to separate the `devid` from the `minor_name` in the encoded string. If `minor_name` is null, then only the `devid` is encoded.
If the `devid` is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine `devid` equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode()` function and decodes the contained device ID and minor name, allocating and returning pointers to the extracted parts through the `retdevid` and `retminor_name` arguments. If the special `devidstr id0` was specified then the returned device ID and minor name will both be null. A non-null returned `devid` must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned `retminor_name` argument).

**RETURN VALUES**

`ddi_devid_init()` returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Out of memory. An invalid `devid_type` was passed in.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

`ddi_devid_register()` returns the following values:

- **DDI_SUCCESS** Success.
- **DDI_FAILURE** Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- **DDI_SUCCESS** Valid device ID.
- **DDI_FAILURE** Invalid device ID.

`ddi_devid_compare()` returns the following values:

- `-1` The first device ID is less than the second device ID.
- `0` The first device ID is equal to the second device ID.
- `1` The first device ID is greater than the second device ID.

`ddi_devid_sizeof()` returns the size of the `devid` in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.
ddi_devid_str_encode() returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid devid. If the return value is non-null then the caller must free the returned string by using the devid_str_free() function.

ddi_devid_str_decode() returns the following values:

- **DDI_SUCCESS**
  - Success.

- **DDI_FAILURE**
  - Failure; the devidstr string was not valid.

**CONTEXT**

These functions can be called from a user or kernel context.

**SEE ALSO**

- devid_get(3DEVID), libdevid(3LIB), attributes(5), attach(9E), detach(9E), kmem_free(9F)

*Writing Device Drivers*
### NAME
/ddi_dev_is_needed(9F)

The `ddi_dev_is_needed` function informs the system that a device's component is required.

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_is_needed(dev_info_t *dip, int component, int level);
```

### INTERFACE
Solaris DDI specific (Solaris DDI)

### LEVEL
- **dip**: Pointer to the device's `dev_info` structure.
- **component**: Component of the driver which is needed.
- **level**: Power level at which the component is needed.

### DESCRIPTION
The `ddi_dev_is_needed` function is obsolete and will be removed in a future release. It is recommended that device drivers use `pm_raise_power(9F)` and `pm_lower_power(9F)`.

The `ddi_dev_is_needed` function informs the system that a device component is needed at the specified power level. The `level` argument must be non-zero.

This function sets a `component` to the required level and sets all devices which depend on this to their normal power levels. If `component 0` of a device using original Power Management interfaces (calls `pm_create_components(9F)`) is at power level 0, the `ddi_dev_is_needed()` call will result in component 0 being returned to normal power and the device being resumed via `attach(9E)` before `ddi_dev_is_needed()` returns.

The state of the device should be examined before each physical access. The `ddi_dev_is_needed()` function should be called to set a `component` to the required power level if the operation to be performed requires the component to be at a power level other than its current level.

The `ddi_dev_is_needed()` function might cause re-entry of the driver. Deadlock may result if driver locks are held across the call to `ddi_dev_is_needed()`.

### RETURN VALUES
The `ddi_dev_is_needed()` function returns:
- **DDI_SUCCESS**: Power successfully set to the requested level.
- **DDI_FAILURE**: An error occurred.

### EXAMPLES
#### EXAMPLE 1 disk driver code
A hypothetical disk driver might include this code:
```c
static int
xxdisk_spun_down(struct xxstate *xsp)
{
    return (xsp->power_level[DISK_COMPONENT] < POWER_SPUN_UP);
}

static int
xxdisk_strategy(struct buf *bp)
```

---

368 man pages section 9: DDI and DKI Kernel Functions • Last Revised 15 Oct 1999
EXAMPLE 1 disk driver code  (Continued)

{...

    mutex_enter(&xxstate_lock);
    /*
     * Since we have to drop the mutex, we have to do this in a loop
     * in case we get preempted and the device gets taken away from
     * us again
     */
    while (device_spun_down(sp)) {
        mutex_exit(&xxstate_lock);
        if (ddi_dev_is_needed(xsp->mydip,
                XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
            bioerror(bp,EIO);
            biodone(bp);
            return (0);
        }
        mutex_enter(&xxstate_lock);
    }
    xsp->device_busy++;
    mutex_exit(&xxstate_lock);

    ...
}

CONTEXT
This function can be called from user or kernel context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
pm(7D), pm-components(9P), attach(9E), detach(9E), power(9E),
pm_busy_components(9F), pm_create_components(9F),
pm_destroy_components(9F), pm_idle_component(9F)

Writing Device Drivers
ddi_dev_is_sid(9F)

NAME  ddi_dev_is_sid – tell whether a device is self-identifying

SYNOPSIS  
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_is_sid(dev_info_t *dip);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

PARAMETERS  dip  A pointer to the device’s dev_info structure.

DESCRIPTION  ddi_dev_is_sid() tells the caller whether the device described by dip is self-identifying, that is, a device that can unequivocally tell the system that it exists. This is useful for drivers that support both a self-identifying as well as a non-self-identifying variants of a device (and therefore must be probed).

RETURN VALUES  
DDI_SUCCESS  Device is self-identifying.
DDI_FAILURE  Device is not self-identifying.

CONTEXT  ddi_dev_is_sid() can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1

1 ...  
2 int  
3 bz_probe(dev_info_t *dip)  
4 {  
5   ...  
6   if (ddi_dev_is_sid(dip) == DDI_SUCCESS) {  
7     /*  
8       * This is the self-identifying version (OpenBoot).  
9       * No need to probe for it because we know it is there.  
10       * The existence of dip && ddi_dev_is_sid() proves this.  
11       */  
12       return (DDI_PROBE_DONTCARE);  
13   }  
14   /*  
15     * Not a self-identifying variant of the device. Now we have to  
16     * do some work to see whether it is really attached to the  
17     * system.  
18     */  
19   ...  

SEE ALSO  probe(9E) Writing Device Drivers
**NAME**

`devmap_setup`, `ddi_devmap_segmap` – set up a user mapping to device memory using the devmap framework

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_setup(dev_t dev, offset_t off, ddi_as_handle_t as,
                  caddr_t *addrp, size_t len, uint_t prot, uint_t maxprot, uint_t flags,
                  cred_t *cred);

int ddi_devmap_segmap(dev_t dev, off_t off, ddi_as_handle_t as,
                       caddr_t *addrp, off_t len, uint_t prot, uint_t maxprot, uint_t flags,
                       cred_t *cred);
```

Solaris DDI specific (Solaris DDI).

**INTERFACE LEVEL**

**PARAMETERS**

dev

Device whose memory is to be mapped.

off

User offset within the logical device memory at which the mapping begins.

as

An opaque data structure that describes the address space into which the device memory should be mapped.

addrp

Pointer to the starting address in the address space into which the device memory should be mapped.

len

Length (in bytes) of the memory to be mapped.

prot

A bit field that specifies the protections. Some possible settings combinations are:

- **PROT_READ** Read access is desired.
- **PROT_WRITE** Write access is desired.
- **PROT_EXEC** Execute access is desired.
- **PROT_USER** User-level access is desired (the mapping is being done as a result of a `mmap(2)` system call).
- **PROT_ALL** All access is desired.

maxprot

Maximum protection flag possible for attempted mapping; the **PROT_WRITE** bit may be masked out if the user opened the special file read-only.

flags

Flags indicating type of mapping. The following flags can be specified:

- **MAP_PRIVATE** Changes are private.
- **MAP_SHARED** Changes should be shared.
- **MAP_FIXED** The user specified an address in `*addrp` rather than letting the system choose an address.

cred

Pointer to the user credential structure.
**devmap_setup()** and **ddi_devmap_segmap()** allow device drivers to use the devmap framework to set up user mappings to device memory. The devmap framework provides several advantages over the default device mapping framework that is used by **ddi_segmap(9F)** or **ddi_segmap_setup(9F)**. Device drivers should use the devmap framework, if the driver wants to:

- use an optimal MMU pagesize to minimize address translations,
- conserve kernel resources,
- receive callbacks to manage events on the mapping,
- export kernel memory to applications,
- set up device contexts for the user mapping if the device requires context switching,
- assign device access attributes to the user mapping, or
- change the maximum protection for the mapping.

**devmap_setup()** must be called in the **segmap(9E)** entry point to establish the mapping for the application. **ddi_devmap_segmap()** can be called in, or be used as, the **segmap(9E)** entry point. The differences between **devmap_setup()** and **ddi_devmap_segmap()** are in the data type used for `off` and `len`.

When setting up the mapping, **devmap_setup()** and **ddi_devmap_segmap()** call the **devmap(9E)** entry point to validate the range to be mapped. The **devmap(9E)** entry point also translates the logical offset (as seen by the application) to the corresponding physical offset within the device address space. If the driver does not provide its own devmap(9E) entry point, **EINVAL** will be returned to the **mmap(2)** system call.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>Non-zero</td>
<td>An error occurred. The return value of <strong>devmap_setup()</strong> and <strong>ddi_devmap_segmap()</strong> should be used directly in the <strong>segmap(9E)</strong> entry point.</td>
</tr>
</tbody>
</table>

**CONTEXT**

**devmap_setup()** and **ddi_devmap_segmap()** can be called from user or kernel context only.

**SEE ALSO**

**mmap(2)**, **devmap(9E)**, **segmap(9E)**, **ddi_segmap(9F)**, **ddi_segmap_setup(9F)**, **cb_ops(9S)**

Writing Device Drivers
ddi_dev_nintrs - return the number of interrupt specifications a device has

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_nintrs(dev_info_t *dip, int *resultp);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION ddi_dev_nintrs() returns the number of interrupt specifications a device has in *resultp.

RETURN VALUES ddi_dev_nintrs() returns:
DDI_SUCCESS A successful return. The number of interrupt specifications that the device has is set in resultp.
DDI_FAILURE The device has no interrupt specifications.

CONTEXT ddi_dev_nintrs() can be called from user or interrupt context.

SEE ALSO isa(4), sbus(4), ddi_add_intr(9F), ddi_dev_nregs(9F), ddi_dev_regsize(9F)

Writing Device Drivers
NAME

ddi_dev_nregs – return the number of register sets a device has

SYNOPSIS

#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_nregs(dev_info_t *dip, int *resultp);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dip

A pointer to the device’s dev_info structure.

resultp

Pointer to an integer that holds the number of register sets on return.

DESCRIPTION

The function ddi_dev_nregs() returns the number of sets of registers the device has.

RETURN VALUES

ddi_dev_nregs() returns:

DDI_SUCCESS

A successful return. The number of register sets is returned in resultp.

DDI_FAILURE

The device has no registers.

CONTEXT

ddi_dev_nregs() can be called from user or interrupt context.

SEE ALSO

ddi_dev_nintrs(9F), ddi_dev_regsize(9F)

Writing Device Drivers
ddi_dev_regsize(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_dev_regsize – return the size of a device’s register</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/conf.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/sunddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int ddi_dev_regsize(dev_info_t *dip, uint_t rnumber, off_t *resultp);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>dip A pointer to the device’s dev_info structure.</td>
</tr>
<tr>
<td></td>
<td>rnumber The ordinal register number. Device registers are</td>
</tr>
<tr>
<td></td>
<td>are associated with a dev_info and are enumerated in</td>
</tr>
<tr>
<td></td>
<td>arbitrary sets from 0 on up. The number of registers a</td>
</tr>
<tr>
<td></td>
<td>device has can be determined from a call to ddi_dev_nregs(9F).</td>
</tr>
<tr>
<td></td>
<td>resultp Pointer to an integer that holds the size, in</td>
</tr>
<tr>
<td></td>
<td>bytes, of the described register (if it exists).</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>ddi_dev_regsize() returns the size, in bytes, of the</td>
</tr>
<tr>
<td></td>
<td>device register specified by dip and rnumber. This is</td>
</tr>
<tr>
<td></td>
<td>useful when, for example, one of the registers is a frame</td>
</tr>
<tr>
<td></td>
<td>buffer with a varying size known only to its proms.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ddi_dev_regsize() returns:</td>
</tr>
<tr>
<td>DDI_SUCCESS</td>
<td>A successful return. The size, in bytes, of the specified</td>
</tr>
<tr>
<td></td>
<td>register, is set in resultp.</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
<td>An invalid (nonexistent) register number was specified.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>ddi_dev_regsize() can be called from user or interrupt</td>
</tr>
<tr>
<td></td>
<td>context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>ddi_dev_nintrs(9F), ddi_dev_nregs(9F)</td>
</tr>
</tbody>
</table>

Writing Device Drivers

Kernel Functions for Drivers 375
ddi_dev_report_fault(9F)

NAME
ddi_dev_report_fault – Report a hardware failure

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_dev_report_fault (dev_info_t *dip, ddi_fault_impact_t impact, ddi_fault_location_t location, const char *message);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
dip Pointer to the driver’s dev_info structure to which the fault report relates. (Normally the caller’s own dev_info pointer).
impact One of a set of enumerated values indicating the impact of the fault on the device’s ability to provide normal service.
location One of a set of enumerated values indicating the location of the fault, relative to the hardware controlled by the driver specified by dip.
message Text of the message describing the fault being reported.

DESCRIPTION
This function provides a standardized mechanism through which device drivers can report hardware faults. Use of this reporting mechanism enables systems equipped with a fault management system to respond to faults discovered by a driver. On a suitably equipped system, this might include automatic failover to an alternative device and/or scheduling replacement of the faulty hardware.

The driver must indicate the impact of the fault being reported on its ability to provide service by passing one of the following values for the impact parameter:

DDI_SERVICE_LOST
Indicates a total loss of service. The driver is unable to implement the normal functions of its hardware.

DDI_SERVICE_DEGRADED
The driver is unable to provide normal service, but can provide a partial or degraded level of service. The driver may have to make repeated attempts to perform an operation before it succeeds, or it may be running at less than its configured speed. A driver may use this value to indicate that an alternative device should be used if available, but that it can continue operation if no alternative exists.

DDI_SERVICE_UNAFFECTED
The service provided by the device is currently unaffected by the reported fault. This value may be used to report recovered errors for predictive failure analysis.

DDI_SERVICE_RESTORED
The driver has resumed normal service, following a previous report that service was lost or degraded. This message implies that any previously reported fault condition no longer exists.

The location parameter should be one of the following values:
ddi_dev_report_fault(9F)

**DDI_DATAPATH_FAULT**
The fault lies in the datapath between the driver and the device. The device may be unplugged, or a problem may exist in the bus on which the device resides. This value is appropriate if the device is not responding to accesses, (for example, the device may not be present) or if a call to `ddi_check_acc_handle(9F)` returns `DDI_FAILURE`.

**DDI_DEVICE_FAULT**
The fault lies in the device controlled by the driver. This value is appropriate if the device returns an error from a selftest function, or if the driver is able to determine that device is present and accessible, but is not functioning correctly.

**DDI_EXTERNAL_FAULT**
The fault is external to the device. For example, an Ethernet driver would use this value when reporting a cable fault.

If a device returns detectably bad data during normal operation (an "impossible" value in a register or DMA status area, for example), the driver should check the associated handle using `ddi_check_acc_handle(9F)` or `ddi_check_dma_handle(9F)` before reporting the fault. If the fault is associated with the handle, the driver should specify `DDI_DATAPATH_FAULT` rather than `DDI_DEVICE_FAULT`. As a consequence of this call, the device's state may be updated to reflect the level of service currently available. See `ddi_get_devstate(9F)`.

Note that if a driver calls `ddi_get_devstate(9F)` and discovers that its device is down, a fault should not be reported- the device is down as the result of a fault that has already been reported. Additionally, a driver should avoid incurring or reporting additional faults when the device is already known to be unusable. The `ddi_dev_report_fault()` call should only be used to report hardware (device) problems and should not be used to report purely software problems such as memory (or other resource) exhaustion.

**EXAMPLES**
An Ethernet driver receives an error interrupt from its device if various fault conditions occur. The driver must read an error status register to determine the nature of the fault, and report it appropriately:

```c
static int xx_error_intr(xx_soft_state *ssp)
{
    ...
    error_status = ddi_get32(ssp->handle, &ssp->regs->xx_err_status);
    if (ddi_check_acc_handle(ssp->handle) != DDI_SUCCESS) {
        ddi_dev_report_fault(ssp->dip, DDI_SERVICE_LOST,
                             DDI_DATAPATH_FAULT, "register access fault");
        return DDI_INTR_UNCLAIMED;
    }
    if (ssp->error_status & XX_CABLE_FAULT) {
        ddi_dev_report_fault(ssp->dip, DDI_SERVICE_LOST,
                             DDI_EXTERNAL_FAULT, "cable fault");
        return DDI_INTR_CLAIMED;
    }
    if (ssp->error_status & XX_JABBER) {
```

Kernel Functions for Drivers  377
ddi_dev_report_fault(9F)

```c
  ddi_dev_report_fault(ssp->dip, DDI_SERVICE_DEGRADED,
                      DDI_EXTERNAL_FAULT, "jabbering detected")
  return DDI_INTR_CLAIMED;
  }
  ...
  }

CONTEXT  The ddi_dev_report_fault() function may be called from user, kernel, or interrupt context.

SEE ALSO  ddi_check_acc_handle(9F), ddi_check_dma_handle(9F),
           ddi_get_devstate(9F)
### NAME
ddi_dma_addr_bind_handle – binds an address to a DMA handle

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, struct as *, caddr_t addr, size_t len, uint_t flags, int (*callback)(caddr_t), caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

Solaris DDI specific (Solaris DDI).

### INTERFACE LEVEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</td>
</tr>
<tr>
<td>as</td>
<td>A pointer to an address space structure. This parameter should be set to NULL, which implies kernel address space.</td>
</tr>
<tr>
<td>addr</td>
<td>Virtual address of the memory object.</td>
</tr>
<tr>
<td>len</td>
<td>Length of the memory object in bytes.</td>
</tr>
<tr>
<td>flags</td>
<td>Valid flags include:</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_WRITE Transfer direction is from memory to I/O.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_READ Transfer direction is from I/O to memory.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_RDWR Both read and write.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_REDZONE Establish an MMU redzone at end of the object.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_PARTIAL Partial resource allocation.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_CONSISTENT Nonsequential, random, and small block transfers.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_STREAMING Sequential, unidirectional, block-sized, and block-aligned transfers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>callback</th>
<th>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DDI_DMA_SLEEP Wait until resources are available.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_DONTWAIT Do not wait until resources are available and do not schedule a callback.</td>
</tr>
</tbody>
</table>

| arg       | Argument to be passed to the callback function, callback, if such a function is specified. |
ddi_dma_addr_bind_handle(9F)

cookiep     A pointer to the first ddi_dma_cookie(9S) structure.
ccountp     Upon a successful return, ccountp points to a value representing
            the number of cookies for this DMA object.

DESCRIPTION

ddi_dma_addr_bind_handle() allocates DMA resources for a memory object such
that a device can perform DMA to or from the object. DMA resources are allocated
considering the device’s DMA attributes as expressed by ddi_dma_attr(9S) (see
ddi_dma_alloc_handle(9F)).

ddi_dma_addr_bind_handle() fills in the first DMA cookie pointed to by cookiep
with the appropriate address, length, and bus type. *ccountp is set to the number of
DMA cookies representing this DMA object. Subsequent DMA cookies must be
retrieved by calling ddi_dma_nextcookie(9F) the number of times specified by
*ccountp-1.

When a DMA transfer completes, the driver frees up system DMA resources by calling
ddi_dma_unbind_handle(9F).

The flags argument contains information for mapping routines.

DDI_DMA_WRITE, DDI_DMA_READ, DDI_DMA_RDWR
  These flags describe the intended direction of the DMA transfer.

DDI_DMA_STREAMING
  This flag should be set if the device is doing sequential, unidirectional, block-sized,
  and block-aligned transfers to or from memory. The alignment and padding
  constraints specified by the minxfer and burstsizes fields in the DMA attribute
  structure, ddi_dma_attr(9S) (see ddi_dma_alloc_handle(9F)) is used to
  allocate the most effective hardware support for large transfers.

DDI_DMA_CONSISTENT
  This flag should be set if the device accesses memory randomly, or if
  synchronization steps using ddi_dma_sync(9F) need to be as efficient as possible.
  I/O parameter blocks used for communication between a device and a driver
  should be allocated using DDI_DMA_CONSISTENT.

DDI_DMA_REDZONE
  If this flag is set, the system attempts to establish a protected red zone after the
  object. The DMA resource allocation functions do not guarantee the success of this
  request as some implementations may not have the hardware ability to support a
  red zone.

DDI_DMA_PARTIAL
  Setting this flag indicates the caller can accept resources for part of the object. That
  is, if the size of the object exceeds the resources available, only resources for a
  portion of the object are allocated. The system indicates this condition by returning
  status DDI_DMA_PARTIAL_MAP. At a later point, the caller can use
  ddi_dma_getwin(9F) to change the valid portion of the object for which resources
  are allocated. If resources were allocated for only part of the object,
  ddi_dma_addr_bind_handle() returns resources for the first DMAwindow.
Even when DDI_DMA_PARTIAL is set, the system may decide to allocate resources for the entire object (less overhead) in which case DDI_DMA_MAPPED is returned.

The callback function *callback* indicates how a caller wants to handle the possibility of resources not being available. If *callback* is set to DDI_DMA_DONTWAIT, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If *callback* is set to DDI_DMA_SLEEP, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set and a DMA resource allocation fails, this value is assumed to be the address of a function to be called when resources become available. When the specified function is called, *arg* is passed to it as an argument. The specified callback function must return either DDI_DMA_CALLBACK_RUNOUT or DDI_DMA_CALLBACK_DONE.

DDI_DMA_CALLBACK_RUNOUT indicates that the callback function attempted to allocate DMA resources but failed. In this case, the callback function is put back on a list to be called again later. DDI_DMA_CALLBACK_DONE indicates that either the allocation of DMA resources was successful or the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are be available. The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_DMA_MAPPED</td>
<td>Successfully allocated resources for the entire object.</td>
</tr>
<tr>
<td>DDI_DMA_PARTIAL_MAP</td>
<td>Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the DDI_DMA_PARTIAL flag in <em>flags</em>.</td>
</tr>
<tr>
<td>DDI_DMA_INUSE</td>
<td>Another I/O transaction is using the DMA handle.</td>
</tr>
<tr>
<td>DDI_DMA_NORESOURCES</td>
<td>No resources are available at the present time.</td>
</tr>
<tr>
<td>DDI_DMA_NOMAPPING</td>
<td>The object cannot be reached by the device requesting the resources.</td>
</tr>
<tr>
<td>DDI_DMA_TOOBIG</td>
<td>The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration.</td>
</tr>
</tbody>
</table>

**CONTEXT**

*ddi_dma_addr_bind_handle()* can be called from user, kernel, or interrupt context, except when *callback* is set to DDI_DMA_SLEEP, in which case it can only be called from user or kernel context.

**SEE ALSO**

*ddi_dma_alloc_handle(9F), ddi_dma_free_handle(9F), ddi_dma_getwin(9F), ddi_dma_mem_alloc(9F), ddi_dma_mem_free(9F), ddi_dma_nextcookie(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F), ddi_umem_iosetup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S)*
If the driver permits partial mapping with the DDI_DMA_PARTIAL flag, the number of cookies in each window may exceed the size of the device’s scatter/gather list as specified in the dma_attr_sgllen field in the ddi_dma_attr(9S) structure. In this case, each set of cookies comprising a DMA window will satisfy the DMA attributes as described in the ddi_dma_attr(9S) structure in all aspects. The driver should set up its DMA engine and perform one transfer for each set of cookies sufficient for its scatter/gather list, up to the number of cookies for this window, before advancing to the next window using ddi_dma_getwin(9F).
ddi_dma_addr_setup – easier DMA setup for use with virtual addresses

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_addr_setup(dev_info_t *dip, struct as *as, caddr_t addr,
                        size_t len, uint_t flags, int (*waitfp) (caddr_t),
                        caddr_t arg, ddi_dma_lim_t *lim, ddi_dma_handle_t *handlep);
```

This interface is obsolete. `ddi_dma_addr_bind_handle(9F)` should be used instead.

**Parameters**

- **dip**: A pointer to the device’s dev_info structure.
- **as**: A pointer to an address space structure. Should be set to NULL, which implies kernel address space.
- **addr**: Virtual address of the memory object.
- **len**: Length of the memory object in bytes.
- **flags**: Flags that would go into the ddi_dma_req structure (see ddi_dma_req(9S)).
- **waitfp**: The address of a function to call back later if resources aren’t available now. The special function addresses DDI_DMA_SLEEP and DDI_DMA_DONTWAIT (see ddi_dma_req(9S)) are taken to mean, respectively, wait until resources are available or, do not wait at all and do not schedule a callback.
- **arg**: Argument to be passed to a callback function, if such a function is specified.
- **lim**: A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). If this pointer is NULL, a default set of DMA limits is assumed.
- **handlep**: Pointer to a DMA handle. See ddi_dma_setup(9F) for a discussion of handle.

**Description**

`ddi_dma_addr_setup()` is an interface to ddi_dma_setup(9F). It uses its arguments to construct an appropriate ddi_dma_req structure and calls ddi_dma_setup(9F) with it.

**Return Values**

See ddi_dma_setup(9F) for the possible return values for this function.

**Context**

`ddi_dma_addr_setup()` can be called from user or interrupt context, except when waitfp is set to DDI_DMA_SLEEP, in which case it can be called from user context only.

**Attributes**

See attributes(5) for a description of the following attributes:
ddi_dma_addr_setup(9F)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), ddi_dma_buf_setup(9F), ddi_dma_free(9F),
ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_sync(9F),
ddi_iopb_alloc(9F), ddi_dma_limit_sparc(9S), ddi_dma_limit_IA(9S),
ddi_dma_req(9S)

Writing Device Drivers

384  man pages section 9: DDI and DKI Kernel Functions • Last Revised 27 Sep 2002
ddi_dma_alloc_handle(9F)

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_alloc_handle(dev_info_t *dip, ddi_dma_attr_t *attr, int (*callback) (caddr_t), caddr_t arg, ddi_dma_handle_t *handlep);

Solaris DDI specific (Solaris DDI).

**dip**
Pointer to the device’s dev_info structure.

**attr**
Pointer to a DMA attribute structure for this device (see ddi_dma_attr(9S)).

**callback**
The address of a function to call back later if resources aren’t available now. The following special function addresses may also be used.

- **DDI_DMA_SLEEP** Wait until resources are available.
- **DDI_DMA_DONTWAIT** Do not wait until resources are available and do not schedule a callback.

**arg**
Argument to be passed to a callback function, if such a function is specified.

**handlep**
Pointer to the DMA handle to be initialized.

ddi_dma_alloc_handle() allocates a new DMA handle. A DMA handle is an opaque object used as a reference to subsequently allocated DMA resources. ddi_dma Alloc_handle() accepts as parameters the device information referred to by dip and the device’s DMA attributes described by a ddi_dma_attr(9S) structure. A successful call to ddi_dma_alloc_handle() fills in the value pointed to by handlep. A DMA handle must only be used by the device for which it was allocated and is only valid for one I/O transaction at a time.

The callback function, callback, indicates how a caller wants to handle the possibility of resources not being available. If callback is set to DDI_DMA_DONTWAIT, then the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If callback is set to DDI_DMA_SLEEP, then the caller wishes to have the the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be a function to call at a later time when resources may become available. When the specified function is called, it is passed arg as an argument. The specified callback function must return either DDI_DMA_CALLBACK_RUNOUT or DDI_DMA_CALLBACK_DONE. DDI_DMA_CALLBACK_RUNOUT indicates that the callback routine attempted to allocate DMA resources but failed to do so, in which case the callback function is put back on a list to be called again later. DDI_DMA_CALLBACK_DONE indicates either success at allocating DMA resources or the driver no longer wishes to retry.
The callback function is called in interrupt context. Therefore, only system functions that are accessible from interrupt context is available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, and so forth.

When a DMA handle is no longer needed, ddi_dma_free_handle(9F) must be called to free the handle.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDI_SUCCESS</strong></td>
<td>Successfully allocated a new DMA handle.</td>
</tr>
<tr>
<td><strong>DDI_DMA_BADATTR</strong></td>
<td>The attributes specified in the ddi_dma_attr(9S) structure make it impossible for the system to allocate potential DMA resources.</td>
</tr>
<tr>
<td><strong>DDI_DMA_NORESOURCES</strong></td>
<td>No resources are available.</td>
</tr>
</tbody>
</table>

**CONTEXT**

ddi_dma_alloc_handle() can be called from user, kernel, or interrupt context, except when callback is set to DDI_DMA_SLEEP, in which case it can be called from user or kernel context only.

**SEE ALSO**

ddi_dma_addr_bind_handle(9F), ddi_dma_buf_bind_handle(9F), ddi_dma_burstsizes(9F), ddi_dma_free_handle(9F), ddi_dma_unbind_handle(9F), ddi_dma_attr(9S)

Writing Device Drivers
### NAME

`ddi_dma_buf_bind_handle` - binds a system buffer to a DMA handle

### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle, struct buf *bp, uint_t flags, int (*callback)(caddr_t), caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

### PARAMETERS

- **handle**: The DMA handle previously allocated by a call to `ddi_dma_alloc_handle(9F)`.
- **bp**: A pointer to a system buffer structure (see `buf(9S)`).
- **flags**: Valid flags include:
  - `DDI_DMA_WRITE`: Transfer direction is from memory to I/O
  - `DDI_DMA_READ`: Transfer direction is from I/O to memory
  - `DDI_DMA_RDWR`: Both read and write
  - `DDI_DMA_REDZONE`: Establish an MMU redzone at end of the object.
  - `DDI_DMA_PARTIAL`: Partial resource allocation
  - `DDI_DMA_CONSISTENT`: Nonsequential, random, and small block transfers.
  - `DDI_DMA_STREAMING`: Sequential, unidirectional, block-sized, and block-aligned transfers.
- **callback**: The address of a function to call back later if resources are not available now. The following special function addresses may also be used.
  - `DDI_DMA_SLEEP`: Wait until resources are available.
  - `DDI_DMA_DONTWAIT`: Do not wait until resources are available and do not schedule a callback.
- **arg**: Argument to be passed to the callback function, `callback`, if such a function is specified.
- **cookiep**: A pointer to the first `ddi_dma_cookie(9S)` structure.
- **ccountp**: Upon a successful return, `ccountp` points to a value representing the number of cookies for this DMA object.
**ddi_dma_buf_bind_handle()**

**DESCRIPTION**

`ddi_dma_buf_bind_handle()` allocates DMA resources for a system buffer such that a device can perform DMA to or from the buffer. DMA resources are allocated considering the device’s DMA attributes as expressed by `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`).

`ddi_dma_buf_bind_handle()` fills in the first DMA cookie pointed to by `cookiep` with the appropriate address, length, and bus type. `*countp` is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved by calling `ddi_dma_nextcookie(9F)` `*countp`-1 times.

When a DMA transfer completes, the driver should free up system DMA resources by calling `ddi_dma_unbind_handle(9F)`.

The `flags` argument contains information for mapping routines.

- **DDI_DMA_WRITE**, **DDI_DMA_READ**, **DDI_DMA_RDWR**
  These flags describe the intended direction of the DMA transfer.

- **DDI_DMA_STREAMING**
  This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the `minxfer` and `burstsizes` fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`) is used to allocate the most effective hardware support for large transfers.

- **DDI_DMA_CONSISTENT**
  This flag should be set if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using `DDI_DMA_CONSISTENT`.

- **DDI_DMA_REDZONE**
  If this flag is set, the system attempts to establish a protected red zone after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support a red zone.

- **DDI_DMA_PARTIAL**
  Setting this flag indicates the caller can accept resources for part of the object. That is, if the size of the object exceeds the resources available, only resources for a portion of the object are allocated. The system indicates this condition returning status `DDI_DMA_PARTIAL_MAP`. At a later point, the caller can use `ddi_dma_getwin(9F)` to change the valid portion of the object for which resources are allocated. If resources were allocated for only part of the object, `ddi_dma_addr_bind_handle()` returns resources for the first DMA window. Even when `DDI_DMA_PARTIAL` is set, the system may decide to allocate resources for the entire object (less overhead) in which case `DDI_DMA_MAPPED` is returned.

The callback function, `callback`, indicates how a caller wants to handle the possibility of resources not being available. If `callback` is set to `DDI_DMA_DONTWAIT`, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If
callback is set to DDI_DMA_SLEEP, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be the address of a function to call at a later time when resources may become available. When the specified function is called, it is passed arg as an argument. The specified callback function must return either DDI_DMA_CALLBACK_RUNOUT or DDI_DMA_CALLBACK_DONE.

DDI_DMA_CALLBACK_RUNOUT indicates that the callback function attempted to allocate DMA resources but failed to do so. In this case the callback function is put back on a list to be called again later. DDI_DMA_CALLBACK_DONE indicates either a successful allocation of DMA resources or that the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are be available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, etc.

**RETURN VALUES**

ddi_dma_buf_bind_handle() returns:

- **DDI_DMA_MAPPED**
  Successfully allocated resources for the entire object.

- **DDI_DMA_PARTIAL_MAP**
  Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the DDI_DMA_PARTIAL flag in flags.

- **DDI_DMA_INUSE**
  Another I/O transaction is using the DMA handle.

- **DDI_DMA_NORESOURCES**
  No resources are available at the present time.

- **DDI_DMA_NOMAPPING**
  The object cannot be reached by the device requesting the resources.

- **DDI_DMA_TOOBIG**
  The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration.

**CONTEXT**

ddi_dma_buf_bind_handle() can be called from user, kernel, or interrupt context, except when callback is set to DDI_DMA_SLEEP, in which case it can be called from user or kernel context only.

**SEE ALSO**

ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F), ddi_dma_free_handle(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F), buf(9S), ddi_dma_attr(9S), ddi_dma_cookie(9S)

*Writing Device Drivers*
If the driver permits partial mapping with the DDI_DMA_PARTIAL flag, the number of cookies in each window may exceed the size of the device’s scatter/gather list as specified in the dma_attr_sgllen field in the ddi_dma_attr(9S) structure. In this case, each set of cookies comprising a DMA window will satisfy the DMA attributes as described in the ddi_dma_attr(9S) structure in all aspects. The driver should set up its DMA engine and perform one transfer for each set of cookies sufficient for its scatter/gather list, up to the number of cookies for this window, before advancing to the next window using ddi_dma_getwin(9F).
ddi_dma_buf_setup(9F)

NAME
ddi_dma_buf_setup – easier DMA setup for use with buffer structures

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_buf_setup(dev_info_t *dip, struct buf *bp, uint_t flags, int (*waitfp) (caddr_t), caddr_t arg, ddi_dma_lim_t *lim, ddi_dma_handle_t *handlep);

INTERFACE

LEVEL

PARAMETERS

This interface is obsolete. ddi_dma_buf_bind_handle(9F) should be used instead.

dip A pointer to the device’s dev_info structure.

bp A pointer to a system buffer structure (see buf(9S)).

flags Flags that go into a ddi_dma_req structure (see ddi_dma_req(9S)).

waitfp The address of a function to call back later if resources aren’t available now. The special function addresses DDI_DMA_SLEEP and DDI_DMA_DONTWAIT (see ddi_dma_req(9S)) are taken to mean, respectively, wait until resources are available, or do not wait at all and do not schedule a callback.

arg Argument to be passed to a callback function, if such a function is specified.

lim A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

handlep Pointer to a DMA handle. See ddi_dma_setup(9F) for a discussion of handle.

DESCRIPTION

ddi_dma_buf_setup() is an interface to ddi_dma_setup(9F). It uses its arguments to construct an appropriate ddi_dma_req structure and calls ddi_dma_setup() with it.

RETURN VALUES

See ddi_dma_setup(9F) for the possible return values for this function.

CONTEXT

ddi_dma_buf_setup() can be called from user or interrupt context, except when waitfp is set to DDI_DMA_SLEEP, in which case it can be called from user context only.

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers 391
SEE ALSO: attributes(5), ddi_dma_addr_setup(9F), ddi_dma_free(9F),
ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), physio(9F), buf(9S),
ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
ddi_dma_burstsizes(9F)

NAME
ddi_dma_burstsizes – find out the allowed burst sizes for a DMA mapping

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_burstsizes(ddi_dma_handle_t handle);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
handle A DMA handle that was filled in by a successful call to ddi_dma_setup(9F).

DESCRIPTION
ddi_dma_burstsizes() returns the allowed burst sizes for a DMA mapping. This value is derived from the dlim_burstsizes member of the ddi_dma_lim_sparc(9S) structure, but it shows the allowable burst sizes after imposing on it the limitations of other device layers in addition to device's own limitations.

RETURN VALUES
ddi_dma_burstsizes() returns a binary encoded value of the allowable DMA burst sizes. See ddi_dma_lim_sparc(9S) for a discussion of DMA burst sizes.

CONTEXT This function can be called from user or interrupt context.

SEE ALSO ddi_dma_devalign(9F), ddi_dma_setup(9F), ddi_dma_lim_sparc(9S), ddi_dma_req(9S)

Writing Device Drivers
**NAME**
ddi_dma_coff – convert a DMA cookie to an offset within a DMA handle

**SYNOPSIS**
```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_coff(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep,
                  off_t *offp);
```

**INTERFACE LEVEL PARAMETERS**
- **handle**
  The `handle` filled in by a call to `ddi_dma_setup(9F)`.
- **cookiep**
  A pointer to a DMA cookie (see `ddi_dma_cookie(9S)`) that contains the appropriate address, length and bus type to be used in programming the DMA engine.
- **offp**
  A pointer to an offset to be filled in.

**DESCRIPTION**
`ddi_dma_coff()` converts the values in DMA cookie pointed to by `cookiep` to an offset (in bytes) from the beginning of the object that the DMA handle has mapped.

`ddi_dma_coff()` allows a driver to update a DMA cookie with values it reads from its device’s DMA engine after a transfer completes and convert that value into an offset into the object that is mapped for DMA.

**RETURN VALUES**
- **DDI_SUCCESS**
  Successfully filled in `offp`.
- **DDI_FAILURE**
  Failed to successfully fill in `offp`.

**CONTEXT**
`ddi_dma_coff()` can be called from user or interrupt context.

**SEE ALSO**
`ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S)`

*Writing Device Drivers*
ddi_dma_curwin(9F)

### NAME
ddi_dma_curwin – report current DMA window offset and size

### SYNOPSIS
```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_curwin(ddi_dma_handle_t handle, off_t *offp, uint_t *lenp);
```

### INTERFACE LEVEL
This interface is obsolete. ddi_dma_getwin(9F) should be used instead.

### PARAMETERS
- **handle**
  - The DMA handle filled in by a call to ddi_dma_setup(9F).
- **offp**
  - A pointer to a value which will be filled in with the current offset from the beginning of the object that is mapped for DMA.
- **lenp**
  - A pointer to a value which will be filled in with the size, in bytes, of the current window onto the object that is mapped for DMA.

### DESCRIPTION
`ddi_dma_curwin()` reports the current DMA window offset and size. If a DMA mapping allows partial mapping, that is if the `DDI_DMA_PARTIAL` flag in the `ddi_dma_req(9S)` structure is set, its current (effective) DMA window offset and size can be obtained by a call to `ddi_dma_curwin()`.

### RETURN VALUES
- **DDI_SUCCESS**
  - The current length and offset can be established.
- **DDI_FAILURE**
  - Otherwise.

### CONTEXT
`ddi_dma_curwin()` can be called from user or interrupt context.

### ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

### SEE ALSO
attributes(5), ddi_dma_getwin(9F), ddi_dma_movwin(9F), ddi_dma_setup(9F), ddi_dma_req(9S)

Writing Device Drivers

Kernel Functions for Drivers  395
ddi_dma_devalign(9F)

### NAME
ddi_dma_devalign – find DMA mapping alignment and minimum transfer size

### SYNOPSIS
```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_devalign(ddi_dma_handle_t handle, uint_t *alignment,
                      uint_t *minxfr);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **handle**
  The DMA handle filled in by a successful call to `ddi_dma_setup(9F)`.
- **alignment**
  A pointer to an unsigned integer to be filled in with the minimum required alignment for DMA. The alignment is guaranteed to be a power of two.
- **minxfr**
  A pointer to an unsigned integer to be filled in with the minimum effective transfer size (see `ddi_iomin(9F)`, `ddi_dma_lim_sparc(9S)`, and `ddi_dma_lim_IA(9S)`). This also is guaranteed to be a power of two.

### DESCRIPTION
ddi_dma_devalign() determines after a successful DMA mapping (see `ddi_dma_setup(9F)`) the minimum required data alignment and minimum DMA transfer size.

### RETURN VALUES
ddi_dma_devalign() returns:
- **DDI_SUCCESS**
  The `alignment` and `minxfr` values have been filled.
- **DDI_FAILURE**
  The handle was illegal.

### CONTEXT
ddi_dma_devalign() can be called from user or interrupt context.

### SEE ALSO
ddi_dma_setup(9F), ddi_iomin(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_IA(9S), ddi_dma_req(9S)

Writing Device Drivers
ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable, ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty, ddi_dmae_getlim, ddi_dmae_getattr — system DMA engine functions

```c
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);
int ddi_dmae_release(dev_info_t *dip, int chnl);
int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp, ddi_dmae_cookie_t *cookiep, int chnl);
int ddi_dmae_disable(dev_info_t *dip, int chnl);
int ddi_dmae_enable(dev_info_t *dip, int chnl);
int ddi_dmae_stop(dev_info_t *dip, int chnl);
int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);
int ddi_dmae_1stparty(dev_info_t *dip, int chnl);
int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);
int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

### PARAMETERS

- `dip` — A `dev_info` pointer that identifies the device.
- `chnl` — A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.
- `callback` — The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:
  - `DDI_DMA_SLEEP` — Wait until resources are available.
  - `DDI_DMA_DONTWAIT` — Do not wait until resources are available and do not schedule a callback.
- `arg` — Argument to be passed to the callback function, if specified.
- `dmaereqp` — A pointer to a DMA engine request structure. See `ddi_dmae_req(9S)`.
- `cookiep` — A pointer to a `ddi_dma_cookie(9S)` object, obtained from `ddi_dma_segtocookie(9F)`, which contains the address and count.
- `countp` — A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.
- `limitsp` — A pointer to a DMA limit structure. See `ddi_dma_lim_IA(9S)`.

Solaris DDI specific (Solaris DDI). The `ddi_dmae_getlim()` interface, described below, is obsolete. Use `ddi_dmae_getattr()`, also described below, to replace it.
There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**

If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

**Third-party DMA**

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of NULL for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S)`.

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver

Kernel Functions for Drivers 399
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by ddi_dmae_getlim(). After calling ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA engine's scatter/gather list to determine whether DMA chaining will be used. See ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on scatter/gather DMA.

**ddi_dmae_getattr**

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use ddi_dmae_getattr(). The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

**RETURN VALUES**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_SUCCESS</td>
<td>Upon success, for all of these routines.</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
<td>May be returned due to invalid arguments.</td>
</tr>
<tr>
<td>DDI_DMA_NORESOURCES</td>
<td>May be returned by ddi_dmae_alloc() if the requested resources are not available and the value of dmac_waitfp is not DDI_DMA_SLEEP.</td>
</tr>
</tbody>
</table>

**CONTEXT**

If ddi_dmae_alloc() is called from interrupt context, then its dmac_waitfp argument and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)
ddi_dmae(9F)

NAME

ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
    ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);

INTERFACE

Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described
below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

LEVEL

PARAMETERS

dip A dev_info pointer that identifies the device.

chnl A DMA channel number. On ISA or EISA buses this number must
    be 0, 1, 2, 3, 5, 6, or 7.

callback The address of a function to call back later if resources are not
    currently available. The following special function addresses may
    also be used:
    DDI_DMA_SLEEP Wait until resources are available.
    DDI_DMA_DONTWAIT Do not wait until resources are available and do not schedule a
                     callback.

arg Argument to be passed to the callback function, if specified.

dmaereqp A pointer to a DMA engine request structure. See
    ddi_dmae_req(9S).

cookiep A pointer to a ddi_dma_cookie(9S) object, obtained from
    ddi_dma_segtocookie(9F), which contains the address and
    count.

countp A pointer to an integer that will receive the count of the number of
    bytes not yet transferred upon completion of a DMA operation.

limitsp A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).
attrp  A pointer to a DMA attribute structure. See ddi_dma_attr(9S).

**DESCRIPTION**

There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**

If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from ddi_dma_segtocookie(9F). See ddi_dma_cookie(9S) for a description of a DMA cookie.

**Third-party DMA**

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except ddi_dmae_1stparty(), to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the ddi_dmae_1stparty() function may be used to perform whatever configuration is necessary to enable this mode.

**ddi_dmae_alloc()**

The ddi_dmae_alloc() function is used to acquire a DMA channel of the system DMA engine. ddi_dmae_alloc() allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using ddi_dmae_release() after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See detach(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of callback determines what action will be taken. If the value of callback is DDI_DMA_DONTWAIT, ddi_dmae_alloc() will return immediately. The value DDI_DMA_SLEEP will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, ddi_dmae_alloc() returns immediately, and when resources might have become available, the callback function is called (with the argument arg from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value DDI_DMA_CALLBACK_DONE. If it tries to allocate the channel but fails to do so, it must return the value DDI_DMA_CALLBACK_RUNOUT. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S).

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine's scatter/gather list to determine whether DMA chaining will be used. See `ddi_dmae_getattr(9S)` and `ddi_dmae_req(9S)` for additional information on scatter/gather DMA.

**ddi_dma_e_getattr**
The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F)`.

**RETURN VALUES**
- **DDI_SUCCESS** Upon success, for all of these routines.
- **DDI_FAILURE** May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES** May be returned by `ddi_dmae_alloc()` if the requested resources are not available and the value of `dmae_waitfp` is not `DDI_DMA_SLEEP`.

**CONTEXT**
If `ddi_dmae_alloc()` is called from interrupt context, then its `dmae_waitfp` argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`eisa(4)`, `isa(4)`, `attributes(5)`, `ddi_dma_buf_setup(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_nextseg(9F)`, `ddi_dma_nextwin(9F)`, `ddi_dma_segtocookie(9F)`, `ddi_dma_setup(9F)`, `ddi_dma_attr(9S)`, `ddi_dma_cookie(9S)`, `ddi_dma_lim_x86(9S)`, `ddi_dma_req(9S)`, `ddi_dmae_req(9S)`
### ddi_dmae_alloc(9F)

**NAME**
- ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
- ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
- ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

**SYNOPSIS**

```c
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);
int ddi_dmae_release(dev_info_t *dip, int chnl);
int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
                   ddi_dma_cookie_t *cookiep, int chnl);
int ddi_dmae_disable(dev_info_t *dip, int chnl);
int ddi_dmae_enable(dev_info_t *dip, int chnl);
int ddi_dmae_stop(dev_info_t *dip, int chnl);
int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);
int ddi_dmae_1stparty(dev_info_t *dip, int chnl);
int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);
int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

**INTERFACE LEVEL**
- Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

**PARAMETERS**

- **dip**
  - A dev_info pointer that identifies the device.

- **chnl**
  - A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.

- **callback**
  - The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:
    ```c
    DDI_DMA_SLEEP  // Wait until resources are available.
    DDI_DMA_DONTWAIT // Do not wait until resources are available and do not schedule a callback.
    ```

- **arg**
  - Argument to be passed to the callback function, if specified.

- **dmaereqp**
  - A pointer to a DMA engine request structure. See ddi_dmae_req(9S).

- **cookiep**
  - A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.

- **countp**
  - A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.

- **limitsp**
  - A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).
ddi_dmae_alloc(9F)

attrp  A pointer to a DMA attribute structure. See ddi_dma_attr(9S).

DESCRIPTION

There are three possible ways that a device can perform DMA engine functions:

Bus master DMA

If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from ddi_dma_segtocookie(9F). See ddi_dma_cookie(9S) for a description of a DMA cookie.

Third-party DMA

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except ddi_dmae_1stparty(), to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

First-party DMA

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the ddi_dmae_1stparty() function may be used to perform whatever configuration is necessary to enable this mode.

ddi_dmae_alloc()

The ddi_dmae_alloc() function is used to acquire a DMA channel of the system DMA engine. ddi_dmae_alloc() allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using ddi_dmae_release() after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See detach(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of callback determines what action will be taken. If the value of callback is DDI_DMA_DONTWAIT, ddi_dmae_alloc() will return immediately. The value DDI_DMA_SLEEP will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, ddi_dmae_alloc() returns immediately, and when resources might have become available, the callback function is called (with the argument arg) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value DDI_DMA_CALLBACK_DONE. If it tries to allocate the channel but fails to do so, it must return the value DDI_DMA_CALLBACK_RUNOUT. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of NULL for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S).

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)``.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See `ddi_dma_lim_IA(9S)` and `ddi_dmae_req(9S)` for additional information on scatter/gather DMA.

**ddi_dmae_getattr**

The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F)`.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>RETURN VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_SUCCESS</td>
<td>Upon success, for all of these routines.</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
<td>May be returned due to invalid arguments.</td>
</tr>
<tr>
<td>DDI_DMA_NORESOURCES</td>
<td>May be returned by <code>ddi_dmae_alloc()</code> if the requested resources are not available and the value of <code>dmae_waitfp</code> is not <code>DDI_DMA_SLEEP</code>.</td>
</tr>
</tbody>
</table>

**CONTEXT**

If `ddi_dmae_alloc()` is called from interrupt context, then its `dmae_waitfp` argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)
ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described
below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

PARAMETERS

dip
A dev_info pointer that identifies the device.

chnl
A DMA channel number. On ISA or EISA buses this number must
be 0, 1, 2, 3, 5, 6, or 7.

callback
The address of a function to call back later if resources are not
currently available. The following special function addresses may
also be used:

DDI_DMA_SLEEP Wait until resources are available.

DDI_DMA_DONTWAIT Do not wait until resources are
available and do not schedule a
callback.

arg
Argument to be passed to the callback function, if specified.

dmaereqp
A pointer to a DMA engine request structure. See
ddi_dmae_req(9S).

cookiep
A pointer to a ddi_dma_cookie(9S) object, obtained from
ddi_dma_segtocookie(9F), which contains the address and
count.

countp
A pointer to an integer that will receive the count of the number of
bytes not yet transferred upon completion of a DMA operation.

limitsp
A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).
There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**
If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

**Third-party DMA**
This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**
Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
The ddi_dmae_prog() function programs the DMA channel for a DMA transfer. The ddi_dmae_req structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to ddi_dmae_prog() may specify a value of NULL for dmaereqp if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing ddi_dmae_prog() a cookie obtained from ddi_dma_segtocookie(9F). Other DMA engine parameters are specified by the DMA engine request structure passed in through dmaereqp. The fields of that structure are documented in ddi_dmae_req(9S).

Before using ddi_dmae_prog(), you must allocate system DMA resources using DMA setup functions such as ddi_dma_buf_setup(9F). ddi_dma_segtocookie(9F) can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to ddi_dmae_prog().

The ddi_dmae_disable() function disables the DMA channel so that it no longer responds to a device's DMA service requests.

The ddi_dmae_enable() function enables the DMA channel for operation. This may be used to re-enable the channel after a call to ddi_dmae_disable(). The channel is automatically enabled after successful programming by ddi_dmae_prog().

The ddi_dmae_stop() function disables the channel and terminates any active operation.

The ddi_dmae_getcnt() function examines the count register of the DMA channel and sets *countp to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, ddi_dmae_1stparty() configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in ddi_dmae_1stparty() mode, the DMA channel must first be allocated using ddi_dmae_alloc() and then configured using ddi_dmae_1stparty(). The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from ddi_dma_segtocookie(9F).

Note that this function is obsolete. Use ddi_dmae_getattr(), described below, instead.

The ddi_dmae_getlim() function fills in the DMA limit structure, pointed to by limitsp, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use ddi_dmae_getlim(). The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see ddi_dma_nextseg(9F) and ddi_dma_nextwin(9F)). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by ddi_dmae_getlim(). After calling ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on scatter/gather DMA.

ddi_dmae_getattr

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use ddi_dmae_getattr(). The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

RETURN VALUES

DDI_SUCCESS	Upon success, for all of these routines.
DDI_FAILURE	May be returned due to invalid arguments.
DDI_DMA_NORESOURCES	May be returned by ddi_dmae_alloc() if the requested resources are not available and the value of dmae_waitfp is not DDI_DMA_SLEEP.

CONTEXT

If ddi_dmae_alloc() is called from interrupt context, then its dmae_waitfp argument and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all these routines may be called from user or interrupt context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

SEE ALSO

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)
### NAME
ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable, ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty, ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

### SYNOPSIS

```c
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);
int ddi_dmae_release(dev_info_t *dip, int chnl);
int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp, ddi_dma_cookie_t *cookiep, int chnl);
int ddi_dmae_disable(dev_info_t *dip, int chnl);
int ddi_dmae_enable(dev_info_t *dip, int chnl);
int ddi_dmae_stop(dev_info_t *dip, int chnl);
int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);
int ddi_dmae_1stparty(dev_info_t *dip, int chnl);
int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);
int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

### PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dip</code></td>
<td>A dev_info pointer that identifies the device.</td>
</tr>
<tr>
<td><code>chnl</code></td>
<td>A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.</td>
</tr>
<tr>
<td><code>callback</code></td>
<td>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_SLEEP – Wait until resources are available.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_DONTWAIT – Do not wait until resources are available and do not schedule a callback.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Argument to be passed to the callback function, if specified.</td>
</tr>
<tr>
<td><code>dmaereqp</code></td>
<td>A pointer to a DMA engine request structure. See ddi_dmae Req(9S).</td>
</tr>
<tr>
<td><code>cookiep</code></td>
<td>A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.</td>
</tr>
<tr>
<td><code>countp</code></td>
<td>A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.</td>
</tr>
<tr>
<td><code>limitsp</code></td>
<td>A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers  413
attrp      A pointer to a DMA attribute structure. See ddi_dma_attr(9S).

DESCRIPTION
There are three possible ways that a device can perform DMA engine functions:

Bus master DMA
If the device is capable of acting as a true bus master, then the driver should
program the device’s DMA registers directly and not make use of the DMA engine
functions described here. The driver should obtain the DMA address and count
from ddi_dma_segtocookie(9F). See ddi_dma_cookie(9S) for a description of
a DMA cookie.

Third-party DMA
This method uses the system DMA engine that is resident on the main system
board. In this model, the device cooperates with the system’s DMA engine to effect
the data transfers between the device and memory. The driver uses the functions
documented here, except ddi_dmae_1stparty(), to initialize and program the
DMA engine. For each DMA data transfer, the driver programs the DMA engine
and then gives the device a command to initiate the transfer in cooperation with
that engine.

First-party DMA
Using this method, the device uses its own DMA bus cycles, but requires a channel
from the system’s DMA engine. After allocating the DMA channel, the
ddi_dmae_1stparty() function may be used to perform whatever configuration
is necessary to enable this mode.

ddi_dmae_alloc()  The ddi_dmae_alloc() function is used to acquire a DMA channel of the system
DMA engine. ddi_dmae_alloc() allows only one device at a time to have a
particular DMA channel allocated. It must be called prior to any other system DMA
engine function on a channel. If the device allows the channel to be shared with other
devices, it must be freed using ddi_dmae_release() after completion of the DMA
operation. In any case, the channel must be released before the driver successfully
detaches. See detach(9E). No other driver may acquire the DMA channel until it is
released.

If the requested channel is not immediately available, the value of callback determines
what action will be taken. If the value of callback is DDI_DMA_DONTWAIT,
 ddii_dmae_alloc() will return immediately. The value DDI_DMA_SLEEP will cause
the thread to sleep and not return until the channel has been acquired. Any other
value is assumed to be a callback function address. In that case, ddi_dmae_alloc()
returns immediately, and when resources might have become available, the callback
function is called (with the argument arg) from interrupt context. When the callback
function is called, it should attempt to allocate the DMA channel again. If it succeeds
or no longer needs the channel, it must return the value DDI_DMA_CALLBACK_DONE.
If it tries to allocate the channel but fails to do so, it must return the value
DDI_DMA_CALLBACK_RUNOUT. In this case, the callback function is put back on a list
to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S).

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode. When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See `ddi_dma_lim_IA(9S)` and `ddi_dmae_req(9S)` for additional information on scatter/gather DMA.

**ddi_dmae_getattr**

The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F).

**RETURN VALUES**

- **DDI_SUCCESS**: Upon success, for all of these routines.
- **DDI_FAILURE**: May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES**: May be returned by `ddi_dmae_alloc()` if the requested resources are not available and the value of `dmae_waitfp` is not `DDI_DMA_SLEEP`.

**CONTEXT**

If `ddi_dmae_alloc()` is called from interrupt context, then its `dmae_waitfp` argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)`
### NAME

- ddi_dmae
- ddi_dmae_alloc
- ddi_dmae_release
- ddi_dmae_prog
- ddi_dmae_disable
- ddi_dmae_enable
- ddi_dmae_stop
- ddi_dmae_getcnt
- ddi_dmae_getlim
- ddi_dmae_getattr

Solaris DDI specific (Solaris DDI). The `ddi_dmae_getlim()` interface, described below, is obsolete. Use `ddi_dmae_getattr()`, also described below, to replace it.

### SYNOPSIS

```c
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp, ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

### INTERFACE LEVEL

Kernel Functions for Drivers

### PARAMETERS

- **dip**: A `dev_info_t` pointer that identifies the device.
- **chnl**: A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.
- **callback**: The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:
  - `DDI_DMA_SLEEP`: Wait until resources are available.
  - `DDI_DMA_DONTWAIT`: Do not wait until resources are available and do not schedule a callback.
- **arg**: Argument to be passed to the callback function, if specified.
- **dmaereqp**: A pointer to a DMA engine request structure. See `ddi_dmae_req(9S)`.
- **cookiep**: A pointer to a `ddi_dma_cookie(9S) object, obtained from `ddi_dma_segtocookie(9F)`, which contains the address and count.
- **countp**: A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.
- **limitsp**: A pointer to a DMA limit structure. See `ddi_dma_lim_IA(9S)`. 

---

**ddi_dmae_getattr(9F)**
ddi_dmae_getattr(9F)

attrp A pointer to a DMA attribute structure. See ddi_dma_attr(9S).

DESCRIPTION There are three possible ways that a device can perform DMA engine functions:

Bus master DMA

If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from ddi_dma_segtocookie(9F). See ddi_dma_cookie(9S) for a description of a DMA cookie.

Third-party DMA

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except ddi_dmae_1stparty(), to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

First-party DMA

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the ddi_dmae_1stparty() function may be used to perform whatever configuration is necessary to enable this mode.

ddi_dmae_alloc() The ddi_dmae_alloc() function is used to acquire a DMA channel of the system DMA engine. ddi_dmae_alloc() allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using ddi_dmae_release() after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See detach(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of callback determines what action will be taken. If the value of callback is DDI_DMA_DONTWAIT, ddi_dmae_alloc() will return immediately. The value DDI_DMA_SLEEP will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, ddi_dmae_alloc() returns immediately, and when resources might have become available, the callback function is called (with the argument arg) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value DDI_DMA_CALLBACK_DONE. If it tries to allocate the channel but fails to do so, it must return the value DDI_DMA_CALLBACK_RUNOUT. In this case, the callback function is put back on a list to be called again later.
**ddi_dmae_prog()**

The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of NULL for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S).

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

**ddi_dmae_disable()**

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

**ddi_dmae_enable()**

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

**ddi_dmae_stop()**

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

**ddi_dmae_getcnt()**

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

**ddi_dmae_1stparty()**

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`. Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

**ddi_dmae_getlim()**

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F) and ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver...
ddi_dmae_getattr(9F)

should further restrict the values in the structure members before passing them to the
DMA setup routines. The driver must not relax any of the restrictions embodied in the
structure after it is filled in by ddi_dmae_getlim(). After calling
ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA
engine’s scatter/gather list to determine whether DMA chaining will be used. See
ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on
scatter/gather DMA.

ddi_dmae_getattr

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to
by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that
perform their own bus mastering or use first-party DMA must create and initialize
their own DMA attribute structures; they should not use ddi_dmae_getattr(). The
DMA attribute structure must be passed to the DMA resource allocation functions to
provide the information necessary to break the DMA request into DMA windows and
DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

RETURN VALUES

DDI_SUCCESS Upon success, for all of these routines.
DDI_FAILURE May be returned due to invalid arguments.
DDI_DMA_NORESOURCES May be returned by ddi_dmae_alloc() if the
requested resources are not available and the value of
dmae_waitfp is not DDI_DMA_SLEEP.

CONTEXT

If ddi_dmae_alloc() is called from interrupt context, then its dmae_waitfp argument
and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all
these routines may be called from user or interrupt context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

SEE ALSO

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F),
ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F),
ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S),
ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S),
ddi_dmae_req(9S)
ddi_dmae_getcnt(9F)

NAME

ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
                  ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);

INTERFACE

Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described
below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

LEVEL

KERNEL FUNCTIONS FOR DRIVERS

PARAMETERS

dip       A dev_info pointer that identifies the device.
chnl      A DMA channel number. On ISA or EISA buses this number must
          be 0, 1, 2, 3, 5, 6, or 7.
callback   The address of a function to call back later if resources are not
          currently available. The following special function addresses may
          also be used:
          DDI_DMA_SLEEP     Wait until resources are available.
          DDI_DMA_DONTWAIT  Do not wait until resources are
                             available and do not schedule a
                             callback.
arg        Argument to be passed to the callback function, if specified.
dmaereqp   A pointer to a DMA engine request structure. See
           ddi_dmae_req(9S).
cookiep    A pointer to a ddi_dma_cookie(9S) object, obtained from
           ddi_dma_segtocookie(9F), which contains the address and
           count.
countp     A pointer to an integer that will receive the count of the number
           of bytes not yet transferred upon completion of a DMA operation.
limitsp    A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).
attrp A pointer to a DMA attribute structure. See ddi_dma_attr(9S).

DESCRIPTION
There are three possible ways that a device can perform DMA engine functions:

Bus master DMA
If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from ddi_dma_segtocookie(9F). See ddi_dma_cookie(9S) for a description of a DMA cookie.

Third-party DMA
This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except ddi_dmae_1stparty(), to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

First-party DMA
Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the ddi_dmae_1stparty() function may be used to perform whatever configuration is necessary to enable this mode.

ddi_dmae_alloc() The ddi_dmae_alloc() function is used to acquire a DMA channel of the system DMA engine. ddi_dmae_alloc() allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using ddi_dmae_release() after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See detach(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of callback determines what action will be taken. If the value of callback is DDI_DMA_DONTWAIT, ddi_dmae_alloc() will return immediately. The value DDI_DMA_SLEEP will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, ddi_dmae_alloc() returns immediately, and when resources might have become available, the callback function is called (with the argument arg) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value DDI_DMA_CALLBACK_DONE. If it tries to allocate the channel but fails to do so, it must return the value DDI_DMA_CALLBACK_RUNOUT. In this case, the callback function is put back on a list to be called again later.
ddi_dmae_prog() The ddi_dmae_prog() function programs the DMA channel for a DMA transfer. The ddi_dmae_req structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to ddi_dmae_prog() may specify a value of NULL for dmaereqp if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing ddi_dmae_prog() a cookie obtained from ddi_dma_segtocookie(9F). Other DMA engine parameters are specified by the DMA engine request structure passed in through dmaereqp. The fields of that structure are documented in ddi_dmae_req(9S).

Before using ddi_dmae_prog(), you must allocate system DMA resources using DMA setup functions such as ddi_dma_buf_setup(9F). ddi_dma_segtocookie(9F) can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to ddi_dmae_prog().

ddi_dmae_disable() The ddi_dmae_disable() function disables the DMA channel so that it no longer responds to a device’s DMA service requests.

ddi_dmae_enable() The ddi_dmae_enable() function enables the DMA channel for operation. This may be used to re-enable the channel after a call to ddi_dmae_disable(). The channel is automatically enabled after successful programming by ddi_dmae_prog().

ddi_dmae_stop() The ddi_dmae_stop() function disables the channel and terminates any active operation.

ddi_dmae_getcnt() The ddi_dmae_getcnt() function examines the count register of the DMA channel and sets *countp to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

ddi_dmae_1stparty() In the case of ISA and EISA buses, ddi_dmae_1stparty() configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in ddi_dmae_1stparty() mode, the DMA channel must first be allocated using ddi_dmae_alloc() and then configured using ddi_dmae_1stparty(). The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from ddi_dma_segtocookie(9F).

ddi_dmae_getlim() Note that this function is obsolete. Use ddi_dmae_getattr(), described below, instead.

The ddi_dmae_getlim() function fills in the DMA limit structure, pointed to by limitsp, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use ddi_dmae_getlim(). The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see ddi_dma_nextseg(9F) and ddi_dma_nextwin(9F)). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See `ddi_dma_lim_IA(9S)` and `ddi_dmae_req(9S)` for additional information on scatter/gather DMA.

**ddi_dmae_getattr**

The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F).

**RETURN VALUES**

- **DDI_SUCCESS**: Upon success, for all of these routines.
- **DDI_FAILURE**: May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES**: May be returned by `ddi_dmae_alloc()` if the requested resources are not available and the value of `dmae_waitfp` is not `DDI_DMA_SLEEP`.

**CONTEXT**

If `ddi_dmae_alloc()` is called from interrupt context, then its `dmae_waitfp` argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)`
### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI). The `ddi_dmae_getlim()` interface, described below, is obsolete. Use `ddi_dmae_getattr()`, also described below, to replace it.

### PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dip</code></td>
<td>A <code>dev_info</code> pointer that identifies the device.</td>
</tr>
<tr>
<td><code>chnl</code></td>
<td>A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.</td>
</tr>
<tr>
<td><code>callback</code></td>
<td>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_SLEEP: Wait until resources are available.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_DONTWAIT: Do not wait until resources are available and do not schedule a callback.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Argument to be passed to the callback function, if specified.</td>
</tr>
<tr>
<td><code>dmaereqp</code></td>
<td>A pointer to a DMA engine request structure. See <code>ddi_dmae_req(9S)</code>.</td>
</tr>
<tr>
<td><code>cookiep</code></td>
<td>A pointer to a <code>ddi_dma_cookie(9S)</code> object, obtained from <code>ddi_dmae_segtocookie(9F)</code>, which contains the address and count.</td>
</tr>
<tr>
<td><code>countp</code></td>
<td>A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.</td>
</tr>
<tr>
<td><code>limitsp</code></td>
<td>A pointer to a DMA limit structure. See <code>ddi_dmae_lim_IA(9S)</code>.</td>
</tr>
</tbody>
</table>
There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**

If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

**Third-party DMA**

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dma_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dma_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dma_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dma_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie()`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req()`. Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup()`. `ddi_dma_segtocookie()` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device's DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode. When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie()`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg()` and `ddi_dma_nextwin()`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver

Kernel Functions for Drivers 427
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by ddi_dmae_getlim(). After calling ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on scatter/gather DMA.

**ddi_dmae_getattr**

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use ddi_dmae_getattr(). The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

**RETURN VALUES**

- **DDI_SUCCESS** Upon success, for all of these routines.
- **DDI_FAILURE** May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES** May be returned by ddi_dmae_alloc() if the requested resources are not available and the value of dmae_waitfp is not DDI_DMA_SLEEP.

**CONTEXT**

If ddi_dmae_alloc() is called from interrupt context, then its dmae_waitfp argument and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all these routines may be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**
eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)
ddi_dmae_prog(9F)

NAME

ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable, ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty, ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp, ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);

INTERFACE

Level: Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip</td>
<td>A dev_info pointer that identifies the device.</td>
</tr>
<tr>
<td>chnl</td>
<td>A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.</td>
</tr>
<tr>
<td>callback</td>
<td>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:</td>
</tr>
<tr>
<td>arg</td>
<td>Argument to be passed to the callback function, if specified.</td>
</tr>
<tr>
<td>dmaereqp</td>
<td>A pointer to a DMA engine request structure. See ddi_dmae_reg(9S).</td>
</tr>
<tr>
<td>cookiep</td>
<td>A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.</td>
</tr>
<tr>
<td>countp</td>
<td>A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.</td>
</tr>
<tr>
<td>limitsp</td>
<td>A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).</td>
</tr>
</tbody>
</table>

Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.
There are three possible ways that a device can perform DMA engine functions:

Bus master DMA
If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

Third-party DMA
This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

First-party DMA
Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of NULL for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S)`.

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device's DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See `ddi_dma_lim_IA(9S)` and `ddi_dmae_req(9S)` for additional information on scatter/gather DMA.

### ddi_dmae_getattr

The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F)`.

### RETURN VALUES

- **DDI_SUCCESS** Upon success, for all of these routines.
- **DDI_FAILURE** May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES** May be returned by `ddi_dmae_alloc()` if the requested resources are not available and the value of `dmae_waitfp` is not `DDI_DMA_SLEEP`.

### CONTEXT

If `ddi_dmae_alloc()` is called from interrupt context, then its `dmae_waitfp` argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

### ATTRIBUTES

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

### SEE ALSO

- `eisa(4)`, `isa(4)`, `attributes(5)`, `ddi_dma_buf_setup(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_nextseg(9F)`, `ddi_dma_nextwin(9F)`, `ddi_dma_segtocookie(9F)`, `ddi_dma_setup(9F)`, `ddi_dma_attr(9S)`, `ddi_dma_cookie(9S)`, `ddi_dma_lim_x86(9S)`, `ddi_dma_req(9S)`, `ddi_dmae_req(9S)`
Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

ddi_dmae_release(9F)

NAME
ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

```
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);
int ddi_dmae_release(dev_info_t *dip, int chnl);
int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
                   ddi_dma_cookie_t *cookiep, int chnl);
int ddi_dmae_disable(dev_info_t *dip, int chnl);
int ddi_dmae_enable(dev_info_t *dip, int chnl);
int ddi_dmae_stop(dev_info_t *dip, int chnl);
int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);
int ddi_dmae_1stparty(dev_info_t *dip, int chnl);
int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);
int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

INTERFACE

Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.

LEVEL

PARAMETERS

dip A dev_info pointer that identifies the device.
chnl A DMA channel number. On ISA or EISA buses this number must
     be 0, 1, 2, 3, 5, 6, or 7.
callback The address of a function to call back later if resources are not
     currently available. The following special function addresses may
     also be used:
     DDI_DMA_SLEEP Wait until resources are available.
     DDI_DMA_DONTWAIT Do not wait until resources are
                      available and do not schedule a
                      callback.
arg Argument to be passed to the callback function, if specified.
dmaereqp A pointer to a DMA engine request structure. See
        ddi_dmae_req(9S).
cookiep A pointer to a ddi_dma_cookie(9S) object, obtained from
         ddi_dma_segtocookie(9F), which contains the address and
         count.
countp A pointer to an integer that will receive the count of the number of
        bytes not yet transferred upon completion of a DMA operation.
limitsp A pointer to a DMA limit structure. See ddi_dma_lim_IA(9S).

There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**

If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

**Third-party DMA**

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
ddi_dmae_prog() The *ddi_dmae_prog()* function programs the DMA channel for a DMA transfer. The
*ddi_dmae_req* structure contains all the information necessary to set up the channel,
except for the memory address and count. Once the channel has been programmed,
successive calls to *ddi_dmae_prog()* may specify a value of NULL for *dmaereqp* if no
changes to the programming are required other than the address and count values. It
disables the channel prior to setup, and enables the channel before returning. The
DMA address and count are specified by passing *ddi_dmae_prog()* a cookie
obtained from *ddi_dma_segtocookie*(9F). Other DMA engine parameters are
specified by the DMA engine request structure passed in through *dmaereqp*. The fields
of that structure are documented in *ddi_dmae_req*(9S).

Before using *ddi_dmae_prog()* you must allocate system DMA resources using
DMA setup functions such as *ddi_dma_buf_setup*(9F). *ddi_dma_segtocookie*(9F) can then be used to retrieve a cookie which contains the
address and count. Then this cookie is passed to *ddi_dmae_prog()*.

ddi_dmae_disable() The *ddi_dmae_disable()* function disables the DMA channel so that it no longer
responds to a device’s DMA service requests.

ddi_dmae_enable() The *ddi_dmae_enable()* function enables the DMA channel for operation. This may
be used to re-enable the channel after a call to *ddi_dmae_disable().* The channel is
automatically enabled after successful programming by *ddi_dmae_prog().*

ddi_dmae_stop() The *ddi_dmae_stop()* function disables the channel and terminates any active
operation.

ddi_dmae_getcnt() The *ddi_dmae_getcnt()* function examines the count register of the DMA channel
and sets *countp* to the number of bytes remaining to be transferred. The channel is
assumed to be stopped.

In the case of ISA and EISA buses, *ddi_dmae_1stparty()* configures a channel in
the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in *ddi_dmae_1stparty()* mode, the DMA channel must first be
allocated using *ddi_dmae_alloc()* and then configured using
*ddi_dmae_1stparty().* The driver then programs the device to perform the I/O,
including the necessary DMA address and count values obtained from
*ddi_dma_segtocookie*(9F).

Note that this function is obsolete. Use *ddi_dmae_getattr()* described below,
instead.

The *ddi_dmae_getlim()* function fills in the DMA limit structure, pointed to by
*limitsp*, with the DMA limits of the system DMA engine. Drivers for devices that
perform their own bus mastering or use first-party DMA must create and initialize
their own DMA limit structures; they should not use *ddi_dmae_getlim().* The
DMA limit structure must be passed to the DMA setup routines so that they will know
how to break the DMA request into windows and segments (see
*ddi_dma_nextseg*(9F) and *ddi_dma_nextwin*(9F)). If the device has any particular
restrictions on transfer size or granularity (such as the size of disk sector), the driver
should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by ddi_dmae_getlim(). After calling ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA engine’s scatter/gather list to determine whether DMA chaining will be used. See ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on scatter/gather DMA.

ddi_dmae_getattr

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use ddi_dmae_getattr(). The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

RETURN VALUES

DDI_SUCCESS Upon success, for all of these routines.
DDI_FAILURE May be returned due to invalid arguments.
DDI_DMA_NORESOURCES May be returned by ddi_dmae_alloc() if the requested resources are not available and the value of dmae_waitfp is not DDI_DMA_SLEEP.

CONTEXT

If ddi_dmae_alloc() is called from interrupt context, then its dmae_waitfp argument and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all these routines may be called from user or interrupt context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

SEE ALSO

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S), ddi_dmae_req(9S)
### NAME

`ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable, ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty, ddi_dmae_getlim, ddi_dmae_getattr` — system DMA engine functions

### SYNOPSIS

```c
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback)(caddr_t), caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp, ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI). The `ddi_dmae_getlim()` interface, described below, is obsolete. Use `ddi_dmae_getattr()`, also described below, to replace it.

### PARAMETERS

- **dip** — A `dev_info` pointer that identifies the device.
- **chnl** — A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.
- **callback** — The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:
  - `DDI_DMA_SLEEP` — Wait until resources are available.
  - `DDI_DMA_DONTWAIT` — Do not wait until resources are available and do not schedule a callback.
- **arg** — Argument to be passed to the callback function, if specified.
- **dmaereqp** — A pointer to a DMA engine request structure. See `ddi_dmae_req(9S)`.
- **cookiep** — A pointer to a `ddi_dma_cookie(9S)` object, obtained from `ddi_dma_segtocookie(9F)`, which contains the address and count.
- **countp** — A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.
- **limitsp** — A pointer to a DMA limit structure. See `ddi_dma_lim_IA(9S)`.
There are three possible ways that a device can perform DMA engine functions:

**Bus master DMA**

If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie(9F)`. See `ddi_dma_cookie(9S)` for a description of a DMA cookie.

**Third-party DMA**

This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system’s DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

**First-party DMA**

Using this method, the device uses its own DMA bus cycles, but requires a channel from the system’s DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach(9E)`. No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.
The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S).

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device's DMA service requests.

The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system’s DMA engine to operate in a “slave” (“cascade”) mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver...
should further restrict the values in the structure members before passing them to the
DMA setup routines. The driver must not relax any of the restrictions embodied in the
structure after it is filled in by ddi_dmae_getlim(). After calling
ddi_dmae_getlim(), a driver must examine, and possibly set, the size of the DMA
engine’s scatter/gather list to determine whether DMA chaining will be used. See
ddi_dma_lim_IA(9S) and ddi_dmae_req(9S) for additional information on
scatter/gather DMA.

**ddi_dmae_getattr**

The ddi_dmae_getattr() function fills in the DMA attribute structure, pointed to
by attrp, with the DMA attributes of the system DMA engine. Drivers for devices that
perform their own bus mastering or use first-party DMA must create and initialize
their own DMA attribute structures; they should not use ddi_dmae_getattr(). The
DMA attribute structure must be passed to the DMA resource allocation functions to
provide the information necessary to break the DMA request into DMA windows and
DMA cookies. See ddi_dma_nextcookie(9F) and ddi_dma_getwin(9F).

**RETURN VALUES**

- **DDI_SUCCESS** Upon success, for all of these routines.
- **DDI_FAILURE** May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES** May be returned by ddi_dmae_alloc() if the
  requested resources are not available and the value of
  dmae_waitfp is not DDI_DMA_SLEEP.

**CONTEXT**

If ddi_dmae_alloc() is called from interrupt context, then its dmae_waitfp argument
and the callback function must not have the value DDI_DMA_SLEEP. Otherwise, all
these routines may be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

eisa(4), isa(4), attributes(5), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F),
ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F), ddi_dma_nextwin(9F),
ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_attr(9S),
ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S),
ddi_dmae_req(9S)
NAME  
ddi_dma_free – release system DMA resources

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_free(ddi_dma_handle_t handle);

INTERFACE
This interface is obsolete. ddi_dma_free_handle(9F) should be used instead.

LEVEL
PARAMETERS
handle     The handle filled in by a call to ddi_dma_setup(9F).

DESCRIPTION
ddi_dma_free() releases system DMA resources set up by ddi_dma_setup(9F).
When a DMA transfer completes, the driver should free up system DMA resources
established by a call to ddi_dma_setup(9F). This is done by a call to
ddi_dma_free(). ddi_dma_free() does an implicit ddi_dma_sync(9F) for you
so any further synchronization steps are not necessary.

RETURN VALUES
ddi_dma_free() returns:
DDI_SUCCESS   Successfully released resources
DDI_FAILURE   Failed to free resources

CONTEXT
ddi_dma_free() can be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F),
(ddi_dma_free_handle(9F), ddi_dma_htoc(9F), ddi_dma_sync(9F),
(ddi_dma_req(9S)

Writing Device Drivers
### ddi_dma_free_handle(9F)

**NAME**  
ddi_dma_free_handle – free DMA handle

**SYNOPSIS**  
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_dma_free_handle(ddi_dma_handle_t *handle);
```

**PARAMETERS**  
- `handle`  
  A pointer to the DMA handle previously allocated by a call to `ddi_dma_alloc_handle(9F)`.

**INTERFACE LEVEL**  
Solaris DDI specific (Solaris DDI).

**DESCRIPTION**  
`ddi_dma_free_handle()` destroys the DMA handle pointed to by `handle`. Any further references to the DMA handle will have undefined results. Note that `ddi_dma_unbind_handle(9F)` must be called prior to `ddi_dma_free_handle()` to free any resources the system may be caching on the handle.

**CONTEXT**  
`ddi_dma_free_handle()` can be called from user, kernel, or interrupt context.

**SEE ALSO**  
`ddi_dma_alloc_handle(9F), ddi_dma_unbind_handle(9F)`

*Writing Device Drivers*
**ddi_dma_get_attr(9F)**

**NAME**
ddi_dma_get_attr – get the device DMA attribute structure from a DMA handle

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_get_attr(ddi_dma_handle_t handle, ddi_dma_attr_t *attrp);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI)

**PARAMETERS**
- **handle**
  The handle filled in by a call to ddi_dma_alloc_handle(9F).
- **attrp**
  Pointer to a buffer suitable for holding a DMA attribute structure. See ddi_dma_attr(9S).

**DESCRIPTION**
`ddi_dma_get_attr()` is used to get a `ddi_dma_attr(9S)` structure. This structure describes the attributes of the DMA data path to which any memory object bound to the given handle will be subject.

**RETURN VALUES**
- **DDI_SUCCESS**
  Successfully passed back attribute structure in buffer pointed to by `attrp`.
- **DDI_DMA_BADATTR**
  A valid attribute structure could not be passed back.

**CONTEXT**
`ddi_dma_get_attr()` can be called from any context.

**SEE ALSO**
`ddi_dma_alloc_handle(9F), ddi_dma_attr(9S)`
ddi_dma_getwin(9F)

NAME
   ddi_dma_getwin – activate a new DMA window

SYNOPSIS
   #include <sys/ddi.h>
   #include <sys/sunddi.h>

   int ddi_dma_getwin(ddi_dma_handle_t handle, uint_t win, off_t *offp,
                      size_t *lenp, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

INTERFACE
   Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
   handle        The DMA handle previously allocated by a call to
                  ddi_dma_alloc_handle(9F).
   win           Number of the window to activate.
   offp          Pointer to an offset. Upon a successful return, offp will contain
                  the new offset indicating the beginning of the window within the
                  object.
   lenp          Upon a successful return, lenp will contain the size, in bytes, of
                  the current window.
   cookiep       A pointer to the first ddi_dma_cookie(9S) structure.
   ccountp       Upon a successful return, ccountp will contain the number of
                  cookies for this DMA window.

DESCRIPTION
   ddi_dma_getwin() activates a new DMA window. If a DMA resource allocation
   request returns DDI_DMA_PARTIAL_MAP indicating that resources for less than the
   entire object were allocated, the current DMA window can be changed by a call to
   ddi_dma_getwin() again.

   The caller must first determine the number of DMA windows, N, using
   ddi_dma_numwin(9F). ddi_dma_getwin() takes a DMA window number from the
   range [0..N-1] as the parameter win and makes it the current DMA window.

   ddi_dma_getwin() fills in the first DMA cookie pointed to by cookiep with the
   appropriate address, length, and bus type. *ccountp is set to the number of DMA
   cookies representing this DMA object. Subsequent DMA cookies must be retrieved
   using ddi_dma_nextcookie(9F).

   ddi_dma_getwin() takes care of underlying resource synchronizations required to
   shift the window. However accessing the data prior to or after moving the window
   requires further synchronization steps using ddi_dma_sync(9F).

   ddi_dma_getwin() is normally called from an interrupt routine. The first invocation
   of the DMA engine is done from the driver. All subsequent invocations of the DMA
   engine are done from the interrupt routine. The interrupt routine checks to see if
   the request has been completed. If it has, the interrupt routine returns without invoking
   another DMA transfer. Otherwise, it calls ddi_dma_getwin() to shift the current
   window and start another DMA transfer.

RETURN VALUES
   ddi_dma_getwin() returns:

444  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 15 Nov 1996
ddi_dma_getwin(9F)

| DDI_SUCCESS | Resources for the specified DMA window are allocated. |
| DDI_FAILURE | win is not a valid window index. |

**CONTEXT**

ddi_dma_getwin() can be called from user, kernel, or interrupt context.

**SEE ALSO**

ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_nextcookie(9F),
ddi_dma_numwin(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F),
ddi_dma_cookie(9S)

*Writing Device Drivers*
ddi_dma_htoc(9F)

NAME      ddi_dma_htoc – convert a DMA handle to a DMA address cookie

SYNOPSIS  #include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_htoc(ddi_dma_handle_t handle, off_t off,  
                  ddi_dma_cookie_t *cookiep);

INTERFACE LEVEL
This interface is obsolete. ddi_dma_addr_bind_handle(9F) or  
ddi_dma_buf_bind_handle(9F) should be used instead.

PARAMETERS
handle      The handle filled in by a call to ddi_dma_setup(9F).
off         An offset into the object that handle maps.
cookie      A pointer to a ddi_dma_cookie(9S) structure.

DESCRIPTION
ddi_dma_htoc() takes a DMA handle (established by ddi_dma_setup(9F)), and  
fills in the cookie pointed to by cookiep with the appropriate address, length, and bus  
type to be used to program the DMA engine.

RETURN VALUES
ddi_dma_htoc() returns:
DDI_SUCCESS   Successfully filled in the cookie pointed to by cookiep.
DDI_FAILURE   Failed to successfully fill in the cookie.

CONTEXT
ddi_dma_htoc() can be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), ddi_dma_addr_bind_handle(9F), ddi_dma_addr_setup(9F),  
ddi_dma_buf_bind_handle(9F), ddi_dma_buf_setup(9F), ddi_dma_setup(9F),  
ddi_dma_sync(9F), ddi_dma_cookie(9S)

Writing Device Drivers
ddi_dma_mem_alloc – allocate memory for DMA transfer

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
                      ddi_device_acc_attr_t *accattrp, uint_t flags,
                      int (*waitfp)(caddr_t), caddr_t arg,
                      caddr_t *kaddrp, size_t *real_length,
                      ddi_acc_handle_t *handlep);

Solaris DDI specific (Solaris DDI).

**INTERFACE LEVEL PARAMETERS**

- **handle**: The DMA handle previously allocated by a call to `ddi_dma_alloc_handle(9F)`.
- **length**: The length in bytes of the desired allocation.
- **accattrp**: Pointer to a device access attribute structure of this device (see `ddi_device_acc_attr(9S)`).
- **flags**: Data transfer mode flags. Possible values are:
  - `DDI_DMA_STREAMING`: Sequential, unidirectional, block-sized, and block-aligned transfers.
  - `DDI_DMA_CONSISTENT`: Nonsequential transfers of small objects.
- **waitfp**: The address of a function to call back later if resources are not available now. The callback function indicates how a caller wants to handle the possibility of resources not being available. If callback is set to `DDI_DMA_DONTWAIT`, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If callback is set to `DDI_DMA_SLEEP`, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set and a DMA resource allocation fails, this value is assumed to be the address of a function to be called when resources become available. When the specified function is called, `arg` is passed to it as an argument. The specified callback function must return either `DDI_DMA_CALLBACK_RUNOUT` or `DDI_DMA_CALLBACK_DONE`.
  - `DDI_DMA_CALLBACK_RUNOUT` indicates that the callback function attempted to allocate DMA resources but failed. In this case, the callback function is put back on a list to be called again later.
  - `DDI_DMA_CALLBACK_DONE` indicates that either the allocation of DMA resources was successful or the driver no longer wishes to retry. The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are available.
The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.

**arg**
Argument to be passed to the callback function, if such a function is specified.

**kaddrp**
On successful return, `kaddrp` points to the allocated memory.

**real_length**
The amount of memory, in bytes, allocated. Alignment and padding requirements may require `ddi_dma_mem_alloc()` to allocate more memory than requested in `length`.

**handlep**
Pointer to a data access handle.

**DESCRIPTION**

`ddi_dma_mem_alloc()` allocates memory for DMA transfers to or from a device. The allocation will obey the alignment, padding constraints and device granularity as specified by the DMA attributes (see `ddi_dma_attr(9S)`) passed to `ddi_dma_alloc_handle(9F)` and the more restrictive attributes imposed by the system.

`flags` should be set to `DDI_DMA_STREAMING` if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the `minxfer` and `burstsizes` fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)` will be used to allocate the most effective hardware support for large transfers. For example, if an I/O transfer can be sped up by using an I/O cache, which has a minimum transfer of one cache line, `ddi_dma_mem_alloc()` will align the memory at a cache line boundary and it will round up `real_length` to a multiple of the cache line size.

`flags` should be set to `DDI_DMA_CONSISTENT` if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using `DDI_DMA_CONSISTENT`.

The device access attributes are specified in the location pointed by the `accattrp` argument (see `ddi_device_acc_attr(9S)`).

The data access handle is returned in `handlep`. `handlep` is opaque – drivers may not attempt to interpret its value. To access the data content, the driver must invoke `ddi_get8(9F)` or `ddi_put8(9F)` (depending on the data transfer direction) with the data access handle.

DMA resources must be established before performing a DMA transfer by passing `kaddrp` and `real_length` as returned from `ddi_dma_mem_alloc()` and the flag `DDI_DMA_STREAMING` or `DDI_DMA_CONSISTENT` to `ddi_dma_addr_bind_handle(9F)`. In addition, to ensure the consistency of a memory object shared between the CPU and the device after a DMA transfer, explicit synchronization steps using `ddi_dma_sync(9F)` or `ddi_dma_unbind_handle(9F)` are required.
ddi_dma_mem_alloc() returns:

DDI_SUCCESS  Memory successfully allocated.
DDI_FAILURE   Memory allocation failed.

CONTEXT      ddi_dma_mem_alloc() can be called from user or interrupt context, except when
              waitfp is set to DDI_DMA_SLEEP, in which case it can be called from user context only.

SEE ALSO     ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
              ddi_dma_mem_free(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F),
              ddi_get8(9F), ddi_put8(9F), ddi_device_acc_attr(9S), ddi_dma_attr(9S)

Writing Device Drivers

WARNINGS     If DDI_NEVERSWAP_ACC is specified, memory can be used for any purpose; but if
              either endian mode is specified, you must use ddi_get/put* and never anything
              else.
ddi_dma_mem_free(9F)

NAME  ddi_dma_mem_free – free previously allocated memory

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_dma_mem_free(ddi_acc_handle_t *handlep);

PARAMETERS  
handlep  Pointer to the data access handle previously allocated by a call to
ddi_dma_mem_alloc(9F).

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

DESCRIPTION  ddi_dma_mem_free() deallocates the memory acquired by
ddi_dma_mem_alloc(9F). In addition, it destroys the data access handle handlep
associated with the memory.

CONTEXT  ddi_dma_mem_free() can be called from user, kernel, or interrupt context.

SEE ALSO  ddi_dma_mem_alloc(9F)

Writing Device Drivers
ddi_dma_movwin(9F)

### NAME

ddi_dma_movwin – shift current DMA window

### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_movwin(ddi_dma_handle_t handle, off_t *offp, uint_t *
lenp, ddi_dma_cookie_t *cookiep);
```

### INTERFACE LEVEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>The DMA handle filled in by a call to ddi_dma_setup(9F).</td>
</tr>
<tr>
<td>offp</td>
<td>A pointer to an offset to set the DMA window to. Upon a successful return, it will be filled in with the new offset from the beginning of the object resources are allocated for.</td>
</tr>
<tr>
<td>lenp</td>
<td>A pointer to a value which must either be the current size of the DMA window (as known from a call to ddi_dma_curwin(9F) or from a previous call to ddi_dma_movwin()). Upon a successful return, it will be filled in with the size, in bytes, of the current window.</td>
</tr>
<tr>
<td>cookiep</td>
<td>A pointer to a DMA cookie (see ddi_dma_cookie(9S)). Upon a successful return, cookiep is filled in just as if an implicit ddi_dma_htoc(9F) had been made.</td>
</tr>
</tbody>
</table>

### DESCRIPTION

ddi_dma_movwin() shifts the current DMA window. If a DMA request allows the system to allocate resources for less than the entire object by setting the DDI_DMA_PARTIAL flag in the ddi_dma_req(9S) structure, the current DMA window can be shifted by a call to ddi_dma_movwin().

The caller must first determine the current DMA window size by a call to ddi_dma_curwin(9F). Using the current offset and size of the window thus retrieved, the caller of ddi_dma_movwin() may change the window onto the object by changing the offset by a value which is some multiple of the size of the DMA window.

ddi_dma_movwin() takes care of underlying resource synchronizations required to shift the window. However, if you want to access the data prior to or after moving the window, further synchronizations using ddi_dma_sync(9F) are required.

This function is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, it returns without invoking another DMA transfer. Otherwise it calls ddi_dma_movwin() to shift the current window and starts another DMA transfer.

### RETURN VALUES

ddi_dma_movwin() returns:

- **DDI_SUCCESS** The current length and offset are legal and have been set.
- **DDI_FAILURE** Otherwise.

Kernel Functions for Drivers 451
ddi_dma_movwin(9F)

CONTEXT  ddi_dma_movwin() can be called from user or interrupt context.

ATTRIBUTES  See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5), ddi_dma_curwin(9F), ddi_dma_getwin(9F), ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S), ddi_dma_req(9S)

Writing Device Drivers

WARNINGS  The caller must guarantee that the resources used by the object are inactive prior to calling this function.
**NAME**
`ddi_dma_nextcookie` – retrieve subsequent DMA cookie

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_dma_nextcookie(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep);
```

**PARAMETERS**
- `handle` The handle previously allocated by a call to `ddi_dma_alloc_handle(9F).`
- `cookiep` A pointer to a `ddi_dma_cookie(9S)` structure.

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**DESCRIPTION**
`ddi_dma_nextcookie()` retrieves subsequent DMA cookies for a DMA object.

`ddi_dma_nextcookie()` fills in the `ddi_dma_cookie(9S)` structure pointed to by `cookiep`. The `ddi_dma_cookie(9S)` structure must be allocated prior to calling `ddi_dma_nextcookie()`.

The DMA cookie count returned by `ddi_dma_buf_bind_handle(9F)`, `ddi_dma_addr_bind_handle(9F)`, or `ddi_dma_getwin(9F)` indicates the number of DMA cookies a DMA object consists of. If the resulting cookie count, \( N \), is larger than 1, `ddi_dma_nextcookie()` must be called \( N-1 \) times to retrieve all DMA cookies.

**CONTEXT**
`ddi_dma_nextcookie()` can be called from user, kernel, or interrupt context.

**EXAMPLES**

**EXAMPLE 1** process a scatter-gather list of I/O requests

This example demonstrates the use of `ddi_dma_nextcookie()` to process a scatter-gather list of I/O requests.

```c
/* setup scatter-gather list with multiple DMA cookies */

ddi_dma_cookie_t dmacookie;
uint_t ccount;
...

status = ddi_dma_buf_bind_handle(handle, bp, DDI_DMA_READ,
    NULL, NULL, &dmacookie, &ccount);
if (status == DDI_DMA_MAPPED) {
    /* program DMA engine with first cookie */
    while (--ccount > 0) {
        ddi_dma_nextcookie(handle, &dmacookie);
        /* program DMA engine with next cookie */
    }
}
...
```
ddi_dma_nextcookie(9F)

**EXAMPLE 1** process a scatter-gather list of I/O requests  (Continued)

**SEE ALSO**
- ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
- ddi_dma_buf_bind_handle(9F), ddi_dma_unbind_handle(9F),
- ddi_dma_cookie(9S)

*Writing Device Drivers*
# ddi_dma_nextseg

- **NAME**: ddi_dma_nextseg - get next DMA segment
- **SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_nextseg(ddi_dma_win_t win, ddi_dma_seg_t seg, ddi_dma_seg_t *nseg);
```
- **INTERFACE LEVEL PARAMETERS**

This interface is obsolete. ddi_dma_nextcookie(9F) should be used instead.

- **DESCRIPTION**

`ddi_dma_nextseg()` gets the next DMA segment within the specified window `win`. If the current segment is `NULL`, the first DMA segment within the window is returned.

A DMA segment is always required for a DMA window. A DMA segment is a contiguous portion of a DMA window (see `ddi_dma_nextwin(9F)`) which is entirely addressable by the device for a data transfer operation.

An example where multiple DMA segments are allocated is where the system does not contain DVMA capabilities and the object may be non-contiguous. In this example the object will be broken into smaller contiguous DMA segments. Another example is where the device has an upper limit on its transfer size (for example an 8-bit address register) and has expressed this in the DMA limit structure (see `ddi_dma_lim_sparc(9S)` or `ddi_dma_lim_x86(9S)`). In this example the object will be broken into smaller addressable DMA segments.

- **RETURN VALUES**

`ddi_dma_nextseg()` returns:

- `DDI_SUCCESS` - Successfully filled in the next segment pointer.
- `DDI_DMA_DONE` - There is no next segment. The current segment is the final segment within the specified window.
- `DDI_DMA_STALE` - `win` does not refer to the currently active window.

- **CONTEXT**

`ddi_dma_nextseg()` can be called from user or interrupt context.

- **EXAMPLES**

For an example, see `ddi_dma_segtocookie(9F)`.

- **ATTRIBUTES**

See `attributes(5)` for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>
SEE ALSO attributes(5), ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F),
ddi_dma_nextcookie(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F),
ddi_dma_sync(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_IA(9S),
ddi_dma_req(9S)

Writing Device Drivers
ddi_dma_nextwin() — get next DMA window

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_nextwin(ddi_dma_handle_t handle, ddi_dma_win_t win,
                     ddi_dma_win_t *nwin);

This interface is obsolete. ddi_dma_getwin() should be used instead.

handle
A DMA handle.

win
The current DMA window or NULL.

nwin
A pointer to the next DMA window to be filled in. If win is NULL, a pointer to the first window within the object is returned.

ddi_dma_nextwin() shifts the current DMA window win within the object referred to by handle to the next DMA window nwin. If the current window is NULL, the first window within the object is returned. A DMA window is a portion of a DMA object or might be the entire object. A DMA window has system resources allocated to it and is prepared to accept data transfers. Examples of system resources are DVMA mapping resources and intermediate transfer buffer resources.

All DMA objects require a window. If the DMA window represents the whole DMA object it has system resources allocated for the entire data transfer. However, if the system is unable to setup the entire DMA object due to system resource limitations, the driver writer may allow the system to allocate system resources for less than the entire DMA object. This can be accomplished by specifying the DDI_DMA_PARTIAL flag as a parameter to ddi_dma_buf_setup() or ddi_dma_addr_setup() or as part of a ddi_dma_req() structure in a call to ddi_dma_setup().

Only the window that has resources allocated is valid per object at any one time. The currently valid window is the one that was most recently returned from ddi_dma_nextwin(). Furthermore, because a call to ddi_dma_nextwin() will reallocate system resources to the new window, the previous window will become invalid. It is a severe error to call ddi_dma_nextwin() before any transfers into the current window are complete.

ddi_dma_nextwin() takes care of underlying memory synchronizations required to shift the window. However, if you want to access the data before or after moving the window, further synchronizations using ddi_dma_sync() are required.

ddi_dma_nextwin() returns:

DDI_SUCCESS Successfully filled in the next window pointer.

DDI_DMA_DONE There is no next window. The current window is the final window within the specified object.

DDI_DMA_STALE win does not refer to the currently active window.

ddi_dma_nextwin() can be called from user or interrupt context.
For an example see `ddi_dma_segtocookie(9F)`.

See `attributes(5)` for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`attributes(5), ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_getwin(9F), ddi_dma_nextseg(9F), ddi_dma_segtocookie(9F), ddi_dma_sync(9F), ddi_dma_req(9S)`

*Writing Device Drivers*
NAME

ddi_dma_numwin – retrieve number of DMA windows

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_numwin(ddi_dma_handle_t handle, uint_t *nwinp);

PARAMETERS

handle The DMA handle previously allocated by a call to
ddi_dma_alloc_handle(9F).

nwinp Upon a successful return, nwinp will contain the number of DMA
windows for this object.

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_numwin() returns the number of DMA windows for a DMA object if
partial resource allocation was permitted.

RETURN VALUES

ddi_dma_numwin() returns:

DDI_SUCCESS Successfully filled in the number of DMA windows.

DDI_FAILURE DMA windows are not activated.

CONTEXT

ddi_dma_numwin() can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_unbind_handle(9F)

Writing Device Drivers
ddi_dma_segtocookie(9F)

NAME

ddi_dma_segtocookie – convert a DMA segment to a DMA address cookie

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_segtocookie(ddi_dma_seg_t seg, off_t *offp, off_t *lenp,
                        ddi_dma_cookie_t *cookiep);

INTERFACE LEVEL

PARAMETERS

This interface is obsolete. ddi_dma_nextcookie(9F) should be used instead.

seg

A DMA segment.

offp

A pointer to an off_t. Upon a successful return, it is filled in with
the offset. This segment is addressing within the object.

lenp

The byte length. This segment is addressing within the object.

cookiep

A pointer to a DMA cookie (see ddi_dma_cookie(9S)).

DESCRIPTION

ddi_dma_segtocookie() takes a DMA segment and fills in the cookie pointed to
by cookiep with the appropriate address, length, and bus type to be used to program
the DMA engine. ddi_dma_segtocookie() also fills in *offp and *lenp, which
specify the range within the object.

RETURN VALUES

ddi_dma_segtocookie() returns:

DDI_SUCCESS Successfully filled in all values.

DDI_FAILURE Failed to successfully fill in all values.

CONTEXT

ddi_dma_segtocookie() can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1 ddi_dma_segtocookie() example

for (win = NULL; (retw = ddi_dma_nextwin(handle, win, &nwin)) !=
    DDI_DMA_DONE; win = nwin) {
    if (retw != DDI_SUCCESS) {
        /* do error handling */
    } else {
        for (seg = NULL; (rets = ddi_dma_nextseg(nwin, seg, &nseg)) !=
            DDI_DMA_DONE; seg = nseg) {
            if (rets != DDI_SUCCESS) {
                /* do error handling */
            } else {
                ddi_dma_segtocookie(nseg, &off, &len, &cookie);
                /* program DMA engine */
            }
        }
        /* program DMA engine */
    }
}
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), ddi_dma_nextcookie(9F), ddi_dma_nextseg(9F),
ddi_dma_nextwin(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S)

*Writing Device Drivers*
ddi_dma_set_sbus64(9F)

NAME ddi_dma_set_sbus64 – allow 64-bit transfers on SBus

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_set_sbus64(ddi_dma_handle_t handle, uint_t burstsizes);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS

handle The handle filled in by a call to ddi_dma_alloc_handle(9F).

burstsizes The possible burst sizes the device’s DMA engine can accept in
32-bit mode.

DESCRIPTION ddi_dma_set_sbus64() informs the system that the device wishes to perform
64-bit data transfers on the SBus. The driver must first allocate a DMA handle using
ddi_dma_alloc_handle(9F) with a ddi_dma_attr(9S) structure describing the
DMA attributes for a 32-bit transfer mode.

burstsizes describes the possible burst sizes the device’s DMA engine can accept in
64-bit mode. It may be distinct from the burst sizes for 32-bit mode set in the
ddi_dma_attr(9S) structure. The system will activate 64-bit SBus transfers if the
SBus supports them. Otherwise, the SBus will operate in 32-bit mode.

After DMA resources have been allocated (see ddi_dma_addr_bind_handle(9F) or
ddi_dma_buf_bind_handle(9F)), the driver should retrieve the available burst
sizes by calling ddi_dma_burstsizes(9F). This function will return the burst sizes
in 64-bit mode if the system was able to activate 64-bit transfers. Otherwise burst
sizes will be returned in 32-bit mode.

RETURN VALUES ddi_dma_set_sbus64() returns:

DDI_SUCCESS Successfully set the SBus to 64-bit mode.

DDI_FAILURE 64-bit mode could not be set.

CONTEXT ddi_dma_set_sbus64() can be called from user, kernel, or interrupt context.

ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>SBus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_burstsizes(9F), ddi_dma_attr(9S)

NOTES 64-bit SBus mode is activated on a per SBus slot basis. If there are multiple SBus cards
in one slot, they all must operate in 64-bit mode or they all must operate in 32-bit
mode.
ddi_dma_setup(9F)

NAME
ddi_dma_setup — setup DMA resources

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_setup(dev_info_t *dip, ddi_dma_req_t *dmareqp,
                   ddi_dma_handle_t *handlep);

INTERFACE LEVEL
This interface is obsolete. The functions ddi_dma_addr_bind_handle(9F),
ddi_dma_alloc_handle(9F), ddi_dma_buf_bind_handle(9F),
ddi_dma_free_handle(9F), and ddi_dma_unbind_handle(9F) should be used
instead.

PARAMETERS
dip A pointer to the device’s dev_info structure.
dmareqp A pointer to a DMA request structure (see ddi_dma_req(9S)).
handlep A pointer to a DMA handle to be filled in. See below for a
discussion of a handle. If handlep is NULL, the call to
ddi_dma_setup() is considered an advisory call, in which case
no resources are allocated, but a value indicating the legality and
the feasibility of the request is returned.

DESCRIPTION
ddi_dma_setup() allocates resources for a memory object such that a device can
perform DMA to or from that object.

A call to ddi_dma_setup() informs the system that device referred to by dip wishes
to perform DMA to or from a memory object. The memory object, the device’s DMA
capabilities, the device driver’s policy on whether to wait for resources, are all
specified in the ddi_dma_req structure pointed to by dmareqp.

A successful call to ddi_dma_setup() fills in the value pointed to by handlep. This is
an opaque object called a DMA handle. This handle is then used in subsequent DMA
calls, until ddi_dma_free(9F) is called.

Again a DMA handle is opaque—drivers may not attempt to interpret its value. When
a driver wants to enable its DMA engine, it must retrieve the appropriate address to
supply to its DMA engine using a call to ddi_dma_htoc(9F), which takes a pointer to
a DMA handle and returns the appropriate DMA address.

When DMA transfer completes, the driver should free up the the allocated DMA
resources by calling ddi_dma_free().

RETURN VALUES
ddi_dma_setup() returns:

DDI_DMA_MAPPED Successfully allocated resources for the object. In the
case of an advisory call, this indicates that the request is legal.

DDI_DMA_PARTIAL_MAP Successfully allocated resources for a part of the object.
This is acceptable when partial transfers are allowed.
using a flag setting in the ddi_dma_req structure (see ddi_dma_req(9S) and ddi_dma_movwin(9F)).

**DDI_DMA_NORESOURCES**  When no resources are available.

**DDI_DMA_NOMAPPING**  The object cannot be reached by the device requesting the resources.

**DDI_DMA_TOOBIG**  The object is too big and exceeds the available resources. The maximum size varies depending on machine and configuration.

**CONTEXT**  ddi_dma_setup() can be called from user or interrupt context, except when the dmar_fp member of the ddi_dma_req structure pointed to by dmareqp is set to DDI_DMA_SLEEP, in which case it can be called from user context only.

**ATTRIBUTES**  See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**  attributes(5), ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_free_handle(9F),
ddi_dma_unbind_handle(9F), ddi_dma_addr_setup(9F),
ddi_dma_buf_setup(9F), ddi_dma_free(9F), ddi_dma_htoc(9F),
ddi_dma_movwin(9F), ddi_dma_sync(9F), ddi_dma_req(9S)

**Writing Device Drivers**

**NOTES**  The construction of the ddi_dma_req structure is complicated. Use of the provided interface functions such as ddi_dma_buf_setup(9F) simplifies this task.
ddi_dma_sync() is used to selectively synchronize either a DMA device’s or a CPU’s view of a memory object that has DMA resources allocated for I/O. This may involve operations such as flashes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When resources are allocated for DMA using ddi_dma_addr_bind_handle() or ddi_dma_buf_bind_handle(), an implicit ddi_dma_sync() is done. When DMA resources are deallocated using ddi_dma_unbind_handle(), an implicit ddi_dma_sync() is done. However, at any time between DMA resource allocation and deallocation, if the memory object has been modified by either the DMA device or a CPU and you wish to ensure that the change is noticed by the party that did not do the modifying, a call to ddi_dma_sync() is required. This is true independent of any attributes of the memory object including, but not limited to, whether or not the memory was allocated for consistent mode I/O (see ddi_dma_mem_alloc(9F)) or whether or not DMA resources have been allocated for consistent mode I/O (see ddi_dma_addr_bind_handle(9F) or ddi_dma_buf_bind_handle(9F)).

This cannot be stated too strongly. If a consistent view of the memory object must be ensured between the time DMA resources are allocated for the object and the time they are deallocated, you must call ddi_dma_sync() to ensure that either a CPU or a DMA device has such a consistent view.

What to set type to depends on the view you are trying to ensure consistency for. If the memory object is modified by a CPU, and the object is going to be read by the DMA engine of the device, use DDI_DMA_SYNC_FORDEV. This ensures that the device’s DMA engine sees any changes that a CPU has made to the memory object. If the DMA engine for the device has written to the memory object, and you are going to read (with a CPU) the object (using an extant virtual address mapping that you have to...
the memory object), use DDI_DMA_SYNC_FORCPU. This ensures that a CPU’s view of
the memory object includes any changes made to the object by the device’s DMA
engine. If you are only interested in the kernel’s view (kernel-space part of the CPU’s
view) you may use DDI_DMA_SYNC_FORKERNEL. This gives a hint to the
system—that is, if it is more economical to synchronize the kernel’s view only, then do
so; otherwise, synchronize for CPU.

RETURN VALUES

ddi_dma_sync() returns:

DDI_SUCCESS  Caches are successfully flushed.

DDI_FAILURE  The address range to be flushed is out of the address range
            established by ddi_dma_addr_bind_handle(9F) or
            ddi_dma_buf_bind_handle(9F).

CONTEXT

ddi_dma_sync() can be called from user or interrupt context.

SEE ALSO

ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_mem_alloc(9F),
ddi_dma_unbind_handle(9F)

Writing Device Drivers
**NAME**

`ddi_dma_unbind_handle` - unbinds the address in a DMA handle

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dma_unbind_handle(ddi_dma_handle_t handle);
```

**PARAMETERS**

`handle` The DMA handle previously allocated by a call to

`ddi_dma_alloc_handle(9F)`.

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**DESCRIPTION**

`ddi_dma_unbind_handle()` frees all DMA resources associated with an existing DMA handle. When a DMA transfer completes, the driver should call `ddi_dma_unbind_handle()` to free system DMA resources established by a call to `ddi_dma_buf_bind_handle(9F)` or `ddi_dma_addr_bind_handle(9F)`. `ddi_dma_unbind_handle()` does an implicit `ddi_dma_sync(9F)` making further synchronization steps unnecessary.

**RETURN VALUES**

- `DDI_SUCCESS` on success
- `DDI_FAILURE` on failure

**CONTEXT**

`ddi_dma_unbind_handle()` can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`,
`ddi_dma_buf_bind_handle(9F)`, `ddi_dma_free_handle(9F)`,
`ddi_dma_sync(9F)`

*Writing Device Drivers*
ddi_driver_major(9F)

NAME  
ddi_driver_major – return driver’s major device number

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

major_t ddi_driver_major(dev_info_t *dip);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI)

DESCRIPTION  
ddi_driver_major() returns the major device number for the driver associated with the supplied dev_info node. This value can then be used as an argument to makedevice(9F) to construct a complete dev_t.

PARAMETERS  
dip  
A pointer to the device’s dev_info structure.

RETURN VALUES  
ddi_driver_major() returns the major number of the driver bound to a device, if any, or DDI_MAJOR_T_NONE otherwise.

CONTEXT  
ddi_driver_major() can be called from kernel or interrupt context.

SEE ALSO  
ddi_driver_name(9F)

Writing Device Drivers
ddi_driver_name(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_driver_name – return normalized driver name</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/ddi.h>  
          | #include <sys/sunddi.h>  
          | const char *ddi_driver_name(dev_info_t *devi); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| PARAMETERS | dip A pointer to the device’s dev_info structure. |
| DESCRIPTION | ddi_driver_name() returns the normalized driver name. This name is typically derived from the device name property or the device compatible property. If this name is a driver alias, the corresponding driver name is returned. |
| RETURN VALUES | ddi_driver_name() returns the actual name of the driver bound to a device. |
| CONTEXT | ddi_driver_name() can be called from kernel, or interrupt context. |
| SEE ALSO | ddi_get_name(9F)  
          | Writing Device Drivers |
| WARNINGS | The name returned by ddi_driver_name() is read-only. |
### ddi_enter_critical(9F)

**NAME**
- ddi_enter_critical, ddi_exit_critical – enter and exit a critical region of control

**SYNOPSIS**
```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned int ddi_enter_critical(void);
void ddi_exit_critical(unsigned int ddic);
```

**INTERFACE LEVEL**
- Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- **ddic**
  - The returned value from the call to `ddi_enter_critical()` must be passed to `ddi_exit_critical()`.

**DESCRIPTION**
- Nearly all driver operations can be done without any special synchronization and protection mechanisms beyond those provided by, for example, mutexes (see `mutex(9F)`). However, for certain devices there can exist a very short critical region of code which *must* be allowed to run uninterrupted. The function `ddi_enter_critical()` provides a mechanism by which a driver can ask the system to guarantee to the best of its ability that the current thread of execution will neither be preempted nor interrupted. This stays in effect until a bracketing call to `ddi_exit_critical()` is made (with an argument which was the returned value from `ddi_enter_critical()`).

The driver may not call any functions external to itself in between the time it calls `ddi_enter_critical()` and the time it calls `ddi_exit_critical()`.

**RETURN VALUES**
- `ddi_enter_critical()` returns an opaque unsigned integer which must be used in the subsequent call to `ddi_exit_critical()`.

**CONTEXT**
- This function can be called from user or interrupt context.

**WARNINGS**
- Driver writers should note that in a multiple processor system this function does not temporarily suspend other processors from executing. This function also cannot guarantee to actually block the hardware from doing such things as interrupt acknowledge cycles. What it *can* do is guarantee that the currently executing thread will not be preempted.

Do not write code bracketed by `ddi_enter_critical()` and `ddi_exit_critical()` that can get caught in an infinite loop, as the machine may crash if you do.

**SEE ALSO**
- `mutex(9F)`

*Writing Device Drivers*
ddi_enter_critical, ddi_exit_critical – enter and exit a critical region of control

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned int ddi_enter_critical(void);
void ddi_exit_critical(unsigned int ddic);
```

Nearly all driver operations can be done without any special synchronization and protection mechanisms beyond those provided by, for example, mutexes (see mutex(9F)). However, for certain devices there can exist a very short critical region of code which must be allowed to run uninterrupted. The function ddi_enter_critical() provides a mechanism by which a driver can ask the system to guarantee to the best of its ability that the current thread of execution will neither be preempted nor interrupted. This stays in effect until a bracketing call to ddi_exit_critical() is made (with an argument which was the returned value from ddi_enter_critical()).

The driver may not call any functions external to itself in between the time it calls ddi_enter_critical() and the time it calls ddi_exit_critical().

The returned value from the call to ddi_enter_critical() must be passed to ddi_exit_critical().

```
Context
```

This function can be called from user or interrupt context.

```
Warnings
```

Driver writers should note that in a multiple processor system this function does not temporarily suspend other processors from executing. This function also cannot guarantee to actually block the hardware from doing such things as interrupt acknowledge cycles. What it can do is guarantee that the currently executing thread will not be preempted.

Do not write code bracketed by ddi_enter_critical() and ddi_exit_critical() that can get caught in an infinite loop, as the machine may crash if you do.

```
See Also
```

mutex(9F)

Writing Device Drivers
ddi_ffs(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_ffs, ddi_fl — find first (last) bit set in a long integer</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>  

    int ddi_ffs(long mask);  
    int ddi_fl(long mask); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| PARAMETERS | mask — A 32-bit argument value to search through. |
| DESCRIPTION | The function ddi_ffs() takes its argument and returns the shift count that the first (least significant) bit set in the argument corresponds to. The function ddi_fl() does the same, only it returns the shift count for the last (most significant) bit set in the argument. |
| RETURN VALUES | 0 — No bits are set in mask.  
N — Bit N is the least significant (ddi_ffs) or most significant (ddi_fl) bit set in mask. Bits are numbered from 1 to 32, with bit 1 being the least significant bit position and bit 32 the most significant position. |
| CONTEXT | This function can be called from user or interrupt context. |
| SEE ALSO | Writing Device Drivers |

472  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 20 Dec 1995
ddi_ffs, ddi_fl — find first (last) bit set in a long integer

### SYNOPSIS
```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_ffs(long mask);
int ddi_fls(long mask);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **mask**: A 32-bit argument value to search through.

### DESCRIPTION
The function `ddi_ffs()` takes its argument and returns the shift count that the first (least significant) bit set in the argument corresponds to. The function `ddi_fls()` does the same, only it returns the shift count for the last (most significant) bit set in the argument.

### RETURN VALUES
- **0**: No bits are set in mask.
- **N**: Bit N is the least significant (`ddi_ffs`) or most significant (`ddi_fls`) bit set in mask. Bits are numbered from 1 to 32, with bit 1 being the least significant bit position and bit 32 the most significant position.

### CONTEXT
This function can be called from user or interrupt context.

### SEE ALSO
- Writing Device Drivers
- `Kernel Functions for Drivers`
### NAME

ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll – read data from the mapped memory address, device register or allocated DMA memory address

### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

### INTERFACE LEVEL PARAMETERS

- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **dev_addr**: Base device address.

### DESCRIPTION

The `ddi_get8()`, `ddi_get16()`, `ddi_get32()`, and `ddi_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See `sysbus(4)`, `isa(4)`, `eisa(4)`, and `sbus(4)` for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See `pci(4)`.

### RETURN VALUES

These functions return the value read from the mapped address.

### CONTEXT

These functions can be called from user, kernel, or interrupt context.

### SEE ALSO

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)

### NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>

Solaris DDI specific (Solaris DDI).
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_get32(9F)

NAME

ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll –
read data from the mapped memory address, device register or allocated DMA
memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr Base device address.

DESCRIPTION

The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read
8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address,
dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)

NOTES

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>

476 man pages section 9: DDI and DKI Kernel Functions • Last Revised 22 Nov 1996
## ddi_get32(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
NAME

ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll –
read data from the mapped memory address, device register or allocated DMA
memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
        ddi_regs_map_setup(9F).

dev_addr Base device address.

DESCRIPTION

The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read
8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address,
dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
        ddi_rep_get8(9F), ddi_rep_put8(9F)

NOTES

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>
## ddi_get64(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_get1</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_get8(9F)

NAME

ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll – read data from the mapped memory address, device register or allocated DMA memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t  ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE LEVEL PARAMETERS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr Base device address.

DESCRIPTION

The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F)

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers  481

ddi_get8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl –
read data from the mapped memory address, device register or allocated DMA
memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr Base device address.

The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read
8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address,
dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

These functions return the value read from the mapped address.

These functions can be called from user, kernel, or interrupt context.

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>

482  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 22 Nov 1996
## ddi_getb(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_get_cred — returns a pointer to the credential structure of the caller

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

cred_t *ddi_get_cred(void);
```

**DESCRIPTION**

Solaris DDI specific (Solaris DDI).

**ddi_get_cred()** returns a pointer to the user credential structure of the caller.

**RETURN VALUES**

**ddi_get_cred()** returns a pointer to the caller’s credential structure.

**CONTEXT**

**ddi_get_cred()** can be called from user context only.

**SEE ALSO**

*Writing Device Drivers*
ddi_get_devstate - Check device state

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

ddi_devstate_t ddi_get_devstate(dev_info_t *dip);

INTERFACE level
Solaris DDI specific (Solaris DDI)

PARAMETERS

dip Pointer to the device’s dev_info structure

DESCRIPTION
The ddi_get_devstate() function returns a value indicating the state of the device specified by dip, as derived from the configuration operations that have been performed on it (or on the bus on which it resides) and any fault reports relating to it.

RETURN VALUES
DDI_DEVSTATE_OFFLINE
The device is offline. In this state, the device driver is not attached, nor will it be attached automatically. The device cannot be used until it is brought online.

DDI_DEVSTATE_DOWN
The device is online but unusable due to a fault.

DDI_DEVSTATEQUIESCED
The bus on which the device resides has been quiesced. This is not a fault, but no operations on the device should be performed while the bus remains quiesced.

DDI_DEVSTATE_DGRATED
The device is online but only able to provide a partial or degraded service, due to a fault.

DDI_DEVSTATE_UP
The device is online and fully operational.

CONTEXT
The ddi_get_devstate() function may be called from user, kernel, or interrupt context.

NOTES
A device driver should call this function to check its own state at each major entry point, and before committing resources to a requested operation. If a driver discovers that its device is already down, it should perform required cleanup actions and return as soon as possible. If appropriate, it should return an error to its caller, indicating that the device has failed (for example, a driver’s read(9E) routine would return EIO).

Depending on the driver, some non-I/O operations (for example, calls to the driver’s ioctl(9E) routine) may still succeed; only functions which would require fully accessible and operational hardware will necessarily fail. If the bus on which the device resides is quiesced, the driver may return a value indicating the operation should be retried later (for example, EAGAIN). Alternatively, for some classes of device, it may be appropriate for the driver to enqueue the operation and service it once the bus has been unquiesced. Note that not all busses support the quiesce/unquiesce operations, so this value may never be seen by some drivers.

SEE ALSO
attach(9E), ioctl(9E), open(9E), read(9E), strategy(9E), write(9E), ddi_dev_report_fault(9F)
ddi_get_driver_private() returns the contents of devi_driver_data. If ddi_set_driver_private() has not been previously called with dip, an unpredictable value is returned.

CONTEXT These functions can be called from user or interrupt context.

SEE ALSO Writing Device Drivers
ddi_get_iblock_cookie(9F)

NAME

ddi_add_intr, ddi_get_iblock_cookie, ddi_remove_intr – hardware interrupt handling routines

SYNOPSIS

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_iblock_cookie(dev_info_t *dip, uint_t inumber,
                          ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_intr(dev_info_t *dip, uint_t inumber,
                  ddi_iblock_cookie_t *iblock_cookiep,
                  ddi_idevice_cookie_t *idevice_cookiep,
                  uint_t (*int_handler) (caddr_t), caddr_t int_handler_arg);

void ddi_remove_intr(dev_info_t *dip, uint_t inumber,
                     ddi_iblock_cookie_t iblock_cookie);
```

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

For ddi_get_iblock_cookie():

- **dip**: Pointer to dev_info structure.
- **inumber**: Interrupt number.
- **iblock_cookiep**: Pointer to an interrupt block cookie.

For ddi_add_intr():

- **dip**: Pointer to dev_info structure.
- **inumber**: Interrupt number.
- **iblock_cookiep**: Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
- **idevice_cookiep**: Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.
- **int_handler**: Pointer to interrupt handler.
- **int_handler_arg**: Argument for interrupt handler.

For ddi_remove_intr():

- **dip**: Pointer to dev_info structure.
- **inumber**: Interrupt number.
- **iblock_cookie**: Block cookie which identifies the interrupt handler to be removed.

**ddi_get_iblock_cookie()** retrieves the interrupt block cookie associated with a particular interrupt specification. This routine should be called before **ddi_add_intr()** to retrieve the interrupt block cookie needed to initialize locks.
mutex(9F), rwlock(9F)) used by the interrupt routine. The interrupt number \texttt{inumber} determines for which interrupt specification to retrieve the cookie. \texttt{inumber} is associated with information provided either by the device (see \texttt{sbus(4)) or the hardware configuration file (see \texttt{sysbus(4)}, \texttt{isa(4)}, \texttt{eisa(4)}, and \texttt{driver.conf(4)}). If only one interrupt is associated with the device, \texttt{inumber} should be 0.

On a successful return, \texttt{*iblock_cookiep} contains information needed for initializing locks associated with the interrupt specification corresponding to \texttt{inumber} (see \texttt{mutex_init(9F) and rw_init(9F)}). The driver can then initialize locks acquired by the interrupt routine before calling \texttt{ddi_add_intr()} which prevents a possible race condition where the driver's interrupt handler is called immediately after the driver has called \texttt{ddi_add_intr()} but before the driver has initialized the locks. This may happen when an interrupt for a different device occurs on the same interrupt level. If the interrupt routine acquires the lock before the lock has been initialized, undefined behavior may result.

\texttt{ddi_add_intr()} adds an interrupt handler to the system. The interrupt number \texttt{inumber} determines which interrupt the handler will be associated with. (Refer to \texttt{ddi_get_iblock_cookie()} above.)

On a successful return, \texttt{iblock_cookiep} contains information used for initializing locks associated with this interrupt specification (see \texttt{mutex_init(9F) and rw_init(9F)}). Note that the interrupt block cookie is usually obtained using \texttt{ddi_get_iblock_cookie()} to avoid the race conditions described above (refer to \texttt{ddi_get_iblock_cookie()} above). For this reason, \texttt{iblock_cookiep} is no longer useful and should be set to NULL.

On a successful return, \texttt{idevice_cookiep} contains a pointer to a \texttt{ddi_idevice_cookie_t} structure (see \texttt{ddi_idevice_cookie(9S)}) containing information useful for some devices that have programmable interrupts. If \texttt{idevice_cookiep} is set to NULL, no value is returned.

The routine \texttt{intr_handler}, with its argument \texttt{int_handler_arg}, is called upon receipt of the appropriate interrupt. The interrupt handler should return \texttt{DDI_INTR_CLAIMED} if the interrupt was claimed, \texttt{DDI_INTR_UNCLAIMED} otherwise.

If successful, \texttt{ddi_add_intr()} will return \texttt{DDI_SUCCESS}. \texttt{DDI_INTR_NOTFOUND} is returned on i86pc and sun4m architectures if the interrupt information cannot be found. If the interrupt information cannot be found on the sun4u architecture, either \texttt{DDI_INTR_NOTFOUND} or \texttt{DDI_FAILURE} can be returned.

\texttt{ddi_remove_intr()} removes an interrupt handler from the system. Unloadable drivers should call this routine during their \texttt{detach(9E)} routine to remove their interrupt handler from the system.
The device interrupt routine for this instance of the device will not execute after `ddi_remove_intr()` returns. `ddi_remove_intr()` may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to `ddi_remove_intr()` or deadlock may result.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See `sysbus(4)`, `isa(4)`, `eisa(4)`, and `sbus(4)` for details.

**RETURN VALUES**

`ddi_add_intr()` and `ddi_get_iblock_cookie()` return:

- **DDI_SUCCESS**  
  On success.

- **DDI_INTR_NOTFOUND**  
  On failure to find the interrupt.

- **DDI_FAILURE**  
  On failure. **DDI_FAILURE** can also be returned on failure to find interrupt (sun4u).

**CONTEXT**

`ddi_add_intr()`, `ddi_remove_intr()`, and `ddi_get_iblock_cookie()` can be called from user or kernel context.

**SEE ALSO**

`driver.conf(4)`, `eisa(4)`, `isa(4)`, `sbus(4)`, `sysbus(4)`, `attach(9E)`, `detach(9E)`, `ddi_intr_hilevel(9F)`, `mutex(9F)`, `mutex_init(9F)`, `rw_init(9F)`, `rwlock(9F)`, `ddi_idevice_cookie(9S)`

**Writing Device Drivers**

**NOTES**

`ddi_get_iblock_cookie()` must not be called after the driver adds an interrupt handler for the interrupt specification corresponding to `inumber`.

All consumers of these interfaces, checking return codes, should verify `return_code != DDI_SUCCESS`. Checking for specific failure codes can result in inconsistent behaviors among platforms.

**BUGS**

The `idevice_cookiep` should really point to a data structure that is specific to the bus architecture that the device operates on. Currently the SBus and PCI buses are supported and a single data structure is used to describe both.
ddi_getiminor(9F)

NAME  ddi_getiminor – get kernel internal minor number from an external dev_t

SYNOPSIS  
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

minor_t ddi_getiminor(dev_t dev);

INTERFACE LEVEL  This interface is obsolete. getminor(9F) should be used instead.

PARAMETERS  The following parameters are supported:

  dev  Device number.

DESCRIPTION  ddi_getiminor() extracts the minor number from a device number. This call should be used only for device numbers that have been passed to the kernel from the user space through opaque interfaces such as the contents of ioctl(9E) and putmsg(2). The device numbers passed in using standard device entry points must continue to be interpreted using the getminor(9F) interface. This new interface is used to translate between user visible device numbers and in kernel device numbers. The two numbers may differ in a clustered system.

  For certain bus types, you can call this DDI function from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details.

CONTEXT  ddi_getiminor() can be called from user context only.

RETURN VALUES  The minor number or EMINOR_UNKNOWN if the minor number of the device is invalid.

ATTRIBUTES  See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5), getmajor(9F), getminor(9F), makedevice(9F)

Writing Device Drivers

WARNINGS  Validity checking is performed. If dev is invalid, EMINOR_UNKNOWN is returned. This behavior differs from getminor(9F).
ddi_get_instance – get device instance number

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_instance(dev_info_t *dip);

Solaris DDI specific (Solaris DDI).

dip Pointer to dev_info structure.

ddi_get_instance() returns the instance number of the device corresponding to

dip.

The system assigns an instance number to every device. Instance numbers for devices
attached to the same driver are unique. This provides a way for the system and the
driver to uniquely identify one or more devices of the same type. The instance number
is derived by the system from different properties for different device types in an
implementation specific manner.

Once an instance number has been assigned to a device, it will remain the same even
across reconfigurations and reboots. Therefore, instance numbers seen by a driver may
not appear to be in consecutive order. For example, if device foo0 has been assigned
an instance number of 0 and device foo1 has been assigned an instance number of 1,
if foo0 is removed, foo1 will continue to be associated with instance number 1 (even
though foo1 is now the only device of its type on the system).

ddi_get_instance() returns the instance number of the device corresponding to
dip.

ddi_get_instance() can be called from user or interrupt context.

SEE ALSO

path_to_inst(4)

Writing Device Drivers
ddi_get_kt_did(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_get_kt_did – get identifier of current thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/types.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/sunddi.h&gt;</td>
</tr>
</tbody>
</table>

```c
kt_did_t ddi_get_kt_did(void);
```

<table>
<thead>
<tr>
<th>INTERFACE LEVEL</th>
<th>Solaris DDI specific (Solaris DDI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTION</td>
<td>The ddi_get_kt_did() function returns a unique 64-bit identifier for the currently running thread.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>This routine can be called from user, kernel, or interrupt context. This routine cannot be called from a high-level interrupt context.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>ddi_get_kt_did() always returns the identifier for the current thread. There are no error conditions.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td>NOTES</td>
<td>The value returned by this function can also be seen in adb or mdb as the did field displayed when using the thread macro. This interface is intended for tracing and debugging purposes.</td>
</tr>
</tbody>
</table>
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

These functions return the value read from the mapped address.

These functions can be called from user, kernel, or interrupt context.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>

See also:

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)
### ddi_getl(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
### NAME
`ddi_get_lbolt` – returns the value of `lbolt`

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

clock_t ddi_get_lbolt(void);
```

### INTERFACE
Solaris DDI specific (Solaris DDI).

### LEVEL
DESCRIPTION
`ddi_get_lbolt()` returns the value of `lbolt` where `lbolt` is an integer that represents the number of clock ticks since the last system reboot. This value is used as a counter or timer inside the system kernel. The tick frequency can be determined by using `drv_usectohz(9F)` which converts microseconds into clock ticks.

### RETURN VALUES
`ddi_get_lbolt()` returns the value of `lbolt`.

### CONTEXT
This routine can be called from any context.

### SEE ALSO
- `ddi_get_time(9F)`, `drv_getparm(9F)`, `drv_usectohz(9F)`
- Writing Device Drivers
- STREAMS Programming Guide
ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll –
read data from the mapped memory address, device register or allocated DMA
memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read
8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address,
dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swAPPING if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

These functions return the value read from the mapped address.

These functions can be called from user, kernel, or interrupt context.

ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
<tr>
<td>Previous Name</td>
<td>New Name</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen -
get property information for leaf device drivers

#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op, int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name, int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *name, int *lengthp);

Solaris DDI specific (Solaris DDI). The ddi_getlongprop(),
  ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions
  are obsolete. Use ddi_prop_lookup(9F) instead of ddi_getlongprop(),
  ddi_getlongprop_buf(), and ddi_getproplen(). Use ddi_prop_get_int(9F)
  instead of ddi_getprop().

PARAMETERS

devo Device number associated with property or DDI_DEV_T_ANY as the
wildcard device number.

dip Pointer to a device info node.

prop_op Property operator.

flags Possible flag values are some combination of:

    DDI_PROP_DONTPASS
do not pass request to parent device information node if property not
    found

    DDI_PROP_CANSLEEP
    the routine may sleep while allocating memory

    DDI_PROP_NOTPROM
    do not look at PROM properties (ignored on architectures that do not
    support PROM properties)

name String containing the name of the property.

valuep If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the
users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the
address of a pointer.
lengthp  On exit, *lengthp will contain the property length. If prop_op is PROP_LEN_AND_VAL_BUF then before calling ddi_prop_op(), lengthp should point to an int that contains the length of callers buffer.

defvalue  The value that ddi_getprop() returns if the property is not found.

**DESCRIPTION**

ddi_prop_op() gets arbitrary-size properties for leaf devices. The routine searches the device’s property list. If it does not find the property at the device level, it examines the flags argument, and if DDI_PROP_DONTPASS is set, then ddi_prop_op() returns DDI_PROP_NOT_FOUND. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns DDI_PROP_UNDEFINED. Otherwise it returns either the property length, or both the length and value of the property to the caller via the valuep and lengthp pointers, depending on the value of prop_op, as described below, and returns DDI_PROP_SUCCESS. If a property cannot be found at all, DDI_PROP_NOT_FOUND is returned.

Usually, the dev argument should be set to the actual device number that this property applies to. However, if the dev argument is DDI_DEV_T_ANY, the wildcard dev, then ddi_prop_op() will match the request based on name only (regardless of the actual dev the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in last-in, first-out (LIFO) order, next search software properties created by the system in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the prop_op argument. If prop_op is PROP_LEN, then ddi_prop_op() just sets the callers length, *lengthp, to the property length and returns the value DDI_PROP_SUCCESS to the caller. The valuep argument is not used in this case. Property lengths are 0 for boolean properties, sizeof (int) for integer properties, and size in bytes for long (variable size) properties.

If prop_op is PROP_LEN_AND_VAL_BUF, then valuep should be a pointer to a user-supplied buffer whose length should be given in *lengthp by the caller. If the requested property exists, ddi_prop_op() first sets *lengthp to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns DDI_PROP_SUCCESS. If the named property exists but the buffer supplied is too small to hold it, it returns DDI_PROP_BUF_TOO_SMALL.

If prop_op is PROP_LEN_AND_VAL_ALLOC, and the property is found, ddi_prop_op() sets *lengthp to the property length. It then attempts to allocate a buffer to return to the caller using the kmem_alloc(9F) routine, so that memory can be later recycled using kmem_free(9F). The driver is expected to call kmem_free() with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets *valuep to point to the allocated buffer, copies the property value into the buffer and returns DDI_PROP_SUCCESS. Otherwise, it returns ddi_getlongprop(9F)
ddi_getlongprop(9F)

DDI_PROP_NO_MEMORY. Note that the flags argument may affect the behavior of memory allocation in ddi_prop_op(). In particular, if DDI_PROP_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop() returns boolean and integer-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

RETURN VALUES

ddi_prop_op() ddi_getlongprop() ddi_getlongprop_buf()

ddi_getproplen() return:

DDI_PROP_SUCCESS Property found and returned.
DDI_PROP_NOT_FOUND Property not found.
DDI_PROP_UNDEFINED Property already explicitly undefined.
DDI_PROP_NO_MEMORY Property found, but unable to allocate memory. lengthp points to the correct property length.
DDI_PROP_BUF_TOO_SMALL Property found, but the supplied buffer is too small. lengthp points to the correct property length.

ddi_getprop() returns:

The value of the property or the value passed into the routine as defvalue if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

CONTEXT

These functions can be called from user or interrupt context, provided DDI_PROP_CANSLEEP is not set; if it is set, they can be called from user context only.

ATTRIBUTES

See attributes(5) for a description of the following attributes:
**ddi_getlongprop(9F)**

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_getlongprop(), ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions are Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ddi_prop_create(9F), ddi_prop_get_int(9F), ddi_prop_lookup(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
ddi_getlongprop_buf(9F)

NAME

ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen –
get property information for leaf device drivers

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name,
    int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char
    *name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char
    *name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *name,
    int *lengthp);

INTERFACE

Solaris DDI specific (Solaris DDI). The ddi_getlongprop(),
ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions
are obsolete. Use ddi_prop_lookup(9F) instead of ddi_getlongprop(),
ddi_getlongprop_buf(), and ddi_getproplen(). Use ddi_prop_get_int(9F)
instead of ddi_getprop().

LEVEL

PARAMETERS

dev Device number associated with property or DDI_DEV_T_ANY as the
    wildcard device number.

dip Pointer to a device info node.

prop_op Property operator.

flags Possible flag values are some combination of:

DDI_PROP_DONTPASS
    do not pass request to parent device information node if property not
    found

DDI_PROP_CANSLEEP
    the routine may sleep while allocating memory

DDI_PROP_NOTPROM
    do not look at PROM properties (ignored on architectures that do not
    support PROM properties)

name String containing the name of the property.

valuep If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the
    users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the
    address of a pointer.
**DESCRIPTION**

`ddi_prop_op()` gets arbitrary-size properties for leaf devices. The routine searches the device’s property list. If it does not find the property at the device level, it examines the `flags` argument, and if `DDI_PROP_DONTPASS` is set, then `ddi_prop_op()` returns `DDI_PROP_NOT_FOUND`. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns `DDI_PROP_UNDEFINED`. Otherwise it returns either the property length, or both the length and value of the property to the caller via the `valuep` and `lengthp` pointers, depending on the value of `prop_op`, as described below, and returns `DDI_PROP_SUCCESS`. If a property cannot be found at all, `DDI_PROP_NOT_FOUND` is returned.

Usually, the `dev` argument should be set to the actual device number that this property applies to. However, if the `dev` argument is `DDI_DEV_T_ANY`, the wildcard `dev`, then `ddi_prop_op()` will match the request based on `name` only (regardless of the actual `dev` the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in last-in, first-out (LIFO) order, next search software properties created by the system in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the `prop_op` argument. If `prop_op` is `PROP_LEN`, then `ddi_prop_op()` just sets the callers length, `*lengthp`, to the property length and returns the value `DDI_PROP_SUCCESS` to the caller. The `valuep` argument is not used in this case. Property lengths are 0 for boolean properties, `sizeof (int)` for integer properties, and size in bytes for long (variable size) properties.

If `prop_op` is `PROP_LEN_AND_VAL_BUF`, then `valuep` should be a pointer to a user-supplied buffer whose length should be given in `*lengthp` by the caller. If the requested property exists, `ddi_prop_op()` first sets `*lengthp` to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns `DDI_PROP_SUCCESS`. If the named property exists but the buffer supplied is too small to hold it, it returns `DDI_PROP_BUF_TOO_SMALL`.

If `prop_op` is `PROP_LEN_AND_VAL_ALLOC`, and the property is found, `ddi_prop_op()` sets `*lengthp` to the property length. It then attempts to allocate a buffer to return to the caller using the `kmem_alloc(9F)` routine, so that memory can be later recycled using `kmem_free(9F)`. The driver is expected to call `kmem_free()` with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets `*valuep` to point to the allocated buffer, copies the property value into the buffer and returns `DDI_PROP_SUCCESS`. Otherwise, it returns `DDI_PROP_NOT_FOUND`.

**On exit,** `*lengthp` will contain the property length. If `prop_op` is `PROP_LEN_AND_VAL_BUF` then before calling `ddi_prop_op()`, `lengthp` should point to an `int` that contains the length of callers buffer.

**Defvalue** The value that `ddi_getprop()` returns if the property is not found.
Note that the flags argument may affect the behavior of memory allocation in ddi_prop_op(). In particular, if DDI_PROP_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop() returns boolean and integer-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_op()</td>
<td>ddi_getlongprop() ddi_getlongprop_buf() ddi_getproplen() return:</td>
</tr>
<tr>
<td>DDI_PROP_NO_MEMORY</td>
<td>Property found and returned.</td>
</tr>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>Property not found.</td>
</tr>
<tr>
<td>DDI_PROP_NOT_FOUND</td>
<td>Property already explicitly undefined.</td>
</tr>
<tr>
<td>DDI_PROP_UNDEFINED</td>
<td>Property found, but unable to allocate memory. lengthp points to the correct property length.</td>
</tr>
<tr>
<td>DDI_PROP_NO_MEMORY</td>
<td>Property found, but the supplied buffer is too small. lengthp points to the correct property length.</td>
</tr>
<tr>
<td>DDI_PROP_BUF_TOO_SMALL</td>
<td></td>
</tr>
</tbody>
</table>

ddi_getprop() returns:

The value of the property or the value passed into the routine as defvalue if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

**CONTEXT**

These functions can be called from user or interrupt context, provided DDI_PROP_CANSLEEP is not set; if it is set, they can be called from user context only.

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:
<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_getlongprop(), ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions are Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), ddi_prop_create(9F), ddi_prop_get_int(9F), ddi_prop_lookup(9F), kmem_alloc(9F), kmem_free(9F)

*Writing Device Drivers*
**ddi_get_name(9F)**

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_binding_name, ddi_get_name – return driver binding name</th>
</tr>
</thead>
</table>
| SYNOPSIS   | #include <sys/ddi.h>  
#include <sys/sunddi.h>  
char *ddi_binding_name(dev_info_t *dip);  
char *ddi_get_name(dev_info_t *dip); |
| INTERFACE  | Solaris DDI specific (Solaris DDI). |
| LEVEL      | parameters |
| PARAMETERS | dip A pointer to the device’s dev_info structure. |
| DESCRIPTION| ddi_binding_name() and ddi_get_name() return the driver binding name. This is the name used to select a driver for the device. This name is typically derived from the device name property or the device compatible property. The name returned may be a driver alias or the driver name. |
| RETURN VALUES | ddi_binding_name() and ddi_get_name() return the name used to bind a driver to a device. |
| CONTEXT    | ddi_binding_name() and ddi_get_name() can be called from user, kernel, or interrupt context. |
| SEE ALSO   | ddi_node_name(9F)  
Writing Device Drivers |
| WARNINGS   | The name returned by ddi_binding_name() and ddi_get_name() is read-only. |
### ddi_get_parent

**NAME**
ddi_get_parent – find the parent of a device information structure

**SYNOPSIS**
```
#include <sys/ddi.h>
#include <sys/sunddi.h>

dev_info_t *ddi_get_parent(dev_info_t *dip);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- `dip` Pointer to a device information structure.

**DESCRIPTION**
`ddi_get_parent()` returns a pointer to the device information structure which is the parent of the one pointed to by `dip`.

**RETURN VALUES**
`ddi_get_parent()` returns a pointer to a device information structure.

**CONTEXT**
`ddi_get_parent()` can be called from user or interrupt context.

**SEE ALSO**
- Writing Device Drivers
ddi_get_pid(9F)

NAME  ddi_get_pid – returns the process ID

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

pid_t ddi_get_pid(void);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION
ddi_get_pid() obtains the process ID of the current process. This value can be used to allow only a select process to perform a certain operation. It can also be used to determine whether a device context belongs to the current process.

RETURN VALUES
ddi_get_pid() returns the process ID.

CONTEXT
This routine can be called from user context only.

SEE ALSO
drv_getparm(9F)

Writing Device Drivers
STREAMS Programming Guide
ddi_getprop(9F)

**NAME**
ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen —
get property information for leaf device drivers

**SYNOPSIS**
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
                 int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name,
                 int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char *
                     name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char *
                        name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *
                    name, int *lengthp);

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI). The ddi_getlongprop(),
ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions
are obsolete. Use ddi_prop_lookup(9F) instead of ddi_getlongprop(),
ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen(). Use ddi_prop_get_int(9F)
instead of ddi_getprop().

**PARAMETERS**

- **dev**
  Device number associated with property or DDI_DEV_T_ANY as the
  wildcard device number.

- **dip**
  Pointer to a device info node.

- **prop_op**
  Property operator.

- **flags**
  Possible flag values are some combination of:

  - **DDI_PROP_DONTPASS**
    do not pass request to parent device information node if property not
    found

  - **DDI_PROP_CANSLEEP**
    the routine may sleep while allocating memory

  - **DDI_PROP_NOTPROM**
    do not look at PROM properties (ignored on architectures that do not
    support PROM properties)

- **name**
  String containing the name of the property.

- **valuep**
  If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the
  users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the
  address of a pointer.

  - **lengthp**

**DESCRIPTION**

`ddi_prop_op()` gets arbitrary-size properties for leaf devices. The routine searches the device’s property list. If it does not find the property at the device level, it examines the `flags` argument, and if `DDI_PROP_DONTPASS` is set, then `ddi_prop_op()` returns `DDI_PROP_NOT_FOUND`. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns `DDI_PROP_UNDEFINED`. Otherwise it returns either the property length, or both the length and value of the property to the caller via the `valuep` and `lengthp` pointers, depending on the value of `prop_op`, as described below, and returns `DDI_PROP_SUCCESS`. If a property cannot be found at all, `DDI_PROP_NOT_FOUND` is returned.

Usually, the `dev` argument should be set to the actual device number that this property applies to. However, if the `dev` argument is `DDI_DEV_T_ANY`, the wildcard `dev`, then `ddi_prop_op()` will match the request based on name only (regardless of the actual `dev` the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in last-in, first-out (LIFO) order, next search software properties created by the `system` in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the `prop_op` argument. If `prop_op` is `PROP_LEN`, then `ddi_prop_op()` just sets the callers length, `*lengthp`, to the property length and returns the value `DDI_PROP_SUCCESS` to the caller. The `valuep` argument is not used in this case. Property lengths are 0 for boolean properties, `sizeof (int)` for integer properties, and size in bytes for long (variable size) properties.

If `prop_op` is `PROP_LEN_AND_VAL_BUF`, then `valuep` should be a pointer to a user-supplied buffer whose length should be given in `*lengthp` by the caller. If the requested property exists, `ddi_prop_op()` first sets `*lengthp` to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns `DDI_PROP_SUCCESS`. If the named property exists but the buffer supplied is too small to hold it, it returns `DDI_PROP_BUF_TOO_SMALL`.

If `prop_op` is `PROP_LEN_AND_VAL_ALLOC`, and the property is found, `ddi_prop_op()` sets `*lengthp` to the property length. It then attempts to allocate a buffer to return to the caller using the `kmem_alloc(9F)` routine, so that memory can be later recycled using `kmem_free(9F)`. The driver is expected to call `kmem_free()` with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets `*valuep` to point to the allocated buffer, copies the property value into the buffer and returns `DDI_PROP_SUCCESS`. Otherwise, it returns...
DDI_PROP_NO_MEMORY. Note that the flags argument may affect the behavior of memory allocation in ddi_prop_op(). In particular, if DDI_PROP_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop() returns boolean and integer-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

RETURN VALUES

ddi_prop_op() ddi_getlongprop() ddi_getlongprop_buf() ddi_getproplen() return:

DDI_PROP_SUCCESS Property found and returned.
DDI_PROP_NOT_FOUND Property not found.
DDI_PROP_UNDEFINED Property already explicitly undefined.
DDI_PROP_NO_MEMORY Property found, but unable to allocate memory. lengthp points to the correct property length.
DDI_PROP_BUF_TOO_SMALL Property found, but the supplied buffer is too small. lengthp points to the correct property length.

ddi_getprop() returns:

The value of the property or the value passed into the routine as defvalue if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

CONTEXT

These functions can be called from user or interrupt context, provided DDI_PROP_CANSLEEP is not set; if it is set, they can be called from user context only.

ATTRIBUTES

See attributes(5) for a description of the following attributes:
SEE ALSO

attributes(5), ddi_prop_create(9F), ddi_prop_get_int(9F),
ddi_prop_lookup(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen –
get property information for leaf device drivers

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name,
int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char
*name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char
*name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *name,
int *lengthp);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_getlongprop(),
ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions
are obsolete. Use ddi_prop_lookup(9F) instead of ddi_getlongprop(),
ddi_getlongprop_buf(), and ddi_getproplen(). Use ddi_prop_get_int(9F)
instead of ddi_getprop().

LEVEL

PARAMETERS  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>Device number associated with property or DDI_DEV_T_ANY as the wildcard device number.</td>
</tr>
<tr>
<td>dip</td>
<td>Pointer to a device info node.</td>
</tr>
<tr>
<td>prop_op</td>
<td>Property operator.</td>
</tr>
<tr>
<td>flags</td>
<td>Possible flag values are some combination of:</td>
</tr>
<tr>
<td></td>
<td>DDI_PROP_DONTPASS do not pass request to parent device information node if property not found</td>
</tr>
<tr>
<td></td>
<td>DDI_PROP_CANSLEEP the routine may sleep while allocating memory</td>
</tr>
<tr>
<td></td>
<td>DDI_PROP_NOTPROM do not look at PROM properties (ignored on architectures that do not support PROM properties)</td>
</tr>
<tr>
<td>name</td>
<td>String containing the name of the property.</td>
</tr>
<tr>
<td>valuep</td>
<td>If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the address of a pointer.</td>
</tr>
</tbody>
</table>
On exit, *lengthp will contain the property length. If prop_op is PROP_LEN_AND_VAL_BUF then before calling ddi_prop_op(), lengthp should point to an int that contains the length of callers buffer.

defvalue The value that ddi_getprop() returns if the property is not found.

DESCRIPTION

ddi_prop_op() gets arbitrary-size properties for leaf devices. The routine searches the device’s property list. If it does not find the property at the device level, it examines the flags argument, and if DDI_PROP_DONTPASS is set, then ddi_prop_op() returns DDI_PROP_NOT_FOUND. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns DDI_PROP_UNDEFINED. Otherwise it returns either the property length, or both the length and value of the property to the caller via the valuep and lengthp pointers, depending on the value of prop_op, as described below, and returns DDI_PROP_SUCCESS. If a property cannot be found at all, DDI_PROP_NOT_FOUND is returned.

Usually, the dev argument should be set to the actual device number that this property applies to. However, if the dev argument is DDI_DEV_T_ANY, the wildcard dev, then ddi_prop_op() will match the request based on name only (regardless of the actual dev the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in last-in, first-out (LIFO) order, next search software properties created by the system in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the prop_op argument. If prop_op is PROP_LEN, then ddi_prop_op() just sets the callers length, *lengthp, to the property length and returns the value DDI_PROP_SUCCESS to the caller. The valuep argument is not used in this case. Property lengths are 0 for boolean properties, sizeof (int) for integer properties, and size in bytes for long (variable size) properties.

If prop_op is PROP_LEN_AND_VAL_BUF, then valuep should be a pointer to a user-supplied buffer whose length should be given in *lengthp by the caller. If the requested property exists, ddi_prop_op() first sets *lengthp to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns DDI_PROP_SUCCESS. If the named property exists but the buffer supplied is too small to hold it, it returns DDI_PROP_BUF_TOO_SMALL.

If prop_op is PROP_LEN_AND_VAL_ALLOC, and the property is found, ddi_prop_op() sets *lengthp to the property length. It then attempts to allocate a buffer to return to the caller using the kmem_alloc(9F) routine, so that memory can be later recycled using kmem_free(9F). The driver is expected to call kmem_free() with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets *valuep to point to the allocated buffer, copies the property value into the buffer and returns DDI_PROP_SUCCESS. Otherwise, it returns
DDI_PROP_NO_MEMORY. Note that the flags argument may affect the behavior of memory allocation in ddi_prop_op(). In particular, if DDI_PROP_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop() returns boolean and integer-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

RETURN VALUES

ddi_prop_op() ddi_getlongprop() ddi_getlongprop_buf() ddi_getproplen() return:

DDI_PROP_SUCCESS          Property found and returned.
DDI_PROP_NOT_FOUND        Property not found.
DDI_PROP_UNDEFINED       Property already explicitly undefined.
DDI_PROP_NO_MEMORY       Property found, but unable to allocate memory. lengthp points to the correct property length.
DDI_PROP_BUF_TOO_SMALL   Property found, but the supplied buffer is too small. lengthp points to the correct property length.

ddi_getprop() returns:

The value of the property or the value passed into the routine as defvalue if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

CONTEXT

These functions can be called from user or interrupt context, provided DDI_PROP_CANSLEEP is not set; if it is set, they can be called from user context only.

ATTRIBUTES

See attributes(5) for a description of the following attributes:
SEE ALSO

attributes(5), ddi_prop_create(9F), ddi_prop_get_int(9F),
ddi_prop_lookup(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
### NAME
- ddi_add_softintr
- ddi_get_soft_iblock_cookie
- ddi_remove_softintr
- ddi_trigger_softintr

**Software interrupt handling routines**

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t *dip, int preference, ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t *idp, ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *idevice_cookiep, uint_t (*int_handler)(caddr_t int_handler_arg), caddr_t int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);

void ddi_trigger_softintr(ddi_softintr_t id);
```

### INTERFACE
- **LEVEL**
  - Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **ddi_get_soft_iblock_cookie()**
  - `dip` Pointer to a `dev_info` structure.
  - `preference` The type of soft interrupt to retrieve the cookie for.
  - `iblock_cookiep` Pointer to a location to store the interrupt block cookie.

- **ddi_add_softintr()**
  - `dip` Pointer to `dev_info` structure.
  - `preference` A hint value describing the type of soft interrupt to generate.
  - `idp` Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.
  - `iblock_cookiep` Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
  - `idevice_cookiep` Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).
  - `int_handler` Pointer to interrupt handler.
  - `int_handler_arg` Argument for interrupt handler.

- **ddi_remove_softintr()**
  - `id` The identifier specifying which soft interrupt handler to remove.

- **ddi_trigger_softintr()**
  - `id` The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.
For `ddi_get_soft_iblock_cookie()`:

`ddi_get_soft_iblock_cookie()` retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before `ddi_add_softintr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the software interrupt routine. `preference` determines which type of soft interrupt to retrieve the cookie for. The possible values for `preference` are:

- `DDI_SOFTINT_LOW`
  - Low priority soft interrupt.
- `DDI_SOFTINT_MED`
  - Medium priority soft interrupt.
- `DDI_SOFTINT_HIGH`
  - High priority soft interrupt.

On a successful return, `iblock_cookiep` contains information needed for initializing locks associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize mutexes acquired by the interrupt routine before calling `ddi_add_softintr()` which prevents a possible race condition where the driver's soft interrupt handler is called immediately after the driver has called `ddi_add_softintr()` but before the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

For `ddi_add_softintr()`:

`ddi_add_softintr()` adds a soft interrupt to the system. The user specified hint `preference` identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for `preference` should be the same as that used in the corresponding call to `ddi_get_soft_iblock_cookie()`. Refer to the description of `ddi_get_soft_iblock_cookie()` above.

The value returned in the location pointed at by `idp` is the soft interrupt identifier. This value is used in later calls to `ddi_remove_softintr()` and `ddi_trigger_softintr()` to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by `iblock_cookiep` is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). Note that the interrupt block cookie is normally obtained using `ddi_get_soft_iblock_cookie()` to avoid the race conditions described above (refer to the description of `ddi_get_soft_iblock_cookie()` above). For this reason, `iblock_cookiep` is no longer useful and should be set to NULL.

`idevice_cookiep` is not used and should be set to NULL.
The routine \texttt{int\_handler}, with its argument \texttt{int\_handler\_arg}, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine \texttt{int\_handler} checks this flag, reachable through \texttt{int\_handler\_arg}, to determine if it should claim the interrupt and do its work.

The interrupt handler must return \texttt{DDI\_INTR\_CLAIMED} if the interrupt was claimed, \texttt{DDI\_INTR\_UNCLAIMED} otherwise.

If successful, \texttt{ddi\_add\_softintr()} will return \texttt{DDI\_SUCCESS}; if the interrupt information cannot be found, it will return \texttt{DDI\_FAILURE}.

For \texttt{ddi\_remove\_softintr()}: \texttt{ddi\_remove\_softintr() \textendash} removes a soft interrupt from the system. The soft interrupt identifier \texttt{id}, which was returned from a call to \texttt{ddi\_add\_softintr()}, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For \texttt{ddi\_trigger\_softintr()}: \texttt{ddi\_trigger\_softintr() \textendash} triggers a soft interrupt. The soft interrupt identifier \texttt{id} is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using \texttt{ddi\_add\_softintr()}.

\textbf{RETURN VALUES} \texttt{ddi\_add\_softintr()} and \texttt{ddi\_get\_soft\_iblock\_cookie()} return:

\begin{itemize}
  \item \texttt{DDI\_SUCCESS} on success
  \item \texttt{DDI\_FAILURE} on failure
\end{itemize}

\textbf{CONTEXT} These functions can be called from user or kernel context. \texttt{ddi\_trigger\_softintr()} may be called from high-level interrupt context as well.

\textbf{EXAMPLES} 1 \texttt{device using high-level interrupts}

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemtable. See \texttt{ddi\_intr\_hilevel(9F)}.
EXAMPLE 1 device using high-level interrupts  

In the example, the high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt handler. If the soft interrupt handler is not currently running, the high-level interrupt routine triggers a soft interrupt so the soft interrupt handler can process the data. Once running, the soft interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data shared between the high-level interrupt handler and the soft interrupt handler. The low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
    ...
    ddi_softintr_t id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t low_mutex;
    int softint_running;
    ...
};
```

```c
struct xxstate *xsp;
static uint_t xxsoftintr(caddr_t);
static uint_t xxhighintr(caddr_t);
... 
```

EXAMPLE 2 sample attach() routine

The following code fragment would usually appear in the driver's attach() routine. ddi_add_intr(9F) is used to add the high-level interrupt handler and ddi_add_softintr() is used to add the low-level interrupt routine.

```
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    ...
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber, &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }
    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
               (void *)&xsp->high_iblock_cookie);
    /* add high-level routine - xxhighintr() */
    if (ddi_add_intr(dip, inumber, NULL, NULL, xxhighintr, (caddr_t)xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }
    ...
}
```
EXAMPLE 2 sample attach() routine  (Continued)

    /* get soft iblock cookie */
    if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
            &xsp->low_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize low-level mutex */
    mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
            (void *)xsp->low_iblock_cookie);

    /* add low level routine - xxsoftintr() */
    if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
            NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }

    ...

}

EXAMPLE 3 High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, ddi_trigger_softintr() is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    ...
    mutex_enter(&xsp->high_mutex);
    /*
    * Verify this device generated the interrupt
    * and disable the device interrupt.
    * Enqueue data for xxsoftintr() processing.
    */

    /* is xxsoftintr() already running ? */
    if (xsp->softint_running)
        need_softint = 0;
    else
        need_softint = 1;
    mutex_exit(&xsp->high_mutex);

    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
EXAMPLE 3 High-level interrupt routine (Continued)

        ddi_trigger_softintr(xsp->id);
        ...
        return (DDI_INTR_CLAIMED);
    }

static uint_t
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    ...
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);
    /* verify there is work to do */
    if (work queue empty || xsp->softint_running ) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }
    xsp->softint_running = 1;
    while ( data on queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));
        /* de-queue data */
        mutex_exit(&xsp->high_mutex);
        /* Process data on queue */
        mutex_enter(&xsp->high_mutex);
        }
    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);
    return (DDI_INTR_CLAIMED);
}

SEE ALSO ddi_add_intr(9F), ddi_in_panic(9F), ddi_intr_hilevel(9F),
        ddi_remove_intr(9F), mutex_init(9F)

Writing Device Drivers

NOTES ddi_add_softintr() may not be used to add the same software interrupt handler
more than once. This is true even if a different value is used for int_handler_arg in each
of the calls to ddi_add_softintr(). Instead, the argument passed to the interrupt
handler should indicate what service(s) the interrupt handler should perform. For
example, the argument could be a pointer to the device’s soft state structure, which could contain a ‘which_service’ field that the handler examines. The driver must set this field to the appropriate value before calling ddi_trigger_softintr().
NAME     ddi_get_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free, 
ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state_p);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
state_p Address of the opaque state pointer which will be initialized by 
        ddi_soft_state_init() to point to implementation dependent data.
size Size of the item which will be allocated by subsequent calls to 
        ddi_soft_state_zalloc().
n_items A hint of the number of items which will be preallocated; zero is allowed.
state An opaque pointer to implementation-dependent data that describes the 
        soft state.
item The item number for the state structure; usually the instance number of the 
        associated devinfo node.

DESCRIPTION
Most device drivers maintain state information with each instance of the device they 
control; for example, a soft copy of a device control register, a mutex that must be held 
while accessing a piece of hardware, a partition table, or a unit structure. These utility 
routines are intended to help device drivers manage the space used by the driver to 
hold such state information.

For example, if the driver holds the state of each instance in a single state structure, 
these routines can be used to dynamically allocate and deallocate a separate structure 
for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the 
implementation uses as a place to hang a set of per-driver structures; everything else is 
managed by these routines.

The routine ddi_soft_state_init() is usually called in the driver’s _init(9E) 
routine to initialize the state pointer, set the size of the soft state structure, and to allow 
the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the driver’s 
attach(9E) routine. The routine is passed an item number which is used to refer to 
the structure in subsequent calls to ddi_get_soft_state() and 
ddi_soft_state_free(). The item number is usually just the instance number of
the `devinfo` node, obtained with `ddi_get_instance(9F)`. The routine attempts to allocate space for the new structure, and if the space allocation was successful, `DDI_SUCCESS` is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling `ddi_get_soft_state()` with the appropriate item number.

The space used by a given soft state structure can be returned to the system using `ddi_soft_state_free()`. This routine is usually called from the driver’s `detach(9E)` entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using `ddi_soft_state_fini()`. This routine can be called from the driver’s `_fini(9E)` routine.

The `ddi_soft_state_zalloc()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

**RETURN VALUES**

`ddi_get_soft_state()`

- **NULL**  The requested state structure was not allocated at the time of the call.
- **pointer**  The pointer to the state structure.

`ddi_soft_state_init()`

- **0**  The allocation was successful.

- **EINVAL**  Either the `size` parameter was zero, or the `state_p` parameter was invalid.

`ddi_soft_state_zalloc()`

- **DDI_SUCCESS**  The allocation was successful.

- **DDI_FAILURE**  The routine failed to allocate the storage required; either the `state` parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

**CONTEXT**

`ddi_soft_state_init()`, `ddi_soft_state_alloc()` can be called from user context only, since they may internally call `kmem_zalloc(9F)` with the `KM_SLEEP` flag.

The `ddi_soft_state_fini()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines can be called from any driver context.
EXAMPLE 1 Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex; /* protects ’csr’ field */
    unsigned int state;
    dev_info_t *dip; /* back pointer to devinfo */
} devstate_t;

static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (error);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ...
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
}

ddi_get_soft_state(9F)
EXAMPLE 1 Creating and Removing Data Structures  (Continued)

static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {
    case DDI_DETACH:
        instance = ddi_get_instance(dip);
        ...
        ddi_soft_state_free(statep, instance);
        return (DDI_SUCCESS);

        default:
        return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
        ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}

SEE ALSO  _fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS  There is no attempt to validate the item parameter given to
ddi_soft_state_zalloc() other than it must be a positive signed integer.
Therefore very large item numbers may cause the driver to hang forever waiting for
virtual memory resources that can never be satisfied.

NOTES  If necessary, a hierarchy of state structures can be constructed by embedding state
pointers in higher order state structures.

DIAGNOSTICS  All of the messages described below usually indicate bugs in the driver and should
not appear in normal operation of the system.
ddi_get_soft_state(9F)

WARNING: ddi_soft_state_zalloc: bad handle
WARNING: ddi_soft_state_free: bad handle
WARNING: ddi_soft_state_fini: bad handle

The implementation-dependent information kept in the state variable is corrupt.
WARNING: ddi_soft_state_free: null handle
WARNING: ddi_soft_state_fini: null handle

The routine has been passed a null or corrupt state pointer. Check that ddi_soft_state_init() has been called.
WARNING: ddi_soft_state_free: item %d not in range [0..<d]

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_get_time(9F)

NAME
  ddi_get_time – returns the current time in seconds

SYNOPSIS
  #include <sys/types.h>
  #include <sys/ddi.h>
  #include <sys/sunddi.h>

  time_t ddi_get_time(void);

INTERFACE
  Solaris DDI specific (Solaris DDI).

LEVEL

DESCRIPTION
  ddi_get_time() returns the current time in seconds since 00:00:00 UTC, January 1, 1970. This value can be used to set of wait or expiration intervals.

RETURN VALUES
  ddi_get_time() returns the time in seconds.

CONTEXT
  This routine can be called from any context.

SEE ALSO
  ddi_get_lbolt(9F), drv_getparm(9F), drv_usectohz(9F)

Writing Device Drivers

STREAMS Programming Guide
ddi_getw(9F)

NAME
ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll –
read data from the mapped memory address, device register or allocated DMA
memory address

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE
level

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle The data access handle returned from setup calls, such as
      ddi_regs_map_setup(9F).

dev_addr Base device address.

DESCRIPTION
The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read
8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address,
dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), elsa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

RETURN VALUES
These functions return the value read from the mapped address.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F)

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
</tbody>
</table>

530  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 22 Nov 1996
## ddi_getw(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_get1</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_get1l</td>
<td>ddi_get64</td>
</tr>
</tbody>
</table>
ddi_in_panic(9F)

NAME ddi_in_panic – determine if system is in panic state

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_in_panic(void);

INTERFACE
Drivers controlling devices on which the system may write a kernel crash dump in the event of a panic can call ddi_in_panic() to determine if the system is panicking.

LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION When the system is panicking, the calls of functions scheduled by timeout(9F) and ddi_trigger_softintr(9F) will never occur. Neither can delay(9F) be relied upon, since it is implemented via timeout(9F).

Drivers that need to enforce a time delay such as SCSI bus reset delay time must busy-wait when the system is panicking.

RETURN VALUES ddi_in_panic() returns 1 if the system is in panic, or 0 otherwise.

CONTEXT ddi_in_panic() may be called from any context.

SEE ALSO dump(9E), delay(9F), ddi_trigger_softintr(9F), timeout(9F)

Writing Device Drivers
ddi_intr_hilevel(9F)

NAME

ddi_intr_hilevel – indicate interrupt handler type

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_hilevel(dev_info_t *dip, uint_t inumber);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dip Pointer to dev_info structure.
inumber Interrupt number.

DESCRIPTION

ddi_intr_hilevel() returns non-zero if the specified interrupt is a "high level" interrupt.

High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler.

In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable.

A typical high level interrupt handler would put data into a circular buffer and schedule a soft interrupt by calling ddi_trigger_softintr(). The circular buffer could be protected by using a mutex that was properly initialized for the interrupt handler.

ddi_intr_hilevel() can be used before calling ddi_add_intr() to decide which type of interrupt handler should be used. Most device drivers are designed with the knowledge that the devices they support will always generate low level interrupts, however some devices, for example those using SBus or VME bus level 6 or 7 interrupts must use this test because on some machines those interrupts are high level (above the scheduler level) and on other machines they are not.

RETURN VALUES

non-zero indicates a high-level interrupt.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

ddi_add_intr(9F), mutex(9F)

Writing Device Drivers
ddi_io_get16(9F)

NAME

ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read
data from the mapped device register in I/O space

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle

Data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr

Device address.

DESCRIPTION

These routines generate a read of various sizes from the device address, dev_addr, in
I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
<tr>
<td>Previous Name</td>
<td>New Name</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>ddi_io_get1</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>
ddi_io_get32(9F)

NAME

ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read data from the mapped device register in I/O space

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Device address.

DESCRIPTION

These routines generate a read of various sizes from the device address, dev_addr, in I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

isa(4), ddi_io_puts8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F),
 ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
 ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
</tbody>
</table>

536  man pages section 9: DDI and DKI Kernel Functions • Last Revised 29 June 1999
### ddi_io_get32(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_get1</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>
ddi_io_get8(9F)

NAME  
ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read data from the mapped device register in I/O space

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE LEVEL
PARAMETERS  
handle Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Device address.

DESCRIPTION  
These routines generate a read of various sizes from the device address, dev_addr, in I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT  
These functions can be called from user, kernel, or interrupt context.

SEE ALSO  
isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES  
For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
</tbody>
</table>
### ddi_io_get8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_get1</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>
ddi_io_getb(9F)

NAME  ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read data from the mapped device register in I/O space

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE LEVEL PARAMETERS  Solaris DDI specific (Solaris DDI).

PARAMETERS  
handle  Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr  Device address.

DESCRIPTION  These routines generate a read of various sizes from the device address, dev_addr, in I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT  These functions can be called from user, kernel, or interrupt context.

SEE ALSO  isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES  For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
</tbody>
</table>
### ddi_io_getb(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getl</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>
ddi_io_getl(9F)

NAME  

ddi_io_get, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl - read data from the mapped device register in I/O space.

SYNOPSIS  

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE LEVEL PARAMETERS  

Solaris DDI specific (Solaris DDI).

PARAMETERS  

handle  

Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr  

Device address.

DESCRIPTION  

These routines generate a read of various sizes from the device address, dev_addr, in I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT  

These functions can be called from user, kernel, or interrupt context.

SEE ALSO  

isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F),

ddi_regs_map_free(9F), ddi_regs_map_setup(9F),

ddi_device_acc_attr(9S)

NOTES  

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
</tbody>
</table>

542 man pages section 9: DDI and DKI Kernel Functions • Last Revised 29 June 1999
## ddi_io_getl(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getl</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers  543
NAME
ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read
data from the mapped device register in I/O space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS
handle Data access handle returned from setup calls, such as
       ddi_regs_map_setup(9F).
dev_addr Device address.

DESCRIPTION
These routines generate a read of various sizes from the device address, dev_addr, in
I/O space. The ddi_io_get8(), ddi_io_get16(), and ddi_io_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F),
       ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
       ddi_device_acc_attr(9S)

NOTES
For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
</tbody>
</table>

544 man pages section 9: DDI and DKI Kernel Functions • Last Revised 29 June 1999
## ddi_io_getw(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getl</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>
ddi_iomin(9F)

NAME    ddi_iomin – find minimum alignment and transfer size for DMA

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

    int ddi_iomin(dev_info_t *dip, int initial, int streaming);

INTERFACE
Parameters
Solaris DDI specific (Solaris DDI).

PARAMETERS

    dip          A pointer to the device’s dev_info structure.
    initial      The initial minimum DMA transfer size in bytes. This may be zero
                  or an appropriate dlim_minxfer value for device’s
                  ddi_dma_lim structure (see ddi_dma_lim_sparc(9S) or
                  ddi_dma_lim_IA(9S)). This value must be a power of two.
    streaming    This argument, if non-zero, indicates that the returned value
                  should be modified to account for streaming mode accesses (see
                  ddi_dma_req(9S) for a discussion of streaming versus
                  non-streaming access mode).

DESCRIPTION

    ddi_iomin(), finds out the minimum DMA transfer size for the device pointed to by
    dip. This provides a mechanism by which a driver can determine the effects of
    underlying caches as well as intervening bus adapters on the granularity of a DMA
    transfer.

RETURN VALUES

    ddi_iomin() returns the minimum DMA transfer size for the calling device, or it
    returns zero, which means that you cannot get there from here.

CONTEXT

    This function can be called from user or interrupt context.

SEE ALSO

    ddi_dma_devalign(9F), ddi_dma_setup(9F), ddi_dma_sync(9F),
    ddi_dma_lim_sparc(9S), ddi_dma_lim_IA(9S), ddi_dma_req(9S)

Writing Device Drivers
ddi_iopb_alloc(9F)

NAME
ddi_iopb_alloc, ddi_iopb_free – allocate and free non-sequentially accessed memory

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_iopb_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, uint_t length, caddr_t *iopbp);

void ddi_iopb_free(caddr_t iopb);

INTERFACE LEVEL
These interfaces are obsolete. Use ddi_dma_mem_alloc(9F) instead of ddi_iopb_alloc(). Use ddi_dma_mem_free(9F) instead of ddi_iopb_free().

ddi_iopb_alloc()

- dip A pointer to the device’s dev_info structure.
- limits A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_IA(9S)). If this pointer is NULL, a default set of DMA limits is assumed.
- length The length in bytes of the desired allocation.
- iopbp A pointer to a caddr_t. On a successful return, *iopbp points to the allocated storage.

ddi_iopb_free()

- iopb The iopb returned from a successful call to ddi_iopb_alloc().

DESCRIPTION
ddi_iopb_alloc() allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using ddi_dma_sync(9F) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as consistent access. The allocation will obey the alignment and padding constraints as specified in the limits argument and other limits imposed by the system.

Note that you still must use DMA resource allocation functions (see ddi_dma_setup(9F)) to establish DMA resources for the memory allocated using ddi_iopb_alloc().

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using ddi_dma_sync(9F) or ddi_dma_free(9F) are still required. The DMA resources will be allocated so that these synchronization steps are as efficient as possible.

ddi_iopb_free() frees up memory allocated by ddi_iopb_alloc().

RETURN VALUES
ddi_iopb_alloc() returns:
- DDI_SUCCESS Memory successfully allocated.
- DDI_FAILURE Allocation failed.

CONTEXT
These functions can be called from user or interrupt context.
ddi_iopb_alloc(9F)

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), ddi_dma_free(9F), ddi_dma_mem_alloc(9F), ddi_dma_mem_free(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_mem_alloc(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers

NOTES

This function uses scarce system resources. Use it selectively.
ddi_iopb_alloc, ddi_iopb_free – allocate and free non-sequentially accessed memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_iopb_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, uint_t length, caddr_t *iopbp);
void ddi_iopb_free(caddr_t iopb);

These interfaces are obsolete. Use ddi_dma_mem_alloc(9F) instead of ddi_iopb_alloc(). Use ddi_dma_mem_free(9F) instead of ddi_iopb_free().

dip
A pointer to the device's dev_info structure.

limits
A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_IA(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

length
The length in bytes of the desired allocation.

iopbp
A pointer to a caddr_t. On a successful return, *iopbp points to the allocated storage.

ddi_iopb_free()

iopb
The iopb returned from a successful call to ddi_iopb_alloc().

ddi_iopb_alloc() allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using ddi_dma_sync(9F) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as consistent access. The allocation will obey the alignment and padding constraints as specified in the limits argument and other limits imposed by the system.

Note that you still must use DMA resource allocation functions (see ddi_dma_setup(9F)) to establish DMA resources for the memory allocated using ddi_iopb_alloc().

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using ddi_dma_sync(9F) or ddi_dma_free(9F) are still required. The DMA resources will be allocated so that these synchronization steps are as efficient as possible.

ddi_iopb_free() frees up memory allocated by ddi_iopb_alloc().

RETURN VALUES

ddi_iopb_alloc() returns:

DDI_SUCCESS Memory successfully allocated.
DDI_FAILURE Allocation failed.

CONTEXT

These functions can be called from user or interrupt context.
ddi_iopb_free(9F)

ATTRIBUTES See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ddi_dma_free(9F), ddi_dma_mem_alloc(9F),
                ddi_dma_mem_free(9F), ddi_dma_setup(9F), ddi_dma_sync(9F),
                ddi_mem_alloc(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S),
                ddi_dma_req(9S)

Writing Device Drivers

NOTES This function uses scarce system resources. Use it selectively.
NAME

`ddi_io_put8`, `ddi_io_put16`, `ddi_io_put32`, `ddi_io_putw`, `ddi_io_putl`, `ddi_io_putb` – write data to the mapped device register in I/O space.

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);

void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);

void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

- `handle` Data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- `dev_addr` Base device address.
- `value` Data to be written to the device.

DESCRIPTION

These routines generate a write of various sizes to the device address, `dev_addr`, in I/O space. The `ddi_io_put8()`, `ddi_io_put16()`, and `ddi_io_put32()` functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

`isa(4)`, `ddi_io_get8(9F)`, `ddi_io_rep_get8(9F)`, `ddi_io_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

NOTES

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_io_putb</code></td>
<td><code>ddi_io_put8</code></td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers  551
### ddi_io_put16(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>
NAME

ddi_io_put8, ddi_io_put16, ddi_io_put32, ddi_io_putw, ddi_io_putl, ddi_io_putb — write data to the mapped device register in I/O space

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr Base device address.
value Data to be written to the device.

DESCRIPTION

These routines generate a write of various sizes to the device address, dev_addr, in I/O space. The ddi_io_put8(), ddi_io_put16(), and ddi_io_put32() functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

isa(4), ddi_io_get8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putb</td>
<td>ddi_io_put8</td>
</tr>
</tbody>
</table>
**ddi_io_put32(9F)**

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>
ddi_io_put8, ddi_io_put16, ddi_io_put32, ddi_io_putw, ddi_io_putl, ddi_io_putb – write data to the mapped device register in I/O space

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
```

Solaris DDI specific (Solaris DDI).

**Handle**
Data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.

**Dev_addr**
Base device address.

**Value**
Data to be written to the device.

These routines generate a write of various sizes to the device address, `dev_addr`, in I/O space. The `ddi_io_put8()`, `ddi_io_put16()`, and `ddi_io_put32()` functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putb</td>
<td>ddi_io_put8</td>
</tr>
</tbody>
</table>
### ddi_io_put8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>

**NAME**

`ddi_io_put8`, `ddi_io_put16`, `ddi_io_put32`, `ddi_io_putw`, `ddi_io_putl`, `ddi_io_putb` – write data to the mapped device register in I/O space

---

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);

void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);

void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
```

**INTERFACE LEVEL PARAMETERS**

- `handle` Data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- `dev_addr` Base device address.
- `value` Data to be written to the device.

**DESCRIPTION**

These routines generate a write of various sizes to the device address, `dev_addr`, in I/O space. The `ddi_io_put8()`, `ddi_io_put16()`, and `ddi_io_put32()` functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`isa(4)`, `ddi_io_get8(9F)`, `ddi_io_rep_get8(9F)`, `ddi_io_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_io_putb</code></td>
<td><code>ddi_io_put8</code></td>
</tr>
</tbody>
</table>
### ddi_io_putb(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>
ddi_io_put8, ddi_io_put16, ddi_io_put32, ddi_io_putw, ddi_io_putl, ddi_io_putb – write data to the mapped device register in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);

These routines generate a write of various sizes to the device address, dev_addr, in I/O space. The ddi_io_put8(), ddi_io_put16(), and ddi_io_put32() functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putb</td>
<td>ddi_io_put8</td>
</tr>
</tbody>
</table>
### ddi_io_putchar(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>
ddi_io_put8, ddi_io_put16, ddi_io_put32, ddi_io_putw, ddi_io_putl, ddi_io_putb – write data to the mapped device register in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);

These routines generate a write of various sizes to the device address, `dev_addr`, in I/O space. The `ddi_io_put8()`, `ddi_io_put16()`, and `ddi_io_put32()` functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putb</td>
<td>ddi_io_put8</td>
</tr>
<tr>
<td>Previous Name</td>
<td>New Name</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>
ddi_io_rep_get16(9F)

NAME
ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw,
ddi_io_rep_getb, ddi_io_rep_getl — read multiple data from the mapped device
register in I/O space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
uint32_t *dev_addr, size_t repcount);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

DESCRIPTION
These routines generate multiple reads from the device address, dev_addr, in I/O
space. repcount data is copied from the device address, dev_addr, to the host address,
host_addr. For each input datum, the ddi_io_rep_get8(), ddi_io_rep_get16(),
and ddi_io_rep_get32() functions read 8 bits, 16 bits, and 32 bits of data,
respectively, from the device address. host_addr must be aligned to the datum
boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

NOTES For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>
ddi_io_rep_get32(9F)

| NAME | ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw, ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device register in I/O space |
| SYNOPSIS | #include <sys/ddi.h> #include <sys/sunddi.h> |
| INTERFACE LEVEL PARAMETERS | handle | The data access handle returned from setup calls, such as ddi_regs_map_setup(9F). |
| | host_addr | Base host address. |
| | dev_addr | Base device address. |
| | repcount | Number of data accesses to perform. |
| DESCRIPTION | These routines generate multiple reads from the device address, dev_addr, in I/O space. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_io_rep_get8(), ddi_io_rep_get16(), and ddi_io_rep_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address. host_addr must be aligned to the datum boundary described by the function. |
| | Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics. |
| CONTEXT | These functions can be called from user, kernel, or interrupt context. |
| SEE ALSO | isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S) |
| NOTES | For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different. |
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>

ddi_io_rep_get32(9F)
ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw, ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device register in I/O space

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount);
void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount);
void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount);
```

**INTERFACE LEVEL PARAMETERS**

- **handle**
  - The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **host_addr**
  - Base host address.
- **dev_addr**
  - Base device address.
- **repcount**
  - Number of data accesses to perform.

**DESCRIPTION**

These routines generate multiple reads from the device address, `dev_addr`, in I/O space. `repcount` data is copied from the device address, `dev_addr`, to the host address, `host_addr`. For each input datum, the `ddi_io_rep_get8()`, `ddi_io_rep_get16()`, and `ddi_io_rep_get32()` functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address. `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`isa(4)`, `ddi_io_get8(9F)`, `ddi_io_put8(9F)`, `ddi_io_rep_put8(9F)`, `ddi_regs_map_free(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>
ddi_io_rep_getb(9F)

#### NAME

ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw,
ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device
register in I/O space

#### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
                      uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
                       uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
                       uint32_t *dev_addr, size_t repcount);
```

#### INTERFACE LEVEL PARAMETERS

- **handle**: The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.

#### DESCRIPTION

These routines generate multiple reads from the device address, *dev_addr*, in I/O
space. repcount data is copied from the device address, *dev_addr*, to the host address,
*host_addr*. For each input datum, the ddi_io_rep_get8(), ddi_io_rep_get16(),
and ddi_io_rep_get32() functions read 8 bits, 16 bits, and 32 bits of data,
respectively, from the device address. *host_addr* must be aligned to the datum
boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

#### CONTEXT

These functions can be called from user, kernel, or interrupt context.

#### SEE ALSO

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

#### NOTES

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.
ddi_io_rep_getb(9F)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>

Last Revised 29 June 1999
### NAME
ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw,
ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device
register in I/O space

### SYNOPSIS
```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
                      uint8_t *dev_addr, , size_t repcount);
void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
                       uint16_t *dev_addr, , size_t repcount);
void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
                       uint32_t *dev_addr, , size_t repcount);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **handle**
  The data access handle returned from setup calls, such as
  ddi_regs_map_setup(9F).
- **host_addr**
  Base host address.
- **dev_addr**
  Base device address.
- **repcount**
  Number of data accesses to perform.

### DESCRIPTION
These routines generate multiple reads from the device address, `dev_addr`, in I/O
space. `repcount` data is copied from the device address, `dev_addr`, to the host address,
`host_addr`. For each input datum, the `ddi_io_rep_get8()`, `ddi_io_rep_get16()`,
and `ddi_io_rep_get32()` functions read 8 bits, 16 bits, and 32 bits of data,
respectively, from the device address. `host_addr` must be aligned to the datum
boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

### CONTEXT
These functions can be called from user, kernel, or interrupt context.

### SEE ALSO
isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

### NOTES
For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>

ddi_io_rep_getl(9F)
ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw,
/ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device
register in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
uint32_t *dev_addr, size_t repcount);

Solaris DDI specific (Solaris DDI).

INTERFACE

PARAMETERS

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

DESCRIPTION

These routines generate multiple reads from the device address, dev_addr, in I/O
space. repcount data is copied from the device address, dev_addr, to the host address,
host_addr. For each input datum, the ddi_io_rep_get8(), ddi_io_rep_get16(),
and ddi_io_rep_get32() functions read 8 bits, 16 bits, and 32 bits of data,
respectively, from the device address. host_addr must be aligned to the datum
boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>
**NAME**

`ddi_io_rep_put8`, `ddi_io_rep_put16`, `ddi_io_rep_put32`, `ddi_io_rep_putw`,
`ddi_io_rep_putl`, `ddi_io_rep_putb` – write multiple data to the mapped device register in I/O space

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uin8_t *dev_addr, size_t repcount);
void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uin16_t *dev_addr, size_t repcount);
void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uin32_t *dev_addr, size_t repcount);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**PARAMETERS**

- `handle` Data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- `host_addr` Base host address.
- `dev_addr` Base device address.
- `repcount` Number of data accesses to perform.

**DESCRIPTION**

These routines generate multiple writes to the device address, `dev_address`, in I/O space. `repcount` data is copied from the host address, `host_addr`, to the device address, `dev_addr`. For each input datum, the `ddi_io_rep_put8()`, `ddi_io_rep_put16()`, and `ddi_io_rep_put32()` functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address. `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`isa(4)`, `ddi_io_get8(9F)`, `ddi_io_put8(9F)`, `ddi_io_rep_get8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)` but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
NAME


SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount);
void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount);
void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle Data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.

DESCRIPTION

These routines generate multiple writes to the device address, dev_address, in I/O space. repcount data is copied from the host address, host_addr, to the device address, dev_addr. For each input datum, the ddi_io_rep_put8(), ddi_io_rep_put16(), and ddi_io_rep_put32() functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address. host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as IA where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
### ddi_io_rep_put32(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
ddi_io_rep_put8(9F)

NAME
ddi_io_rep_put8, ddi_io_rep_put16, ddi_io_rep_put32, ddi_io_rep_putw,
ddi_io_rep_putl, ddi_io_rep_putb – write multiple data to the mapped device register
in I/O space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
                     uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
                      uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
                      uint32_t *dev_addr, size_t repcount);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle Data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

DESCRIPTION
These routines generate multiple writes to the device address, dev_address, in I/O
space. repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_io_rep_put8(), ddi_io_rep_put16(),
and ddi_io_rep_put32() functions write 8 bits, 16 bits, and 32 bits of data,
respectively, to the device address. host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
ddi_io_rep_put8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
ddi_io_rep_put8, ddi_io_rep_put16, ddi_io_rep_put32, ddi_io_rep_putw,
ddi_io_rep_putl, ddi_io_rep_putb – write multiple data to the mapped device register
in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
uin8_t *dev_addr, size_t repcount);

void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
uin16_t *dev_addr, size_t repcount);

void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
uin32_t *dev_addr, size_t repcount);

Solaris DDI specific (Solaris DDI).

handle Data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

These routines generate multiple writes to the device address, dev_address, in I/O
space. repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_io_rep_put8(), ddi_io_rep_put16(),
and ddi_io_rep_put32() functions write 8 bits, 16 bits, and 32 bits of data,
respectively, to the device address. host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
ddi_io_rep_put8, ddi_io_rep_put16, ddi_io_rep_put32, ddi_io_rep_putw,
ddi_io_rep_putl, ddi_io_rep_putb - write multiple data to the mapped device register
in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
uint32_t *dev_addr, size_t repcount);

Solaris DDI specific (Solaris DDI).

handle Data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

These routines generate multiple writes to the device address, dev_address, in I/O
space. repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_io_rep_put8(), ddi_io_rep_put16(),
and ddi_io_rep_put32() functions write 8 bits, 16 bits, and 32 bits of data,
respectively, to the device address. host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
ddi_io_rep_putl(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
ddi_io_rep_put8, ddi_io_rep_put16, ddi_io_rep_put32, ddi_io_rep_putw,
ddi_io_rep_putl, ddi_io_rep_putb – write multiple data to the mapped device register
in I/O space

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
uin8_t *dev_addr, size_t repcount);

void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
uin16_t *dev_addr, size_t repcount);

void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
uin32_t *dev_addr, size_t repcount);

Solaris DDI specific (Solaris DDI).

handle
host_addr
dev_addr
repcount

Data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
Base host address.
Base device address.
Number of data accesses to perform.

These routines generate multiple writes to the device address, dev_address, in I/O
space. repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_io_rep_put8(), ddi_io_rep_put16(),
and ddi_io_rep_put32() functions write 8 bits, 16 bits, and 32 bits of data,
respectively, to the device address. host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

For drivers using these functions, it may not be easy to maintain a single source to
support devices with multiple bus versions. For example, devices may offer I/O space
in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true
in instruction set architectures such as IA where accesses to the memory and I/O
space are different.

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
### ddi_io_rep_putw(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
ddi_log_sysevent(9F)

**NAME**

ddi_log_sysevent – log system event for drivers

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_log_sysevent(dev_info_t *dip, char *vendor, char *class, char *subclass, nvlist_t *attr_list, sysevent_id_t *eidp, int sleep_flag);
```

**INTERFACE LEVEL PARAMETERS**

- **dip**: A pointer to the dev_info node for this driver.
- **vendor**: A pointer to a string defining the vendor. Third-party drivers should use their company’s stock symbol (or similarly enduring identifier). Sun-supplied drivers should use DDI_VENDOR_SUNW.
- **class**: A pointer to a string defining the event class.
- **subclass**: A pointer to a string defining the event subclass.
- **attr_list**: A pointer to an nvlist_t, listing the name-value attributes associated with the event or NULL if there are no such attributes for this event.
- **eidp**: The address of a sysevent_id_t structure in which the event’s sequence number and timestamp are returned if the event is successfully queued. May be NULL if this information is not of interest. See below for the definition of sysevent_id_t.
- **sleep_flag**: Indicates how a caller wants to handle the possibility of resources not being available. If sleep_flag is DDI_NOSLEEP, the caller does not care if the allocation fails or the queue is full and can handle a failure appropriately. If sleep_flag is DDI_SLEEP, the caller wishes to have the allocation and queuing routines wait for resources to become available.

**DESCRIPTION**

ddi_log_sysevent() causes a system event, of the specified class and subclass, to be generated on behalf of the driver and queued for delivery to syseventd, the user-land sysevent daemon.

The publisher string for the event is constructed using the vendor name and driver name, with the format:

```
*<vendor>:kern:<driver-name>*
```

The two fields of eidp, eid_seq and eid_ts, are sufficient to uniquely identify an event.

**STRUCTURE MEMBERS**

The structure members of sysevent_id_t are:

```c
uint64_t eid_seq; /* sysevent sequence number */
hrttime_t eid_ts; /* sysevent timestamp */
```

**RETURN VALUES**

ddi_log_sysevent() returns:
ddi_log_sysevent(9F)

DDI_SUCCESS
The event has been queued for delivery successfully.

DDI_ENOMEM
There is not enough memory to queue the system event at this time. DDI_ENOMEM cannot be returned when sleep_flag is DDI_SLEEP.

DDI_EBUSY
The system event queue is full at this time. DDI_EBUSY cannot be returned when sleep_flag is DDI_SLEEP.

DDI_ETRANSPORT
The syseventd daemon is not responding and events cannot be queued or delivered at this time. DDI_ETRANSPORT can be returned even when sleep_flag is DDI_SLEEP.

DDI_ECONTEXT
sleep_flag is DDI_SLEEP and the driver is running in interrupt context.

ddi_log_sysevent() can be called from user or interrupt context, except when sleep_flag is DDI_SLEEP, in which case it can be called from user context only.

EXAMPLE 1 Logging System Event with No Attributes

```c
if (ddi_log_sysevent(dip, DDI_VENDOR_SUNW, "class", "subclass", NULL, NULL, DDI_SLEEP) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "error logging system event\n");
}
```

EXAMPLE 2 Logging System Event with Two Name/Value Attributes, an Integer and a String

```c
nvlist_t *attr_list;
sysevent_id_t eid;

if (nvlist_alloc(&attr_list, NV_UNIQUE_NAME_TYPE, NV_FLAG_KMSLEEP) == 0)
{
    int int_value = 100;
    char str_value[] = "Sample Text";

    err = nvlist_add_uint32(attr_list, int_name, int_value);
    if (err == 0)
        err = nvlist_add_string(attr_list, str_name, str_value);
    if (err == 0)
        err = ddi_log_sysevent(dip, DDI_VENDOR_SUNW, "class", "subclass", attr_list, &eid, DDI_SLEEP);
    if (err != DDI_SUCCESS)
        cmn_err(CE_WARN, "error logging system event\n");
    nvlist_free(attr_list);
}
```

EXAMPLE 3 Use Timeout to Handle nvlist and System Event Resource Allocation Failures

Since no blocking calls are made, this example would be useable from a driver needing to generate an event from interrupt context.

```c
static int
xx_se_timeout_handler(xx_state_t *xx)
{
```
EXAMPLE 3 Use Timeout to Handle nvlist and System Event Resource Allocation Failures  (Continued)

```c
xx->xx_timeoutid = (xx_generate_event(xx) ?
    timeout(xx_se_timeout_handler, xx, 4) : 0);
```

```c
static int
xx_generate_event(xx_state_t *xx)
{
    int err;

    err = nvlist_alloc(&xx->xx_ev_attrlist, NV_UNIQUE_NAME_TYPE, 0);
    if (err != 0)
        return (1);
    err = nvlist_add_uint32(&xx->xx_ev_attrlist,
        xx->xx_ev_name, xx->xx_ev_value);
    if (err != 0) {
        nvlist_free(xx->xx_ev_attrlist);
        return(1);
    }
    err = ddi_log_sysevent(xx->xx_dip, DDI_VENDOR_SUNW,
        xx->xx_ev_class, xx->xx_ev_sbclass,
        xx->xx_ev_attrlist, NULL, DDI_NOSLEEP);
    nvlist_free(xx->xx_ev_attrlist);
    if (err == DDI_SUCCESS || err == DDI_ETRANSPORT) {
        if (err == DDI_ETRANSPORT)
            cmn_err(CE_WARN, "cannot log system event\n");
        return (0);
    }
    return (1);
}
```

SEE ALSO syseventd(1M), attributes(5), nvlist_add_boolean(9F), nvlist_alloc(9F)

Writing Device Drivers
**ddi_mapdev(9F)**

**NAME**

`ddi_mapdev` – create driver-controlled mapping of device

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_mapdev(dev_t dev, off_t offset, struct asp *asp,
                caddr_t *addrp, off_t len, uint_t prot, uint_t maxprot,
                uint_t flags, cred_t *cred,
                struct ddi_mapdev_ctl *ctl, ddi_mapdev_handle_t *handlep, void *
devprivate);
```

**INTERFACE LEVEL PARAMETERs**

- `dev` The device whose memory is to be mapped.
- `offset` The offset within device memory at which the mapping begins.
- `asp` An opaque pointer to the user address space into which the device memory should be mapped.
- `addrp` Pointer to the starting address within the user address space to which the device memory should be mapped.
- `len` Length (in bytes) of the memory to be mapped.
- `prot` A bit field that specifies the protections.
- `maxprot` Maximum protection flag possible for attempted mapping.
- `flags` Flags indicating type of mapping.
- `cred` Pointer to the user credentials structure.
- `ctl` A pointer to a `ddi_mapdev_ctl(9S)` structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.
- `handlep` An opaque pointer to a device mapping handle. A handle to the new device mapping is generated and placed into the location pointed to by `handlep`. If the call fails, the value of `handlep` is undefined.
- `devprivate` Driver private mapping data. This value is passed into each mapping call back routine.

**DESCRIPTION**

This interface is obsolete. `devmap_setup(9F)` should be used instead.

Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use `devmap_setup(9F)` instead. See `devmap_setup(9F)` for details.

`ddi_mapdev()` sets up user mappings to device space. The driver is notified of user events on the mappings via the entry points defined by `ctl`.

The user events that the driver is notified of are:

- `access` User has accessed an address in the mapping that has no translations.
duplication User has duplicated the mapping. Mappings are duplicated when the process calls `fork(2)`.

unmapping User has called `munmap(2)` on the mapping or is exiting.

See `mapdev_access(9E)`, `mapdev_dup(9E)`, and `mapdev_free(9E)` for details on these entry points.

The range to be mapped, defined by `offset` and `len` must be valid.

The arguments `dev`, `asp`, `addrp`, `len`, `prot`, `maxprot`, `flags`, and `cred` are provided by the `segmap(9E)` entry point and should not be modified. See `segmap(9E)` for a description of these arguments. Unlike `ddi_segmap(9F)`, the drivers `mmap(9E)` entry point is not called to verify the range to be mapped.

With the handle, device drivers can use `ddi_mapdev_intercept(9F)` and `ddi_mapdev_nointercept(9F)` to inform the system of whether or not they are interested in being notified when the user process accesses the mapping. By default, user accesses to newly created mappings will generate a call to the `mapdev_access()` entry point. The driver is always notified of duplications and unmaps.

The device may also use the handle to assign certain characteristics to the mapping. See `ddi_mapdev_set_device_acc_attr(9F)` for details.

The device driver can use these interfaces to implement a device context and control user accesses to the device space. `ddi_mapdev()` is typically called from the `segmap(9E)` entry point.

**RETURN VALUES**
`ddi_mapdev()` returns zero on success and non-zero on failure. The return value from `ddi_mapdev()` should be used as the return value for the drivers `segmap()` entry point.

**CONTEXT**
This routine can be called from user or kernel context only.

**ATTRIBUTES**
See `attributes(5)` for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`fork(2)`, `mmap(2)`, `munmap(2)`, `attributes(5)`, `mapdev_access(9E)`, `mapdev_dup(9E)`, `mapdev_free(9E)`, `mmap(9E)`, `segmap(9E)`, `devmap_setup(9F)`
`ddi_mapdev_intercept(9F)`, `ddi_mapdev_nointercept(9F)`
`ddi_mapdev_set_device_acc_attr(9F)`, `ddi_segmap(9F)`
`ddi_mapdev_ctl(9S)`

*Writing Device Drivers*
Only mappings of type MAP_PRIVATE should be used with `ddi_mapdev()`.
ddi_mapdev_intercept(9F)

NAME

ddi_mapdev_intercept, ddi_mapdev_nointercept – control driver notification of user accesses

SYNOPSIS

```c
#include <sys/sunddi.h>

int ddi_mapdev_intercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);
int ddi_mapdev_nointercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);
```

INTERFACE LEVEL

These interfaces are obsolete. Use devmap_load(9F) instead of ddi_mapdev_intercept(). Use devmap_unload(9F) instead of ddi_mapdev_nointercept().

PARAMETERS

- `handle`: An opaque pointer to a device mapping handle.
- `offset`: An offset in bytes within device memory.
- `len`: Length in bytes.

DESCRIPTION

Future releases of Solaris will provide these functions for binary and source compatibility. However, for increased functionality, use devmap_load(9F) or devmap_unload(9F) instead. See devmap_load(9F) and devmap_unload(9F) for details.

The ddi_mapdev_intercept() and ddi_mapdev_nointercept() functions control whether or not user accesses to device mappings created by ddi_mapdev(9F) in the specified range will generate calls to the mapdev_access(9E) entry point. ddi_mapdev_intercept() tells the system to intercept the user access and notify the driver to invalidate the mapping translations. ddi_mapdev_nointercept() tells the system to not intercept the user access and allow it to proceed by validating the mapping translations.

For both routines, the range to be affected is defined by the `offset` and `len` arguments. Requests affect the entire page containing the `offset` and all pages up to and including the page containing the last byte as indicated by `offset + len`.

Supplying a value of 0 for the `len` argument affects all addresses from the `offset` to the end of the mapping. Supplying a value of 0 for the `offset` argument and a value of 0 for `len` argument affect all addresses in the mapping.

To manage a device context, a device driver would call ddi_mapdev_intercept() on the context about to be switched out, switch contexts, and then call ddi_mapdev_nointercept() on the context switched in.

RETURN VALUES

ddi_mapdev_intercept() and ddi_mapdev_nointercept() return the following values:

- 0: Successful completion.
- Non-zero: An error occurred.
EXAMPLE 1 managing a device context that is one page in length

The following shows an example of managing a device context that is one page in length.

```c
ddi_mapdev_handle_t cur_hdl;
static int
xxmapdev_access(ddi_mapdev_handle_t handle, void *devprivate,
    off_t offset)
{
    int err;
    /* enable access callbacks for the current mapping */
    if (cur_hdl != NULL) {
        if ((err = ddi_mapdev_intercept(cur_hdl, offset, 0)) != 0)
            return (err);
    }
    /* Switch device context - device dependent*/
    ...
    /* Make handle the new current mapping */
    cur_hdl = handle;
    /*
    * Disable callbacks and complete the access for the
    * mapping that generated this callback.
    */
    return (ddi_mapdev_nointercept(handle, offset, 0));
}
```

CONTEXT These routines can be called from user or kernel context only.

ATTRIBUTES See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), mapdev_access(9E), devmap_load(9F), ddi_mapdev(9F)

Writing Device Drivers
ddi_mapdev_intercept, ddi_mapdev_nointercept – control driver notification of user accesses

```
#include <sys/sunddi.h>

int ddi_mapdev_intercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);
int ddi_mapdev_nointercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);
```

These interfaces are obsolete. Use devmap_load(9F) instead of ddi_mapdev_intercept(). Use devmap_unload(9F) instead of ddi_mapdev_nointercept().

**INTERFACE LEVEL**

**PARAMETERS**

- **handle**: An opaque pointer to a device mapping handle.
- **offset**: An offset in bytes within device memory.
- **len**: Length in bytes.

**DESCRIPTION**

Future releases of Solaris will provide these functions for binary and source compatibility. However, for increased functionality, use devmap_load(9F) or devmap_unload(9F) instead. See devmap_load(9F) and devmap_unload(9F) for details.

The `ddi_mapdev_intercept()` and `ddi_mapdev_nointercept()` functions control whether or not user accesses to device mappings created by `ddi_mapdev(9F)` in the specified range will generate calls to the `mapdev_access(9E)` entry point. `ddi_mapdev_intercept()` tells the system to intercept the user access and notify the driver to invalidate the mapping translations. `ddi_mapdev_nointercept()` tells the system to not intercept the user access and allow it to proceed by validating the mapping translations.

For both routines, the range to be affected is defined by the **offset** and **len** arguments. Requests affect the entire page containing the **offset** and all pages up to and including the page containing the last byte as indicated by **offset** + **len**.

Supplying a value of 0 for the **len** argument affects all addresses from the **offset** to the end of the mapping. Supplying a value of 0 for the **offset** argument and a value of 0 for **len** argument affect all addresses in the mapping.

To manage a device context, a device driver would call `ddi_mapdev_intercept()` on the context about to be switched out, switch contexts, and then call `ddi_mapdev_nointercept()` on the context switched in.

**RETURN VALUES**

`ddi_mapdev_intercept()` and `ddi_mapdev_nointercept()` return the following values:

- **0**: Successful completion.
- **Non-zero**: An error occurred.
EXAMPLE 1  managing a device context that is one page in length

The following shows an example of managing a device context that is one page in length.

```c
ddi_mapdev_handle_t cur_hdl;
static int
xxmapdev_access(ddi_mapdev_handle_t handle, void *devprivate,
    off_t offset)
{
    int err,
        /* enable access callbacks for the current mapping */
        if (cur_hdl != NULL) {
            if ((err = ddi_mapdev_intercept(cur_hdl, offset, 0)) != 0)
                return (err);
        }
        /* Switch device context - device dependent*/
        ...
        /* Make handle the new current mapping */
        cur_hdl = handle;
        /*
        * Disable callbacks and complete the access for the
        * mapping that generated this callback.
        */
        return (ddi_mapdev_nointercept(handle, offset, 0));
}
```

These routines can be called from user or kernel context only.

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), mapdev_access(9E), devmap_load(9F), ddi_mapdev(9F)

Writing Device Drivers
ddi_mapdev_set_device_acc_attr

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_mapdev_set_device_acc_attr(ddi_mapdev_handle_t
mapping_handle, off_t offset, off_t len, ddi_device_acc_attr_t
*accattrp, uint_t rnumber);

PARAMETERS

mapping_handle A pointer to a device mapping handle.

offset The offset within device memory to which the device access
attributes structure applies.

len Length (in bytes) of the memory to which the device access
attributes structure applies.

*accattrp Pointer to a ddi_device_acc_attr(9S) structure. Contains the
device access attributes to be applied to this range of memory.

rnumber Index number to the register address space set.

DESCRIPTION
Future releases of Solaris will provide this function for binary and source
compatibility. However, for increased functionality, use devmap(9E) instead. See
devmap(9E) for details.

The ddi_mapdev_set_device_acc_attr() function assigns device access
attributes to a range of device memory in the register set given by rnumber.

*accattrp defines the device access attributes. See ddi_device_acc_attr(9S) for
more details.

mapping_handle is a mapping handle returned from a call to ddi_mapdev(9F).

The range to be affected is defined by the offset and len arguments. Requests affect the
entire page containing the offset and all pages up to and including the page containing
the last byte as indicated by offset+len. Supplying a value of 0 for the len argument
affects all addresses from the offset to the end of the mapping. Supplying a value of 0
for the offset argument and a value of 0 for the len argument affect all addresses in the
mapping.

RETURN VALUES
The ddi_mapdev_set_device_acc_attr() function returns the following values:

DDI_SUCCESS The attributes were successfully set.

DDI_FAILURE It is not possible to set these attributes for this mapping
handle.

CONTEXT This routine can be called from user or kernel context only.
Writing Device Drivers

SEE ALSO: segmap(9E), ddi_mapdev(9F), ddi_segmap_setup(9F), ddi_device_acc_attr(9S)
# ddi_map_regs

## SYNOPSIS

```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_map_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp, off_t offset, off_t len);

void ddi_unmap_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp, off_t offset, off_t len);
```

## INTERFACE LEVEL

These interfaces are obsolete. Use `ddi_regs_map_setup(9F)` instead of `ddi_map_regs()`. Use `ddi_regs_map_free(9F)` instead of `ddi_unmap_regs()`.

### ddi_map_regs()

- **dip**: Pointer to the device’s dev_info structure.
- **rnumber**: Register set number.
- **kaddrp**: Pointer to the base kernel address of the mapped region (set on return).
- **offset**: Offset into register space.
- **len**: Length to be mapped.

### ddi_unmap_regs()

- **dip**: Pointer to the device’s dev_info structure.
- **rnumber**: Register set number.
- **kaddrp**: Pointer to the base kernel address of the region to be unmapped.
- **offset**: Offset into register space.
- **len**: Length to be unmapped.

## DESCRIPTION

`ddi_map_regs()` maps in the register set given by `rnumber`. The register number determines which register set will be mapped if more than one exists. The base kernel virtual address of the mapped register set is returned in `kaddrp`. `offset` specifies an offset into the register space to start from and `len` indicates the size of the area to be mapped. If `len` is non-zero, it overrides the length given in the register set description. See the discussion of the `reg` property in `sbus(4)` and for more information on register set descriptions. If `len` and `offset` are 0, the entire space is mapped.

`ddi_unmap_regs()` undoes mappings set up by `ddi_map_regs()`. This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to `ddi_unmap_regs()` must correspond to a previous call to `ddi_map_regs()`). Releasing portions of previous mappings is not allowed. `rnumber` determines which register set will be unmapped if more than one exists. The `kaddrp`, `offset` and `len` specify the area to be unmapped. `kaddrp` is a pointer to the address returned from `ddi_map_regs()`: `offset` and `len` should match what `ddi_map_regs()` was called with.
ddi_map_regs(9F)

**RETURN VALUES**

ddi_map_regs() returns:

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

attributes(5), sbus(4), ddi_regs_map_free(9F), ddi_regs_map_setup(9F)

*Writing Device Drivers*
ddi_mem_alloc(9F)

NAME

ddi_mem_alloc, ddi_mem_free – allocate and free sequentially accessed memory

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_mem_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, uint_t length, uint_t flags, caddr_t *kaddrp, uint_t *real_length);

void ddi_mem_free(caddr_t kaddr);

INTERFACE

These interfaces are obsolete. ddi_dma_mem_alloc(9F) and ddi_dma_mem_free(9F) should be used instead.

ddi_mem_alloc()

dip
A pointer to the device’s dev_info structure.

limits
A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_IA(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

length
The length in bytes of the desired allocation.

flags
The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.

kaddrp
On a successful return, *kaddrp points to the allocated memory.

real_length
The length in bytes that was allocated. Alignment and padding requirements may cause ddi_mem_alloc() to allocate more memory than requested in length.

ddi_mem_free()

kaddr
The memory returned from a successful call to ddi_mem_alloc().

DESCRIPTION

ddi_mem_alloc() allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as streaming access. The allocation will obey the alignment and padding constraints as specified by the limits argument and other limits imposed by the system.

Note that you must still use DMA resource allocation functions (see ddi_dma_setup(9F)) to establish DMA resources for the memory allocated using ddi_mem_alloc(). ddi_mem_alloc() returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. ddi_dma_setup(9F) requires the actual length.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using ddi_dma_sync(9F) or ddi_dma_free(9F) are required.

ddi_mem_free() frees up memory allocated by ddi_mem_alloc().

RETURN VALUES

ddi_mem_alloc() returns:

DDI_SUCCESS Memory successfully allocated.
ddi_mem_alloc(9F)

DDI_FAILURE  Allocation failed.

CONTEXT  ddi_mem_alloc() can be called from user or interrupt context, except when flags is set to 1, in which case it can be called from user context only.

ATTRIBUTES  See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5), ddi_dma_free(9F), ddi_dma_mem_alloc(9F),
           ddi_dma_mem_free(9F), ddi_dma_setup(9F), ddi_dma_sync(9F),
           ddi_iopb_alloc(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S),
           ddi_dma_req(9S)

Writing Device Drivers
ddi_mem_free(9F)

NAME

ddi_mem_alloc, ddi_mem_free – allocate and free sequentially accessed memory

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_mem_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, uint_t length, uint_t flags, caddr_t *kaddrp, uint_t *real_length);

void ddi_mem_free(caddr_t kaddr);

INTERFACE LEVEL

These interfaces are obsolete. ddi_dma_mem_alloc(9F) and ddi_dma_mem_free(9F) should be used instead.

ddi_mem_alloc()

dip A pointer to the device’s dev_info structure.

limits A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_IA(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

length The length in bytes of the desired allocation.

flags The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.

kaddrp On a successful return, *kaddrp points to the allocated memory.

real_length The length in bytes that was allocated. Alignment and padding requirements may cause ddi_mem_alloc() to allocate more memory than requested in length.

ddi_mem_free()

kaddr The memory returned from a successful call to ddi_mem_alloc().

DESCRIPTION

ddi_mem_alloc() allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as streaming access. The allocation will obey the alignment and padding constraints as specified by the limits argument and other limits imposed by the system.

Note that you must still use DMA resource allocation functions (see ddi_dma_setup(9F)) to establish DMA resources for the memory allocated using ddi_mem_alloc(). ddi_mem_alloc() returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. ddi_dma_setup(9F) requires the actual length.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using ddi_dma_sync(9F) or ddi_dma_free(9F) are required.

ddi_mem_free() frees up memory allocated by ddi_mem_alloc().

RETURN VALUES

ddi_mem_alloc() returns:

DDI_SUCCESS Memory successfully allocated.
ddi_mem_free(9F)

DDI_FAILURE Allocation failed.

CONTEXT ddi_mem_alloc() can be called from user or interrupt context, except when flags is set to 1, in which case it can be called from user context only.

ATTRIBUTES See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ddi_dma_free(9F), ddi_dma_mem_alloc(9F), ddi_dma_mem_free(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_iopb_alloc(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
NAME
ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory.

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr Base device address.

DESCRIPTION
These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9F)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
**NAME**

ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

**INTERFACE LEVEL PARAMETERS**

- **handle** The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **dev_addr** Base device address.

**DESCRIPTION**

These routines generate a read of various sizes from memory space or allocated DMA memory. The `ddi_mem_get8()`, `ddi_mem_get16()`, `ddi_mem_get32()`, and `ddi_mem_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Base device address.

These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

ddi_mem_get8(9F), ddi_mem_get16(9F), ddi_mem_get32(9F), ddi_mem_get64(9F), ddi_reg_map_setup(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
ddi_mem_get8(9F)

NAME

ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw,
ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the
memory space or allocated DMA memory

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

The data access handle returned from setup calls, such as

PARAMETERS

ddi_regs_map_setup(9F).

dev_addr

Base device address.

DESCRIPTION

These routines generate a read of various sizes from memory space or allocated DMA
memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and
ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data,
respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>

man pages section 9: DDI and DKI Kernel Functions • Last Revised 30 Sep 1996
ddi_mem_get8(9F)

NAME

ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw,
ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the
memory space or allocated DMA memory

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
dev_addr Base device address.

DESCRIPTION

These routines generate a read of various sizes from memory space or allocated DMA
memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and
ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data,
respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F),
ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Base device address.

These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>

NOTES

610 man pages section 9: DDI and DKI Kernel Functions • Last Revised 30 Sep 1996
ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Base device address.

These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
**NAME**

ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb – read data from mapped device in the memory space or allocated DMA memory

**SYNOPSIS**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

**INTERFACE LEVEL PARAMETERS**

- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **dev_addr**: Base device address.

**DESCRIPTION**

These routines generate a read of various sizes from memory space or allocated DMA memory. The `ddi_mem_get8()`, `ddi_mem_get16()`, `ddi_mem_get32()`, and `ddi_mem_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_put8(9F)`, `ddi_mem_rep_get8(9F)`, `ddi_mem_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>

Solaris DDI specific (Solaris DDI).
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb, ddi_mem_putw, ddi_mem_putl, ddi_mem_putll – write data to mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr Base device address.
value The data to be written to the device.

These routines generate a write of various sizes to memory space or allocated DMA memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

These functions can be called from user, kernel, or interrupt context.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>

Solaris DDI specific (Solaris DDI).

SEE ALSO

ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)
**ddi_mem_put16(9F)**

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
NAME  
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb,
ddi_mem_putw, ddi_mem_putl, ddi_mem_putll - write data to mapped device in the
memory space or allocated DMA memory

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr,
                   uint8_t value);

void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr,
                    uint16_t value);

void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr,
                    uint32_t value);

void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr,
                    uint64_t value);

PARAMETERS
handle       The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr     Base device address.

value        The data to be written to the device.

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

DESCRIPTION
These routines generate a write of various sizes to memory space or allocated DMA
memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and
ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data,
respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
### ddi_mem_put32(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
NAME  
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb,
ddi_mem_putw, ddi_mem_putl, ddi_mem_putll – write data to mapped device in the
memory space or allocated DMA memory

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, 
                  uint8_t value);

void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, 
                   uint16_t value);

void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, 
                   uint32_t value);

void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, 
                   uint64_t value);

PARAMETERS  
handle      The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr    Base device address.

value       The data to be written to the device.

INTERFACE LEVEL DESCRIPTION  
Solaris DDI specific (Solaris DDI).

These routines generate a write of various sizes to memory space or allocated DMA
memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and
ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data,
respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT  
These functions can be called from user, kernel, or interrupt context.

SEE ALSO  
ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F),
ddi_device_acc_attr(9S)

NOTES  
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
ddi_mem_put64(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
**NAME**

ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb, ddi_mem_putw, ddi_mem_putl, ddi_mem_putll – write data to mapped device in the memory space or allocated DMA memory

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);
```

**PARAMETERS**

- `handle` – The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- `dev_addr` – Base device address.
- `value` – The data to be written to the device.

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**DESCRIPTION**

These routines generate a write of various sizes to memory space or allocated DMA memory. The `ddi_mem_put8()`, `ddi_mem_put16()`, `ddi_mem_put32()`, and `ddi_mem_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, `dev_addr`, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
ddi_mem_put8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
NAME | ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb, ddi_mem_putw, ddi_mem_putl, ddi_mem_putll - write data to mapped device in the memory space or allocated DMA memory

SYNOPSIS | 
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t *value);
void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t *value);
void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t *value);
void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t *value);

PARAMETERS | handle | The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr | Base device address.
value | The data to be written to the device.

INTERFACE LEVEL | Solaris DDI specific (Solaris DDI).

DESCRIPTION | These routines generate a write of various sizes to memory space or allocated DMA memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT | These functions can be called from user, kernel, or interrupt context.

SEE ALSO | ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES | The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
### ddi_mem_putb(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb,
 ddi_mem_putw, ddi_mem_putl, ddi_mem_putll – write data to mapped device in the
 memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr,
 uint8_t value);

void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr,
 uint16_t value);

void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr,
 uint32_t value);

void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr,
 uint64_t value);

HANDLE The data access handle returned from setup calls, such as
 ddi_regs_map_setup(9F).

dev_addr Base device address.

value The data to be written to the device.

INTERFACE Solaris DDI specific (Solaris DDI).

DESCRIPTION These routines generate a write of various sizes to memory space or allocated DMA
 memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and
 ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data,
 respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
 between the host and the device based on the encoded information in the data access
 handle. The translation may involve byte-swapping if the host and the device have
 incompatible endian characteristics.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F),
 ddi_device_acc_attr(9S)

NOTES The functions described in this manual page previously used symbolic names which
 specified their data access size; the function names have been changed so they now
 specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
**ddi_mem_putl(9F)**

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putl</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb, ddi_mem_putw, ddi_mem_putl, ddi_mem_putll - write data to mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Base device address.

cvalue The data to be written to the device.

INTERFACE Solaris DDI specific (Solaris DDI).

LEVEL DESCRIPTION These routines generate a write of various sizes to memory space or allocated DMA memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
ddi_mem_putll(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putl</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_pub,
/ddi_mem_putw, ddi_mem_putl, ddi_mem_putll - write data to mapped device in the
memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_put8(ddi_acc_handle_t
handle, uint8_t *dev_addr,
uint8_t value);

void ddi_mem_put16(ddi_acc_handle_t
handle, uint16_t *dev_addr,
uint16_t value);

void ddi_mem_put32(ddi_acc_handle_t
handle, uint32_t *dev_addr,
uint32_t value);

void ddi_mem_put64(ddi_acc_handle_t
handle, uint64_t *dev_addr,
uint64_t value);

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

dev_addr Base device address.

value The data to be written to the device.

INTERFACE Solaris DDI specific (Solaris DDI).

DESCRIPTION These routines generate a write of various sizes to memory space or allocated DMA
memory. The ddi_mem_put8(), ddi_mem_put16(), ddi_mem_put32(), and
/ddi_mem_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data,
respectively, to the device address, dev_addr, in memory space.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F),
/ddi_device_acc_attr(9S)

NOTES The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
</tbody>
</table>
### ddi_mem_putw(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_put1</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_get16(9F)

NAME
ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64,
ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb –
read multiple data from mapped device in the memory space or allocated DMA
memory

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void
ddi_mem_rep_get8
(ddi_acc_handle_t
handle,
uint8_t *
host_addr,
uint8_t *
dev_addr,
size_t
repcount,
uint_t
flags);

void
ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *
host_addr,
uint16_t *
dev_addr,
size_t
repcount,
uint_t
flags);

void
ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *
host_addr,
uint32_t *
dev_addr,
size_t
repcount,
uint_t
flags);

void
ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *
host_addr,
uint64_t *
dev_addr,
size_t
repcount,
uint_t
flags);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

DESCRIPTION
These routines generate multiple reads from memory space or allocated DMA
memory. repcount data is copied from the device address, dev_addr, in memory space to
the host address, host_addr. For each input datum, the ddi_mem_rep_get8(),
ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64()
functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device
address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_getb</td>
<td>ddi_mem_rep_get8</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>ddi_mem_rep_get16</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>ddi_mem_rep_get32</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>ddi_mem_rep_get64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64, ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getl – read multiple data from mapped device in the memory space or allocated DMA memory

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);
```

Solaris DDI specific (Solaris DDI).

**INTERFACE LEVEL PARAMETERS**

- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags:
  - `DDI_DEV_AUTOINCR`: Automatically increment the device address, `dev_addr`, during data accesses.
  - `DDI_DEV_NO_AUTOINCR`: Do not advance the device address, `dev_addr`, during data accesses.

**DESCRIPTION**

These routines generate multiple reads from memory space or allocated DMA memory. `repcount` data is copied from the device address, `dev_addr`, in memory space to the host address, `host_addr`. For each input datum, the `ddi_mem_rep_get8()`, `ddi_mem_rep_get16()`, `ddi_mem_rep_get32()`, and `ddi_mem_rep_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`, `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_getb</code></td>
<td><code>ddi_mem_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getw</code></td>
<td><code>ddi_mem_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getl</code></td>
<td><code>ddi_mem_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getll</code></td>
<td><code>ddi_mem_rep_get64</code></td>
</tr>
</tbody>
</table>
NAME  

ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64,  
ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb –  
read multiple data from mapped device in the memory space or allocated DMA  
memory

SYNOPSIS  

#include <sys/ddi.h>  
#include <sys/sunddi.h>  

void  

ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,  
                 uint8_t *dev_addr, size_t repcount, uint_t flags);  

void  

ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,  
                  uint16_t *dev_addr, size_t repcount, uint_t flags);  

void  

ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,  
                  uint32_t *dev_addr, size_t repcount, uint_t flags);  

void  

ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,  
                  uint64_t *dev_addr, size_t repcount, uint_t flags);  

INTERFACE  

LEVEL  

PARAMETERS  

Solaris DDI specific (Solaris DDI).

handle   The data access handle returned from setup calls, such as  
         ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr  Base device address.

repcount  Number of data accesses to perform.

flags     Device address flags:
            DDI_DEV_AUTOINCR Automatically increment the device  
                     address, dev_addr, during data  
                     accesses.
            DDI_DEV_NO_AUTOINCR Do not advance the device address,  
                                   dev_addr, during data accesses.

DESCRIPTION  

These routines generate multiple reads from memory space or allocated DMA  
memory. repcount data is copied from the device address, dev_addr, in memory space to  
the host address, host_addr. For each input datum, the ddi_mem_rep_get8(),  
ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64()  
functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device  
address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary  
described by the function.

Each individual datum will automatically be translated to maintain a consistent view  
between the host and the device based on the encoded information in the data access  
handle. The translation may involve byte-swapping if the host and the device have  
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_getb</td>
<td>ddi_mem_rep_get8</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>ddi_mem_rep_get16</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>ddi_mem_rep_get32</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>ddi_mem_rep_get64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_get8(9F)

NAME

ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64, ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb — read multiple data from mapped device in the memory space or allocated DMA memory

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION

These routines generate multiple reads from memory space or allocated DMA memory. repcount data is copied from the device address, dev_addr, in memory space to the host address, host_addr. For each input datum, the ddi_mem_rep_get8(), ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

These functions can be called from user, kernel, or interrupt context.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_getb</td>
<td>ddi_mem_rep_get8</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>ddi_mem_rep_get16</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>ddi_mem_rep_get32</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>ddi_mem_rep_get64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_getb(9F)

NAME
ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64,
ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb –
read multiple data from mapped device in the memory space or allocated DMA
memory

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_get8 (ddi_acc_handle_t handle, uint8_t *host_addr,
                       uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get16 (ddi_acc_handle_t handle, uint16_t *host_addr,
                        uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get32 (ddi_acc_handle_t handle, uint32_t *host_addr,
                        uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get64 (ddi_acc_handle_t handle, uint64_t *host_addr,
                        uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle The data access handle returned from setup calls, such as
       ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.
flags Device address flags:

        DDI_DEV_AUTOINCR Automatically increment the device
                         address, dev_addr, during data
                         accesses.

        DDI_DEV_NO_AUTOINCR Do not advance the device address,
                             dev_addr, during data accesses.

DESCRIPTION
These routines generate multiple reads from memory space or allocated DMA
memory. repcount data is copied from the device address, dev_addr, in memory space to
the host address, host_addr. For each input datum, the ddi_mem_rep_get8(),
ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device
address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary
described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_getb</code></td>
<td><code>ddi_mem_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getw</code></td>
<td><code>ddi_mem_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getl</code></td>
<td><code>ddi_mem_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getll</code></td>
<td><code>ddi_mem_rep_get64</code></td>
</tr>
</tbody>
</table>
NAME  ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64, ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb – read multiple data from mapped device in the memory space or allocated DMA memory

SYNOPSIS  
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);
```

INTERFACE LEVEL

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</td>
</tr>
<tr>
<td>host_addr</td>
<td>Base host address.</td>
</tr>
<tr>
<td>dev_addr</td>
<td>Base device address.</td>
</tr>
<tr>
<td>repcount</td>
<td>Number of data accesses to perform.</td>
</tr>
<tr>
<td>flags</td>
<td>Device address flags:</td>
</tr>
<tr>
<td></td>
<td>DDI_DEV_AUTOINCR</td>
</tr>
<tr>
<td></td>
<td>DDI_DEV_NO_AUTOINCR</td>
</tr>
</tbody>
</table>

DESCRIPTION

These routines generate multiple reads from memory space or allocated DMA memory. `repcount` data is copied from the device address, `dev_addr`, in memory space to the host address, `host_addr`. For each input datum, the `ddi_mem_rep_get8()`, `ddi_mem_rep_get16()`, `ddi_mem_rep_get32()`, and `ddi_mem_rep_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`. `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
- `ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_getb</code></td>
<td><code>ddi_mem_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getw</code></td>
<td><code>ddi_mem_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getl</code></td>
<td><code>ddi_mem_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getll</code></td>
<td><code>ddi_mem_rep_get64</code></td>
</tr>
</tbody>
</table>
ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64, ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb – read multiple data from mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>  
#include <sys/sunddi.h>

void ddi_mem_rep_get8 (ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get16 (ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get32 (ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get64 (ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

These routines generate multiple reads from memory space or allocated DMA memory. repcount data is copied from the device address, dev_addr, in memory space to the host address, host_addr. For each input datum, the ddi_mem_rep_get8(), ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_getb</code></td>
<td><code>ddi_mem_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getw</code></td>
<td><code>ddi_mem_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getl</code></td>
<td><code>ddi_mem_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getll</code></td>
<td><code>ddi_mem_rep_get64</code></td>
</tr>
</tbody>
</table>
NAME

ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64,
  ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb

- read multiple data from mapped device in the memory space or allocated DMA
  memory

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
                       uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
                        uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
                        uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,
                        uint64_t *dev_addr, size_t repcount, uint_t flags);
```

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

- `handle` The data access handle returned from setup calls, such as
  ddi_regs_map_setup(9F).
- `host_addr` Base host address.
- `dev_addr` Base device address.
- `repcount` Number of data accesses to perform.
- `flags` Device address flags:
  - `DDI_DEV_AUTOINCR` Automatically increment the device address, `dev_addr`, during data accesses.
  - `DDI_DEV_NO_AUTOINCR` Do not advance the device address, `dev_addr`, during data accesses.

DESCRIPTION

These routines generate multiple reads from memory space or allocated DMA memory. `repcount` data is copied from the device address, `dev_addr`, in memory space to the host address, `host_addr`. For each input datum, the ddi_mem_rep_get8(), ddi_mem_rep_get16(), ddi_mem_rep_get32(), and ddi_mem_rep_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, `dev_addr`. `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_put8(9F),
  ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_getb</code></td>
<td><code>ddi_mem_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getw</code></td>
<td><code>ddi_mem_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getl</code></td>
<td><code>ddi_mem_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_getll</code></td>
<td><code>ddi_mem_rep_get64</code></td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb – write multiple data to mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8 (ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put16 (ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put32 (ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put64 (ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
   Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
   Do not advance the device address, dev_addr, during data accesses.

These routines generate multiple writes to memory space or allocated DMA memory. repcount data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_putb</td>
<td>ddi_mem_rep_put8</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>ddi_mem_rep_put16</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>ddi_mem_rep_put32</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>ddi_mem_rep_put64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb – write multiple data to mapped device in the memory space or allocated DMA memory.

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);
```

**Solaris DDI specific (Solaris DDI).**

**NAME**

- ddi_mem_rep_put8
- ddi_mem_rep_put16
- ddi_mem_rep_put32
- ddi_mem_rep_put64
- ddi_mem_rep_putw
- ddi_mem_rep_putl
- ddi_mem_rep_putll
- ddi_mem_rep_putb

**SYNOPSIS**

- The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- Base host address.
- Base device address.
- Number of data accesses to perform.
- Device address flags:
  - `DDI_DEV_AUTOINCR`: Automatically increment the device address, `dev_addr`, during data accesses.
  - `DDI_DEV_NO_AUTOINCR`: Do not advance the device address, `dev_addr`, during data accesses.

**INTERFACE LEVEL PARAMETERS**

- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags:
  - `DDI_DEV_AUTOINCR`: Automatically increment the device address, `dev_addr`, during data accesses.
  - `DDI_DEV_NO_AUTOINCR`: Do not advance the device address, `dev_addr`, during data accesses.

**DESCRIPTION**

These routines generate multiple writes to memory space or allocated DMA memory. `repcount` data is copied from the host address, `host_addr`, to the device address, `dev_addr`, in memory space. For each input datum, the `ddi_mem_rep_put8()`, `ddi_mem_rep_put16()`, `ddi_mem_rep_put32()`, and `ddi_mem_rep_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_putb</td>
<td>ddi_mem_rep_put8</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>ddi_mem_rep_put16</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>ddi_mem_rep_put32</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>ddi_mem_rep_put64</td>
</tr>
</tbody>
</table>
The Solaris DDI specific (Solaris DDI).

**Interface Level Parameters**

- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags:
  - `DDI_DEV_AUTOINCR`: Automatically increment the device address, `dev_addr`, during data accesses.
  - `DDI_DEV_NO_AUTOINCR`: Do not advance the device address, `dev_addr`, during data accesses.

**Description**

These routines generate multiple writes to memory space or allocated DMA memory. `repcount` data is copied from the host address, `host_addr`, to the device address, `dev_addr`, in memory space. For each input datum, the `ddi_mem_rep_put8()`, `ddi_mem_rep_put16()`, `ddi_mem_rep_put32()`, and `ddi_mem_rep_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_putb</td>
<td>ddi_mem_rep_put8</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>ddi_mem_rep_put16</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>ddi_mem_rep_put32</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>ddi_mem_rep_put64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb – write multiple data to mapped device in the memory space or allocated DMA memory

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.
flags Device address flags:

DDI_DEV_AUTOINCR
  Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
  Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION These routines generate multiple writes to memory space or allocated DMA memory. repcount data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

These functions can be called from user, kernel, or interrupt context.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_putb</td>
<td>ddi_mem_rep_put8</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>ddi_mem_rep_put16</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>ddi_mem_rep_put32</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>ddi_mem_rep_put64</td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb – write multiple data to mapped device in the memory space or allocated DMA memory

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);
```

Solaris DDI specific (Solaris DDI).

**NAME**

- ddi_mem_rep_put8
- ddi_mem_rep_put16
- ddi_mem_rep_put32
- ddi_mem_rep_put64
- ddi_mem_rep_putw
- ddi_mem_rep_putl
- ddi_mem_rep_putll
- ddi_mem_rep_putb

**SYNOPSIS**

- handle: The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
- host_addr: Base host address.
- dev_addr: Base device address.
- repcount: Number of data accesses to perform.
- flags: Device address flags:
  - DDI_DEV_AUTOINCR
    - Automatically increment the device address, dev_addr, during data accesses.
  - DDI_DEV_NO_AUTOINCR
    - Do not advance the device address, dev_addr, during data accesses.

**INTERFACE LEVEL PARAMETERS**

These routines generate multiple writes to memory space or allocated DMA memory. repcount data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_get8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_putb</code></td>
<td><code>ddi_mem_rep_put8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putw</code></td>
<td><code>ddi_mem_rep_put16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putl</code></td>
<td><code>ddi_mem_rep_put32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putll</code></td>
<td><code>ddi_mem_rep_put64</code></td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb - write multiple data to mapped device in the memory space or allocated DMA memory.

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.
flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION These routines generate multiple writes to memory space or allocated DMA memory. repcount data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_putb</code></td>
<td><code>ddi_mem_rep_put8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putw</code></td>
<td><code>ddi_mem_rep_put16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putl</code></td>
<td><code>ddi_mem_rep_put32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putll</code></td>
<td><code>ddi_mem_rep_put64</code></td>
</tr>
</tbody>
</table>
ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), ddi_mem_rep_put64() -
write multiple data to mapped device in the memory space or allocated DMA memory.

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
                      uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
                      uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
                      uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
                      uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

- **handle**: The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags:
  - **DDI_DEV_AUTOINCR**: Automatically increment the device address, dev_addr, during
data accesses.
  - **DDI_DEV_NO_AUTOINCR**: Do not advance the device address, dev_addr, during data
    accesses.

These routines generate multiple writes to memory space or allocated DMA memory. repcount
data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(),
ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device
address. dev_addr and host_addr must be aligned to the datum boundary described by
the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F),
 ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_putb</code></td>
<td><code>ddi_mem_rep_put8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putw</code></td>
<td><code>ddi_mem_rep_put16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putl</code></td>
<td><code>ddi_mem_rep_put32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putll</code></td>
<td><code>ddi_mem_rep_put64</code></td>
</tr>
</tbody>
</table>
NAME

ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64, ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb – write multiple data to mapped device in the memory space or allocated DMA memory.

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

INTERFACE LEVEL PARAMETERS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
  Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
  Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION

These routines generate multiple writes to memory space or allocated DMA memory. repcount data is copied from the host address, host_addr, to the device address, dev_addr, in memory space. For each input datum, the ddi_mem_rep_put8(), ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
**ddi_mem_rep_putw(9F)**

When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_mem_rep_putb</code></td>
<td><code>ddi_mem_rep_put8</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putw</code></td>
<td><code>ddi_mem_rep_put16</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putl</code></td>
<td><code>ddi_mem_rep_put32</code></td>
</tr>
<tr>
<td><code>ddi_mem_rep_putll</code></td>
<td><code>ddi_mem_rep_put64</code></td>
</tr>
</tbody>
</table>
ddi_mmap_get_model(9F)

NAME  
ddi_mmap_get_model – return data model type of current thread

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint_t ddi_mmap_get_model(void);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

DESCRIPTION  
ddi_mmap_get_model() returns the C Language Type Model which the current
thread expects. ddi_mmap_get_model() is used in combination with
ddi_model_convert_from(9F) in the mmap(9E) driver entry point to determine
whether there is a data model mismatch between the current thread and the device
driver. The device driver might have to adjust the shape of data structures before
exporting them to a user thread which supports a different data model.

RETURN VALUES  
DDI_MODEL_ILP32  
Current thread expects 32-bit (ILP32) semantics.

DDI_MODEL_LP64  
Current thread expects 64-bit (LP64) semantics.

DDI_FAILURE  
The ddi_mmap_get_model() function was not called
from the mmap(9E) entry point.

CONTEXT  
The ddi_mmap_get_model() function can only be called from the mmap(9E) driver
entry point.

EXAMPLES  
EXAMPLE 1: Using ddi_mmap_get_model()

The following is an example of the mmap(9E) entry point and how to support 32-bit
and 64-bit applications with the same device driver.

struct data32 {
    int len;
    caddr32_t addr;
};

struct data {
    int len;
    caddr_t addr;
};

xxmmap(dev_t dev, off_t off, int prot) {
    struct data dtc; /* a local copy for clash resolution */
    struct data *dp = (struct data *)shared_area;

    switch (ddi_model_convert_from(ddi_mmap_get_model())) {
    case DDI_MODEL_ILP32:
    {
        struct data32 *da32p;
        da32p = (struct data32 *)shared_area;
        dp = &dtc;
        dp->len = da32p->len;
        dp->address = da32->address;
        break;
    }
ddi_mmap_get_model(9F)

EXAMPLE 1: Using ddi_mmap_get_model() (Continued)

    case DDI_MODEL_NONE:
        break;
    }
    /* continues along using dp */
    ...
ddi_model_convert_from() is used to determine if the current thread uses a different C Language Type Model than the device driver. The 64-bit version of Solaris will require a 64-bit kernel to support both 64-bit and 32-bit user mode programs. The difference between a 32-bit program and a 64-bit program is in its C Language Type Model: a 32-bit program is ILP32 (integer, longs, and pointers are 32-bit) and a 64-bit program is LP64 (longs and pointers are 64-bit). There are a number of driver entry points such as ioctl(9E) and mmap(9E) where it is necessary to identify the C Language Type Model of the user-mode originator of a kernel event. For example, any data which flows between programs and the device driver or vice versa need to be identical in format. A 64-bit device driver may need to modify the format of the data before sending it to a 32-bit application. ddi_model_convert_from() is used to determine if data that is passed between the device driver and the application requires reformatting to any non-native data model.

### EXAMPLE 1: Using ddi_model_convert_from() in the ioctl() entry point to support both 32-bit and 64-bit applications.

The following is an example how to use ddi_model_convert_from() in the ioctl() entry point to support both 32-bit and 64-bit applications.

```c
struct passargs32 {
    int len;
    caddr32_t addr;
};

struct passargs {
    int len;
    caddr_t addr;
};
xioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp, int *rvalp) {
    struct passargs pa;

    switch (ddi_model_convert_from(mode & FMODELS)) { 
    case DDI_MODEL_ILP32: 
```

### Kernel Functions for Drivers 663
EXAMPLE 1: Using `ddi_model_convert_from()` in the `ioctl()` entry point to support both 32-bit and 64-bit applications.  

(Continued)

```c
struct passargs32 pa32;

ddi_copyin(arg, &pa32, sizeof (struct passargs32), mode);
pa.len = pa32.len;
pa.address = pa32.address;
break;
}
```

```c
case DDI_MODEL_NONE:
    ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
    break;
```

```c
do_ioctl(&pa);
```

SEE ALSO

`ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)`

*Writing Device Drivers*
ddi_node_name returns the devinfo node name contained in the dev_info node pointed to by dip.

ddi_node_name can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers
### NAME
ddi_no_info(9F)

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_no_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
                 void **result);
```

### INTERFACE LEVEL PARAMETERS
- **Solaris DDI specific (Solaris DDI)**
- **dev_info_t *dip**
  - Pointer to dev_info structure.
- **ddi_info_cmd_t infocmd**
  - Command argument. Valid command values are: DDI_INFO_DEVT2DEVINFO and DDI_INFO_DEVT2INSTANCE.
- **void *arg**
  - Command-specific argument.
- **void **result**
  - Pointer to where the requested information is stored.

### DESCRIPTION
The `ddi_no_info()` function always returns DDI_FAILURE. It is provided as a convenience routine for drivers not providing a `cb_ops(9S)` or for network drivers only providing DLPI-2 services. Such drivers can use `ddi_no_info()` in the `devo_getinfo` entry point (see `getinfo(9E)`) of the `dev_ops(9S)` structure.

### RETURN VALUES
The `ddi_no_info()` function always returns DDI_FAILURE.

### SEE ALSO
getinfo(9E), qassociate(9F), cb_ops(9S), dev_ops(9S)
**ddi_peek16(9F)**

**NAME**
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd – read a value from a location

**SYNOPSIS**
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);

int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);

int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);

int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

**PARAMETERS**

- **dip**  
  A pointer to the device’s dev_info structure.

- **addr**  
  Virtual address of the location to be examined.

- **valuep**  
  Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

**DESCRIPTION**
These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

**RETURN VALUES**

- **DDI_SUCCESS**  
  The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.

- **DDI_FAILURE**  
  An error occurred while trying to read the location. *valuep is unchanged.

**CONTEXT**
These functions can be called from user or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:

```c
if (ddi_peek8(dip, csr, (int8_t *)&s)) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...
    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "!%s%d: device type 0x%x\n",
                ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddi_peekl</td>
<td>ddi_peek32</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>ddi_peek64</td>
</tr>
</tbody>
</table>
### NAME
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd – read a value from a location

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

### PARAMETERS
- **dip** A pointer to the device’s dev_info structure.
- **addr** Virtual address of the location to be examined.
- **valuep** Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

### DESCRIPTION
These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

### RETURN VALUES
- **DDI_SUCCESS** The value at the given virtual address was successfully read, and if `valuep` is non-null, `*valuep` will have been updated.
- **DDI_FAILURE** An error occurred while trying to read the location. `*valuep` is unchanged.

### CONTEXT
These functions can be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:

```c
if (ddi_peek8(dip, csr, (int8_t *)&0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ... /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "%s%d: device type 0x%x
                ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ... ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>ddi.peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>

670 man pages section 9: DDI and DKI Kernel Functions • Last Revised 27 Sep 2002
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd
– read a value from a location

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

PARAMETERS

*dip* A pointer to the device’s dev_info structure.
*addr* Virtual address of the location to be examined.
*valuep* Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

DESCRIPTION

These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

RETURN VALUES

DDI_SUCCESS The value at the given virtual address was successfully read, and if *valuep* is non-null, *valuep* will have been updated.

DDI_FAILURE An error occurred while trying to read the location. *valuep* is unchanged.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLES

**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:

```c
if (ddi_peek8(dip, csr, (int8_t *)0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:  (Continued)

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
...
/* map device registers */
...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
            ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "!%s%d: device type 0x%x",
            ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ... 
    ...

    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO  attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES  The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddipeekl</td>
<td>ddi_peek32</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>ddi_peek64</td>
</tr>
</tbody>
</table>
ddi_peek8, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd – read a value from a location

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

RETURN VALUES

DDI_SUCCESS The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.

DDI_FAILURE An error occurred while trying to read the location. *valuep is unchanged.

These functions can be called from user or interrupt context.

EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

```c
if (ddi_peek8(dip, csr, (int8_t *)&0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
ddi.peek8(9F)

**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:  
(Continued)

**EXAMPLE 2** Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddiAttach_cmd_t cmd)
{
    /* map device registers */
    ...
    if (ddi.peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn.err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn.err(CE_CONT, "%s%d: device type 0x%x
                ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

**SEE ALSO** attach(9E), probe(9E), ddi_poke(9F)

**Writing Device Drivers**

**NOTES** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi.peekc</td>
<td>ddi.peek8</td>
</tr>
<tr>
<td>ddi.peeks</td>
<td>ddi.peek16</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>ddi.peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>

674  man pages section 9: DDI and DKI Kernel Functions • Last Revised 27 Sep 2002
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd - read a value from a location

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peek1(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

PARAMETERS
dip A pointer to the device's dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

DESCRIPTION
These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver's probe(9E) or attach(9E) routines.

RETURN VALUES
DDI_SUCCESS The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.
DDI_FAILURE An error occurred while trying to read the location. *valuep is unchanged.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

if (ddi_peek8(dip, csr, (int8_t *)0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
ddi_peek(9F)

**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:  
(Continued)

**EXAMPLE 2** Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
        ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "%s%d: device type 0x%x
        ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

**SEE ALSO**  
attach(9E), probe(9E), ddi_poke(9F)

**Writing Device Drivers**

**NOTES**  
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi.peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi.peek16</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>ddi.peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd – read a value from a location

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(), and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(), ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

DDI_SUCCESS The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.
DDI_FAILURE An error occurred while trying to read the location. *valuep is unchanged.

These functions can be called from user or interrupt context.

EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

if (ddi_peek8(dip, csr, (int8_t *)&0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:  (Continued)

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "!%s%d: device type 0x%x!
                , ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...

    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO  attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES  The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi.peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi.peek16</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>ddi.peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>
### NAME
- ddi.peek, ddi.peek8, ddi.peek16, ddi.peek32, ddi.peek64, ddi.peekc, ddi.peeks, ddi.peekl, ddi.peekd

**ddi.peekd** – read a value from a location

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi.peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi.peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi.peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi.peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). The ddi.peekc(), ddi.peeks(), ddi.peekl(), and ddi.peekd() functions are obsolete. Use, respectively, ddi.peek8(), ddi.peek16(), ddi.peek32(), and ddi.peek64(), instead.

### PARAMETERS
- **dip** A pointer to the device’s dev_info structure.
- **addr** Virtual address of the location to be examined.
- **valuep** Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

### DESCRIPTION
These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

### RETURN VALUES
- **DDI_SUCCESS** The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.
- **DDI_FAILURE** An error occurred while trying to read the location. *valuep is unchanged.

### CONTEXT
These functions can be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space:

```c
if (ddi.peek8(dip, csr, (int8_t *)&amp;0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:  (Continued)

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
  ...
  /* map device registers */
  ...

  if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "%s%d: cannot read device identifier",
    ddi_get_name(dip), ddi_get_instance(dip));
    goto failure;
  } else
    cmn_err(CE_CONT, "%s%d: device type 0x%x
    ddi_get_name(dip), ddi_get_instance(dip), id_value);
  ...
  ...
  ddi_report_dev(dip);
  return (DDI_SUCCESS);

failure:
  /* free any resources allocated */
  ...
  return (DDI_FAILURE);
}
```

SEE ALSO  attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddi_peekl</td>
<td>ddi_peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>
ddi_peekl(9F)

NAME
ddi.peek, ddi.peek8, ddi.peek16, ddi.peek32, ddi.peek64, ddi.peekc, ddi.peeks,
ddi.peekl, ddi.peekd – read a value from a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi.peekc(), ddi.peeks(), ddi.peekl(), and ddi.peekd() functions are obsolete. Use, respectively, ddi.peek8(), ddi.peek16(), ddi.peek32(), and ddi.peek64(), instead.

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

DESCRIPTION
These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver’s probe(9E) or attach(9E) routines.

RETURN VALUES
DDI_SUCCESS The value at the given virtual address was successfully read, and if valuep is non-null, *valuep will have been updated.
DDI_FAILURE An error occurred while trying to read the location. *valuep is unchanged.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

if (ddi.peek8(dip, csr, (int8_t *)&valuep) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "%s%d: device type 0x%x\n",
                ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...

    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi.peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi.peek16</td>
</tr>
<tr>
<td>ddi.peekl</td>
<td>ddi.peek32</td>
</tr>
<tr>
<td>ddi.peekd</td>
<td>ddi.peek64</td>
</tr>
</tbody>
</table>
ddi_peeks(9F)

NAME
ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks,
ddi_peekl, ddi_peekd – read a value from a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_peekc(), ddi_peeks(), ddi_peekl(),
and ddi_peekd() functions are obsolete. Use, respectively, ddi_peek8(),
ddi_peek16(), ddi_peek32(), and ddi_peek64(), instead.

LEVEL

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the
value read from the location will simply be discarded.

DESCRIPTION
These routines cautiously attempt to read a value from a specified virtual address, and
return the value to the caller, using the parent nexus driver to assist in the process
where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an
error code is returned.

The routines are most useful when first trying to establish the presence of a device on
the system in a driver’s probe(9E) or attach(9E) routines.

RETURN VALUES
DDI_SUCCESS The value at the given virtual address was successfully read, and if
valuep is non-null, *valuep will have been updated.

DDI_FAILURE An error occurred while trying to read the location. *valuep is
unchanged.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel
address space:

```
if (ddi_peek8(dip, csr, (int8_t *)0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```
EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

(Continued)

EXAMPLE 2 Reading and logging the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "%s%d: device type 0x%x
                
        
    

    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO attach(9E), probe(9E), ddipoke(9F)

Writing Device Drivers

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peeckc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddi_peekl</td>
<td>ddi_peek32</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>ddi_peek64</td>
</tr>
</tbody>
</table>
**NAME**

ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

**SYNOPSIS**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

**PARAMETERS**

- **dip** A pointer to the device's dev_info structure.
- **addr** Virtual address of the location to be written to.
- **value** Value to be written to the location.

**DESCRIPTION**

These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

**RETURN VALUES**

- **DDI_SUCCESS** The value was successfully written to the given virtual address.
- **DDI_FAILURE** An error occurred while trying to write to the location.

**CONTEXT**

These functions can be called from user or interrupt context.

**SEE ALSO**

attach(9E), probe(9E), ddi_peek(9F)

**Writing Device Drivers**

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers  685
**ddi_poke16(9F)**

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke32</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
ddi_poke32(9F)

**NAME**
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

**SYNOPSIS**
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

**PARAMETERS**
dip A pointer to the device's dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

**DESCRIPTION**
These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

**RETURN VALUES**
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

**CONTEXT**
These functions can be called from user or interrupt context.

**SEE ALSO**
attach(9E), probe(9E), ddi_peek(9F)

_Writing Device Drivers_

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers 687
ddi_poke32(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
NAME:  ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS:  
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);
```

INTERFACE LEVEL:  Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

PARAMETERS:
- dip: A pointer to the device's dev_info structure.
- addr: Virtual address of the location to be written to.
- value: Value to be written to the location.

DESCRIPTION:  These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

RETURN VALUES:
- DDI_SUCCESS: The value was successfully written to the given virtual address.
- DDI_FAILURE: An error occurred while trying to write to the location.

CONTEXT:  These functions can be called from user or interrupt context.

SEE ALSO:  attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES:  The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
ddi_poke64(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
ddi_poke8(9F)

NAME
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

PARAMETERS
dip A pointer to the device's dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
### ddi_poke8(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
NAME ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

INTERFACE Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver’s probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
### ddi_poke(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
ddi_pokec(9F)

NAME
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes,
ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(),
and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(),
iddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using
the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring,
an error code is returned.

These routines are most useful when first trying to establish the presence of a given
device on the system in a driver’s probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use
the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
ddi_pokec(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
**NAME**
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes,
ddi_pokel, ddi_poked – write a value to a location

**SYNOPSIS**
```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(),
and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(),
ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

**PARAMETERS**
- *dip*  A pointer to the device’s dev_info structure.
- *addr*  Virtual address of the location to be written to.
- *value*  Value to be written to the location.

**DESCRIPTION**
These routines cautiously attempt to write a value to a specified virtual address, using
the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring,
an error code is returned.

These routines are most useful when first trying to establish the presence of a given
device on the system in a driver’s probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use
the ddi_peek(9F) routines instead if possible.

**RETURN VALUES**
- **DDI_SUCCESS**  The value was successfully written to the given virtual address.
- **DDI_FAILURE**  An error occurred while trying to write to the location.

**CONTEXT**
These functions can be called from user or interrupt context.

**SEE ALSO**
attach(9E), probe(9E), ddi_peek(9F)

*Writing Device Drivers*

**NOTES**
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
ddi_poked(9F)

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
ddi_pokec(9F)

NAME
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, 
ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), 
and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), 
ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

LEVEL

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using
the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring,
a error code is returned.

These routines are most useful when first trying to establish the presence of a given
device on the system in a driver’s probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use
the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>
**ddi_pokel(9F)**

<table>
<thead>
<tr>
<th>ddi_pokes</th>
<th>ddi_poke16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

Solaris DDI specific (Solaris DDI). The ddi_pokec(), ddi_pokes(), ddi_pokel(), and ddi_poked() functions are obsolete. Use, respectively, ddi_poke8(), ddi_poke16(), ddi_poke32(), and ddi_poke64(), instead.

PARAMETERS
dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver’s probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
</tbody>
</table>

Kernel Functions for Drivers  701
### ddi_pokes(9F)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokes</td>
<td>ddi_poke16</td>
</tr>
<tr>
<td>ddi_poke1</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
NAME

ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all,
ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS

#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int length);

int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *
name);

int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int length);

int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

void ddi_prop_remove_all(dev_info_t *dip);

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI). The ddi_prop_create() and
ddi_prop_modify() functions are obsolete. Use ddi_prop_update(9F) instead of
these functions.

PARAMETERS

ddi_prop_create()

devo dev_t of the device.
dip dev_info_t pointer of the device.
flags flag modifiers. The only possible flag value is
DDI_PROP_CANSLEEP: Memory allocation may sleep.
name name of property.
valuep pointer to property value.
length property length.

ddi_prop_undefine()

devo dev_t of the device.
dip dev_info_t pointer of the device.
flags flag modifiers. The only possible flag value is
DDI_PROP_CANSLEEP: Memory allocation may sleep.
name name of property.

ddi_prop_modify()

devo dev_t of the device.
dip dev_info_t pointer of the device.
Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses `ddi_getproplen(9F)` to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver's property list associated with `dip`.

Property definitions are stacked; they are added to the beginning of the driver's property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be returned to the caller.

The individual functions are described as follows:

`ddi_prop_create()`
`ddi_prop_create()` adds a property to the device's property list. If the property is not associated with any particular `dev` but is associated with the physical device itself, then the argument `dev` should be the special device `DDI_DEV_T_NONE`. If you do not have a `dev` for your device (for example during `attach(9E)` time), you can create one using `makedevice(9F)` with a major number of `DDI_MAJOR_T_UNKNOWN`. `ddi_prop_create()` will then make the correct `dev` for your device.

For boolean properties, you must set `length` to 0. For all other properties, the `length` argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If `flags` does not contain `DDI_PROP_CANSLEEP`, `ddi_prop_create()` returns `DDI_PROP_NO_MEMORY` on memory allocation failure or `DDI_PROP_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.
ddi_prop_undefine()

ddi_prop_undefine() is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. However, ddi_prop_undefine() will not terminate a search when the ddi_prop_get_int64(9F) or ddi_prop_lookup_int64_array(9F) routines are used for lookup of 64-bit property value. See ddi_prop_op(9F).

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi_prop_create().

ddi_prop_modify()

ddi_prop_modify() modifies the length and the value of a property. If ddi_prop_modify() finds the property in the driver’s property list, allocates memory for the property value and returns DDI_PROP_SUCCESS. If the property was not found, the function returns DDI_PROP_NOT_FOUND.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi_prop_create().

ddi_prop_remove()

ddi_prop_remove() unlinks a property from the device’s property list. If ddi_prop_remove() finds the property (an exact match of both name and dev), it unlinks the property, frees its memory, and returns DDI_PROP_SUCCESS, otherwise, it returns DDI_PROP_NOT_FOUND.

ddi_prop_remove_all()

ddi_prop_remove_all() removes the properties of all the dev_t’s associated with the dip. It is called before unloading a driver.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| ddi_prop_create() | DDI_PROP_SUCCESS | On success.
| | DDI_PROP_NO_MEMORY | On memory allocation failure.
| | DDI_PROP_INVAL_ARG | If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
| ddi_prop_undefine() | DDI_PROP_SUCCESS | On success.
| | DDI_PROP_NO_MEMORY | On memory allocation failure.
| | DDI_PROP_INVAL_ARG | If an attempt is made to create a property with dev DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
| ddi_prop_modify() | DDI_PROP_SUCCESS | On success.
| | DDI_PROP_NO_MEMORY | On memory allocation failure.
| | DDI_PROP_INVAL_ARG | If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
**ddi_prop_create(9F)**

- **DDI_PROP_NOT_FOUND**: On property search failure.
- **DDI_PROP_SUCCESS**: On success.
- **DDI_PROP_INVAL_ARG**: If an attempt is made to create a property with `dev` equal to `DDI_DEV_T_ANY` or if `name` is `NULL` or `name` is the `NULL` string.
- **DDI_PROP_NOT_FOUND**: On property search failure.

**CONTEXT**

If `DDI_PROP_CANSLEEP` is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

**EXAMPLES**

**EXAMPLE 1** Creating a Property

The following example creates a property called `nblocks` for each partition on a disk.

```c
int propval = 8192;

for (minor = 0; minor < 8; minor++) {  
    (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),  
    dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t) &propval,  
    sizeof (int));  
    ...
}
```

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_prop_create() and ddi_prop_modify() are Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

- `driver.conf(4)`, `attributes(5)`, `attach(9E)`, `ddi_getproplen(9F)`,
- `ddi_prop_op(9F)`, `ddi_prop_update(9F)`, `makedevice(9F)`

*Writing Device Drivers*
ddi_prop_exists() checks for the existence of a property regardless of the property value data type.

Properties are searched for based on the dip, name, and match_dev. The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return 0 if not found and 1 if found.

Usually, the match_dev argument should be set to the actual device number that this property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY, then ddi_prop_exists() will match the request regardless of the match_dev the property was created with. That is the first property whose name matches name will be returned. If a property was created with match_dev set to DDI_DEV_T_NONE then the only way to look up this property is with a match_dev set to DDI_DEV_T_ANY. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.
ddi_prop_exists(9F)

**RETURN VALUES**

- **ddi_prop_exists()** returns 1 if the property exists and 0 otherwise.

**CONTEXT**

- These functions can be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1: Using ddi_prop_exists()**

The following example demonstrates the use of ddi_prop_exists().

```c
/*
 * Enable "whizzy" mode if the "whizzy-mode" property exists
 */
if (ddi_prop_exists(xx_dev, xx_dip, DDI_PROP_NOTPRM, "whizzy-mode") == 1) {
    xx_enable_whizzy_mode(xx_dip);
} else {
    xx_disable_whizzy_mode(xx_dip);
}
```

**SEE ALSO**

- ddi_prop_get_int(9F), ddi_prop_lookup(9F), ddi_prop_remove(9F), ddi_prop_update(9F)

*Writing Device Drivers*
NAME

ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_int64_array,
ddi_prop_lookup_string_array, ddi_prop_lookup_string,
ddi_prop_lookup_byte_array, ddi_prop_free

- look up property information

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, char **datap);

void ddi_prop_free(void *data);

PARAMETERS

match_dev
Device number associated with property or DDI_DEV_T_ANY.

dip
Pointer to the device info node of device whose property list
should be searched.

flags
Possible flag values are some combination of:

DDI_PROP_DONTPASS
   Do not pass request to parent device information node if the
   property is not found.

DDI_PROP_NOTPROM
   Do not look at PROM properties (ignored on platforms that do
   not support PROM properties).

name
String containing the name of the property.

nelementsp
The address of an unsigned integer which, upon successful return,
will contain the number of elements accounted for in the memory
pointed at by datap. The elements are either integers, strings or
bytes depending on the interface used.

datap

   ddi_prop_lookup_int_array()
   The address of a pointer to an array of integers which, upon
   successful return, will point to memory containing the integer
   array property value.
ddi_prop_free(9F)

ddi_prop_lookup_int64_array()
The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

ddi_prop_lookup_string_array()
The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

ddi_prop_lookup_string()
The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

ddi_prop_lookup_byte_array()
The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

INTERFACE
LEVEL
DESCRIPTION
Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the dip, name, match_dev, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the match_dev argument should be set to the actual device number that this property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual match_dev the property was created with. If a property was created with match_dev set to DDI_DEV_T_NONE, then the only way to look up this property is with a match_dev set to DDI_DEV_T_ANY. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case `*datap` points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. `nelementsp` is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by `*datap`.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if one called `ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free(my_int_ptr)`.

`ddi_prop_lookup_int_array()`
This routine searches for and returns an array of integer property values. An array of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap` should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.

`ddi_prop_lookup_int64_array()`
This routine searches for and returns an array of 64-bit integer property values. The array is defined to be `*nelementsp` number of `int64_t` elements. `datap` should be set to the address of a pointer to an array of `int64_t`'s which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.

`ddi_prop_lookup_string_array()`
This routine searches for and returns a property that is an array of strings. `datap` should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the `argv` argument to `execve(2)`.

`ddi_prop_lookup_string()`
This routine searches for and returns a property that is a null-terminated string. `datap` should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.

`ddi_prop_lookup_byte_array()`
This routine searches for and returns a property that is an array of bytes. `datap` should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
ddi_prop_free(9F)

ddi_prop_free()
Frees the resources associated with a property previously allocated using
ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(),
ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or
ddi_prop_lookup_byte_array().

RETURN VALUES
The functions ddi_prop_lookup_int_array(),
ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),
ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the
following values:

DDI_PROP_SUCCESS          Upon success.
DDI_PROP_INVAL_ARG         If an attempt is made to look up a property
                          with match_dev equal to DDI_DEV_T_NONE,
                          name is NULL or name is the null string.
DDI_PROP_NOT_FOUND         Property not found.
DDI_PROP_UNDEFINED         Property explicitly not defined (see
                          ddi_prop_undefine(9F)).
DDI_PROP_CANNOT_Decode    The value of the property cannot be
d                          decoded.

CONTEXT
These functions can be called from user or kernel context.

EXAMPLES  

EXAMPLE 1 Using ddi_prop_lookup_int_array()

The following example demonstrates the use of ddi_prop_lookup_int_array().

int  *options;
int   noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
   "options", &options, &noptions) == DDI_PROP_SUCCESS) {

   /*
   * Do "our thing" with the options data from the property
   */
   xx_process_options(options, noptions);

   /*
   * Free the memory allocated for the property data
   */
   ddi_prop_free(options);
}

SEE ALSO execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),
ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)
Writing Device Drivers

/ddi_prop_free(9F)
ddi_prop_get_int64(9F)

NAME

ddi_prop_get_int, ddi_prop_get_int64 – lookup integer property

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_get_int(dev_t match_dev, dev_info_t *dip, uint_t flags,
char *name, int defvalue);

int64_t ddi_prop_get_int64(dev_t match_dev, dev_info_t *dip, uint_t flags,
char *name, int64_t defvalue);

PARAMETERS

match_dev  Device number associated with property or DDI_DEV_T_ANY.
dip  Pointer to the device info node of device whose property list
    should be searched.
flags  Possible flag values are some combination of:
    DDI_PROP_DONTPASS
           Do not pass request to parent device information node if
           property not found.
    DDI_PROP_NOTPROM
           Do not look at PROM properties (ignored on platforms that do
           not support PROM properties).
name  String containing the name of the property.
defvalue  An integer value that is returned if the property cannot be found.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

The ddi_prop_get_int() and ddi_prop_get_int64() functions search for an
integer property and, if found, returns the value of the property.

Properties are searched for based on the dip, name, match_dev, and the type of the data
(integer). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return defvalue.

Usually, the match_dev argument should be set to the actual device number that this
property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY,
then ddi_prop_get_int() and ddi_prop_get_int() will match the request.
regardless of the `match_dev` the property was created with. If a property was created with `match_dev` set to DDI_DEV_T_NONE, then the only way to look up this property is with a `match_dev` set to DDI_DEV_T_ANY. PROM properties are always created with `match_dev` set to DDI_DEV_T_NONE.

`name` must always be set to the name of the property being looked up.

The return value of the routine is the value of the property. If the property is not found, the argument `defvalue` is returned as the value of the property.

`ddi_prop_get_int64()` will not search the PROM for 64-bit property values.

**RETURN VALUES**

`ddi_prop_get_int()` and `ddi_prop_get_int64()` return the value of the property. If the property is not found, the argument `defvalue` is returned. If the property is found, but cannot be decoded into an `int` or an `int64`, then `DDI_PROP_NOT_FOUND` is returned.

**CONTEXT**

`ddi_prop_get_int()` and `ddi_prop_get_int64()` can be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1** Using `ddi_prop_get_int()`

The following example demonstrates the use of `ddi_prop_get_int()`.

```c
/*
 * Get the value of the integer "width" property, using
 * our own default if no such property exists
 */
width = ddi_prop_get_int(xx_dev, xx_dip, 0, "width",
                        XX_DEFAULT_WIDTH);
```

**SEE ALSO**

`ddi_prop_exists(9F), ddi_prop_lookup(9F), ddi_prop_remove(9F),
 ddi_prop_update(9F)`

*Writing Device Drivers*
ddi_prop_get_int(9F)

NAME  ddi_prop_get_int, ddi_prop_get_int64 – lookup integer property

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_get_int(dev_t match_dev, dev_info_t *dip, uint_t flags,
char *name, int defvalue);

int64_t ddi_prop_get_int64(dev_t match_dev, dev_info_t *dip, uint_t
flags, char *name, int64_t defvalue);

PARAMETERS

match_dev  Device number associated with property or DDI_DEV_T_ANY.
dip  Pointer to the device info node of device whose property list
should be searched.
flags  Possible flag values are some combination of:

DDI_PROP_DONTPASS
   Do not pass request to parent device information node if
   property not found.

DDI_PROP_NOTPROM
   Do not look at PROM properties (ignored on platforms that do
   not support PROM properties).

name  String containing the name of the property.
defvalue  An integer value that is returned if the property cannot be found.

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

DESCRIPTION  The ddi_prop_get_int() and ddi_prop_get_int64() functions search for an
integer property and, if found, returns the value of the property.

Properties are searched for based on the dip, name, match_dev, and the type of the data
(integer). The property search order is as follows:
1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device
info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device
information node.
6. Return defvalue.

Usually, the match_dev argument should be set to the actual device number that this
property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY,
then ddi_prop_get_int() and ddi_prop_get_int() will match the request.
regardless of the `match_dev` the property was created with. If a property was created 
with `match_dev` set to `DDI_DEV_T_NONE`, then the only way to look up this property 
is with a `match_dev` set to `DDI_DEV_T_ANY`. PROM properties are always created with 
`match_dev` set to `DDI_DEV_T_NONE`.

`name` must always be set to the name of the property being looked up.

The return value of the routine is the value of the property. If the property is not 
found, the argument `defvalue` is returned as the value of the property.

`ddi_prop_get_int64()` will not search the PROM for 64-bit property values.

**RETURN VALUES**

`ddi_prop_get_int()` and `ddi_prop_get_int64()` return the value of the 
property. If the property is not found, the argument `defvalue` is returned. If the 
property is found, but cannot be decoded into an `int` or an `int64`, then 
`DDI_PROP_NOT_FOUND` is returned.

**CONTEXT**

`ddi_prop_get_int()` and `ddi_prop_get_int64()` can be called from user or 
kernel context.

**EXAMPLES**

**EXAMPLE 1 Using ddi_prop_get_int()**

The following example demonstrates the use of `ddi_prop_get_int()`.

```c
/* 
 * Get the value of the integer "width" property, using 
 * our own default if no such property exists 
 */
width = ddi_prop_get_int(xx_dev, xx_dip, 0, "width", 
XX_DEFAULT_WIDTH);
```

**SEE ALSO**

`ddi_prop_exists(9F), ddi_prop_lookup(9F), ddi_prop_remove(9F), 
ddi_prop_update(9F)`

*Writing Device Drivers*
ddi_prop_lookup(9F)

NAME  ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_int64_array, 
ddi_prop_lookup_string_array, ddi_prop_lookup_string, 
ddi_prop_lookup_byte_array, ddi_prop_free - look up property information

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip, 
   uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip, 
   uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip, 
   uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, 
   uint_t flags, char *name, char **datap);

void ddi_prop_free(void *data);

PARAMETERS  
match_dev  Device number associated with property or DDI_DEV_T_ANY.

dip  Pointer to the device info node of device whose property list
     should be searched.

flags  Possible flag values are some combination of:

   DDI_PROP_DONTPASS
       Do not pass request to parent device information node if the
       property is not found.

   DDI_PROP_NOTPROM
       Do not look at PROM properties (ignored on platforms that do
       not support PROM properties).

name  String containing the name of the property.

nelementsp  The address of an unsigned integer which, upon successful return,
            will contain the number of elements accounted for in the memory
            pointed at by datap. The elements are either integers, strings or
            bytes depending on the interface used.

datap  

   ddi_prop_lookup_int_array()

       The address of a pointer to an array of integers which, upon
       successful return, will point to memory containing the integer
       array property value.
ddi_prop_lookup_int64_array()
The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

ddi_prop_lookup_string_array()
The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

ddi_prop_lookup_string()
The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

ddi_prop_lookup_byte_array()
The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the dip, name, match_dev, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the match_dev argument should be set to the actual device number that this property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual match_dev the property was created with. If a property was created with match_dev set to DDI_DEV_T_NONE, then the only way to look up this property is with a match_dev set to DDI_DEV_T_ANY. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case `*datap` points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. `nelementsp` is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by `*datap`.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if one called `ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free(my_int_ptr)`.

```c
ddi_prop_lookup_int_array()
```
This routine searches for and returns an array of integer property values. An array of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap` should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.

```c
ddi_prop_lookup_int64_array()
```
This routine searches for and returns an array of 64-bit integer property values. The array is defined to be `*nelementsp` number of `int64_t` elements. `datap` should be set to the address of a pointer to an array of `int64_t`'s which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.

```c
ddi_prop_lookup_string_array()
```
This routine searches for and returns a property that is an array of strings. `datap` should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the `argv` argument to `execve(2)`.

```c
ddi_prop_lookup_string()
```
This routine searches for and returns a property that is a null-terminated string. `datap` should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.

```c
ddi_prop_lookup_byte_array()
```
This routine searches for and returns a property that is an array of bytes. `datap` should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
ddi_prop_free()  
Frees the resources associated with a property previously allocated using  
ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(),  
(ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or  
(ddi_prop_lookup_byte_array().

RETURN VALUES  
The functions ddi_prop_lookup_int_array(),  
(ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),  
(ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the  
following values:

DDI_PROP_SUCCESS  Upon success.
DDI_PROP_INVAL_ARG If an attempt is made to look up a property  
with match_dev equal to DDI_DEV_T_NONE,  
name is NULL or name is the null string.
DDI_PROP_NOT_FOUND Property not found.
DDI_PROP_UNDEFINED Property explicitly not defined (see  
(ddi_prop_undefine(9F)).
DDI_PROP_CANNOT_DECODE The value of the property cannot be  
de decoded.

CONTEXT  
These functions can be called from user or kernel context.

EXAMPLES  
EXAMPLE 1 Using ddi_prop_lookup_int_array()

The following example demonstrates the use of ddi_prop_lookup_int_array().

int *options;
int noptions;

/*
 * Get the data associated with the integer "options" property  
* array, along with the number of option integers
*/
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,  
"options", &options, &noptions) == DDI_PROP_SUCCESS) {
  /*
   * Do "our thing" with the options data from the property
   */
  xx_process_options(options, noptions);
  /*
   * Free the memory allocated for the property data
   */
  ddi_prop_free(options);
}

SEE ALSO  
execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),  
ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)

Kernel Functions for Drivers  721
ddi_prop_lookup(9F)

Writing Device Drivers
ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_int64_array,
ddi_prop_lookup_string_array, ddi_prop_lookup_string,
ddi_prop_lookup_byte_array, ddi_prop_free – look up property information

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip,
uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip,
uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip,
uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip,
uint_t flags, char *name, char **datap);

void ddi_prop_free(void *data);

PARAMETERS

match_dev Device number associated with property or DDI_DEV_T_ANY.
dip Pointer to the device info node of device whose property list
should be searched.
flags Possible flag values are some combination of:

-DDI_PROP_DONTPASS
    Do not pass request to parent device information node if the
    property is not found.

-DDI_PROP_NOTPROM
    Do not look at PROM properties (ignored on platforms that do
    not support PROM properties).

name String containing the name of the property.
nelementsp The address of an unsigned integer which, upon successful return,
will contain the number of elements accounted for in the memory
pointed at by datap. The elements are either integers, strings or
bytes depending on the interface used.
datap ddi_prop_lookup_int_array()
    The address of a pointer to an array of integers which, upon
successful return, will point to memory containing the integer
array property value.
ddi_prop_lookup_byte_array(9F)

**ddi_prop_lookup_int64_array()**

The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

**ddi_prop_lookup_string_array()**

The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to *execve*(2).

**ddi_prop_lookup_string()**

The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

**ddi_prop_lookup_byte_array()**

The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

**INTERFACE LEVEL DESCRIPTION**

Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the *dip*, *name*, *match_dev*, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If *DDI_PROP_NOTPROM* is not set, search the PROM properties (if they exist).
5. If *DDI_PROP_DONTPASS* is not set, pass this request to the parent device information node.
6. Return *DDI_PROP_NOT_FOUND*.

Usually, the *match_dev* argument should be set to the actual device number that this property is associated with. However, if the *match_dev* argument is *DDI_DEV_T_ANY*, the property look up routines will match the request regardless of the actual *match_dev* the property was created with. If a property was created with *match_dev* set to *DDI_DEV_T_NONE*, then the only way to look up this property is with a *match_dev* set to *DDI_DEV_T_ANY*. PROM properties are always created with *match_dev* set to *DDI_DEV_T_NONE*.

*name* must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case `*datap` points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. `nelementsp` is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by `*datap`.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if one called `ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free(my_int_ptr)`.

`ddi_prop_lookup_int_array()`
This routine searches for and returns an array of integer property values. An array of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap` should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.

`ddi_prop_lookup_int64_array()`
This routine searches for and returns an array of 64-bit integer property values. The array is defined to be `*nelementsp` number of `int64_t` elements. `datap` should be set to the address of a pointer to an array of `int64_t`s which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.

`ddi_prop_lookup_string_array()`
This routine searches for and returns a property that is an array of strings. `datap` should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the `argv` argument to `execve(2)`.

`ddi_prop_lookup_string()`
This routine searches for and returns a property that is a null-terminated string. `datap` should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.

`ddi_prop_lookup_byte_array()`
This routine searches for and returns a property that is an array of bytes. `datap` should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
ddi_prop_free()

Frees the resources associated with a property previously allocated using
    ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(),
    ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or
    ddi_prop_lookup_byte_array().

RETURN VALUES

The functions ddi_prop_lookup_int_array(),
    ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),
    ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the
following values:

   DDI_PROP_SUCCESS       Upon success.
   DDI_PROP_INVAL_ARG     If an attempt is made to look up a property
                          with match_dev equal to DDI_DEV_T_NONE,
                          name is NULL or name is the null string.
   DDI_PROP_NOT_FOUND     Property not found.
   DDI_PROP_UNDEFINED    Property explicitly not defined (see
                          ddi_prop_undefine(9F)).
   DDI_PROP_CANNOT_DECODE The value of the property cannot be
                          decoded.

CONTEXT

These functions can be called from user or kernel context.

EXAMPLES

EXAMPLE 1 Using ddi_prop_lookup_int_array()

The following example demonstrates the use of ddi_prop_lookup_int_array().

```c
int *options;
int noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
    "options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
    * Do "our thing" with the options data from the property
    */
    xx_process_options(options, noptions);
    /*
    * Free the memory allocated for the property data
    */
    ddi_prop_free(options);
}
```

SEE ALSO

execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),
    ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)

726 man pages section 9: DDI and DKI Kernel Functions • Last Revised 11 Apr 2001
Writing Device Drivers

ddi_prop_lookup_byte_array(9F)
NAME
  ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_int64_array,
  ddi_prop_lookup_string_array, ddi_prop_lookup_string,
  ddi_prop_lookup_byte_array, ddi_prop_free - look up property information

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip,
                               uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip,
                                   uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip,
                                   uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip,
                            uint_t flags, char *name, char **datap);

void ddi_prop_free(void *data);

PARAMETERS
match_dev  Device number associated with property or DDI_DEV_T_ANY.

  dip     Pointer to the device info node of device whose property list
          should be searched.

  flags   Possible flag values are some combination of:
          DDI_PROP_DONTPASS
          Do not pass request to parent device information node if the
          property is not found.

          DDI_PROP_NOTPROM
          Do not look at PROM properties (ignored on platforms that do
          not support PROM properties).

  name    String containing the name of the property.

  nelementsp  The address of an unsigned integer which, upon successful return,
              will contain the number of elements accounted for in the memory
              pointed at by datap. The elements are either integers, strings or
              bytes depending on the interface used.

  datap

       ddi_prop_lookup_int_array()
       The address of a pointer to an array of integers which, upon
       successful return, will point to memory containing the integer
       array property value.
ddi_prop_lookup_int64_array()
The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

ddi_prop_lookup_string_array()
The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

ddi_prop_lookup_string()
The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

ddi_prop_lookup_byte_array()
The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the dip, name, match_dev, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the match_dev argument should be set to the actual device number that this property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual match_dev the property was created with. If a property was created with match_dev set to DDI_DEV_T_NONE, then the only way to look up this property is with a match_dev set to DDI_DEV_T_NONE. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`,
`ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`,
`ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is
the address of a pointer which, upon successful return, will point to memory
containing the value of the property. In each case `datap` points to a different type of
property value. See the individual descriptions of the routines below for details on the
different return values. `nelementsp` is the address of an unsigned integer which, upon
successful return, will contain the number of integer, string or byte elements
accounted for in the memory pointed at by `datap`.

All of the property look up routines may block to allocate memory needed to hold the
value of the property.

When a driver has obtained a property with any look up routine and is finished with
that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()`
must be called with the address of the allocated property. For instance, if one called
`ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an
integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free`
(`my_int_ptr`).

`ddi_prop_lookup_int_array()`
This routine searches for and returns an array of integer property values. An array
of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap`
should be set to the address of a pointer to an array of integers which, upon
successful return, will point to memory containing the integer array value of the
property.

`ddi_prop_lookup_int64_array()`
This routine searches for and returns an array of 64-bit integer property values. The
array is defined to be `*nelementsp` number of `int64_t` elements. `datap` should be set
to the address of a pointer to an array of `int64_t`’s which, upon successful return,
will point to memory containing the integer array value of the property. This
routine will not search the PROM for 64-bit property values.

`ddi_prop_lookup_string_array()`
This routine searches for and returns a property that is an array of strings. `datap`
should be set to address of a pointer to an array of strings which, upon successful
return, will point to memory containing the array of strings. The array of strings is
formatted as an array of pointers to null-terminated strings, much like the `argv`
argument to `execve(2)`.

`ddi_prop_lookup_string()`
This routine searches for and returns a property that is a null-terminated string.
`datap` should be set to the address of a pointer to string which, upon successful
return, will point to memory containing the string value of the property.

`ddi_prop_lookup_byte_array()`
This routine searches for and returns a property that is an array of bytes. `datap`
should be set to the address of a pointer to an array of bytes which, upon successful
return, will point to memory containing the byte array value of the property.
ddi_prop_free()  
Frees the resources associated with a property previously allocated using  
 ddii_prop_lookup_int_array(), ddi_prop_lookup_int64_array(),  
 ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or  
 ddi_prop_lookup_byte_array().

RETURN VALUES  
The functions ddi_prop_lookup_int_array(),  
 ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),  
 ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the  
following values:

- **DDI_PROP_SUCCESS**  
  Upon success.

- **DDI_PROP_INVAL_ARG**  
  If an attempt is made to look up a property  
  with `match_dev` equal to DDI_DEV_T_NONE,  
  `name` is NULL or `name` is the null string.

- **DDI_PROP_NOT_FOUND**  
  Property not found.

- **DDI_PROP_UNDEFINED**  
  Property explicitly not defined (see  
  ddi_prop_undefine(9F)).

- **DDI_PROP_CANNOT_DECODE**  
  The value of the property cannot be  
  decoded.

CONTEXT  
These functions can be called from user or kernel context.

EXAMPLES  
**EXAMPLE 1 Using ddi_prop_lookup_int_array()**

The following example demonstrates the use of ddi_prop_lookup_int_array().

```c
int *options;
int noptions;

/*  
 * Get the data associated with the integer "options" property  
 * array, along with the number of option integers  
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,  
 "options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*  
    * Do "our thing" with the options data from the property  
    */
    xx_process_options(options, noptions);
    /*  
    * Free the memory allocated for the property data  
    */
    ddi_prop_free(options);
}
```

SEE ALSO  
execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),  
ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)
**NAME**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_lookup</td>
<td>Look up property information.</td>
</tr>
<tr>
<td>ddi_prop_lookup_int_array</td>
<td>Integer array property lookup.</td>
</tr>
<tr>
<td>ddi_prop_lookup_int64_array</td>
<td>Integer 64-bit array property lookup.</td>
</tr>
<tr>
<td>ddi_prop_lookup_string_array</td>
<td>String array property lookup.</td>
</tr>
<tr>
<td>ddi_prop_lookup_string</td>
<td>String property lookup.</td>
</tr>
<tr>
<td>ddi_prop_lookup_byte_array</td>
<td>Byte array property lookup.</td>
</tr>
<tr>
<td>ddi_prop_free</td>
<td>Free the property information.</td>
</tr>
</tbody>
</table>

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, int **datap, uint_t *nelementsp);
int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);
int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, char ***datap, uint_t *nelementsp);
int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, char **datap);
void ddi_prop_free(void *data);
```

**PARAMETERS**

- `match_dev`: Device number associated with property or `DDI_DEV_T_ANY`.
- `dip`: Pointer to the device info node of device whose property list should be searched.
- `flags`: Possible flag values are some combination of:
  - `DDI_PROP_DONTPASS`: Do not pass request to parent device information node if the property is not found.
  - `DDI_PROP_NOTPROM`: Do not look at PROM properties (ignored on platforms that do not support PROM properties).
- `name`: String containing the name of the property.
- `nelementsp`: The address of an unsigned integer which, upon successful return, will contain the number of elements accounted for in the memory pointed at by `datap`. The elements are either integers, strings or bytes depending on the interface used.
- `datap`: The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.
ddi_prop_lookup_int_array(9F)

**ddi_prop_lookup_int64_array()**
The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

**ddi_prop_lookup_string_array()**
The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to *execve*(2).

**ddi_prop_lookup_string()**
The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

**ddi_prop_lookup_byte_array()**
The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

### Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the *dip*, *name*, *match_dev*, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the *match_dev* argument should be set to the actual device number that this property is associated with. However, if the *match_dev* argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual *match_dev* the property was created with. If a property was created with *match_dev* set to DDI_DEV_T_NONE, then the only way to look up this property is with a *match_dev* set to DDI_DEV_T_ANY. PROM properties are always created with *match_dev* set to DDI_DEV_T_NONE.

*name* must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`,
`ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`,
`ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is
the address of a pointer which, upon successful return, will point to memory
containing the value of the property. In each case, `*datap` points to a different type of
property value. See the individual descriptions of the routines below for details on the
different return values. `nelementsp` is the address of an unsigned integer which, upon
successful return, will contain the number of integer, string or byte elements
accounted for in the memory pointed at by `*datap`.

All of the property look up routines may block to allocate memory needed to hold the
value of the property.

When a driver has obtained a property with any look up routine and is finished with
that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()`
must be called with the address of the allocated property. For instance, if one called
`ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an
integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free`
(`my_int_ptr`).

`ddi_prop_lookup_int_array()`
This routine searches for and returns an array of integer property values. An array
of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap`
should be set to the address of a pointer to an array of integers which, upon
successful return, will point to memory containing the integer array value of the
property.

`ddi_prop_lookup_int64_array()`
This routine searches for and returns an array of 64-bit integer property values. The
array is defined to `*nelementsp` number of `int64_t` elements. `datap` should be set
to the address of a pointer to an array of `int64_t`'s which, upon successful return,
will point to memory containing the integer array value of the property. This
routine will not search the PROM for 64-bit property values.

`ddi_prop_lookup_string_array()`
This routine searches for and returns a property that is an array of strings. `datap`
should be set to address of a pointer to an array of strings which, upon successful
return, will point to memory containing the array of strings. The array of strings is
formatted as an array of pointers to null-terminated strings, much like the `argv`
argument to `execve(2)`.

`ddi_prop_lookup_string()`
This routine searches for and returns a property that is a null-terminated string.
`datap` should be set to the address of a pointer to string which, upon successful
return, will point to memory containing the string value of the property.

`ddi_prop_lookup_byte_array()`
This routine searches for and returns a property that is an array of bytes. `datap`
should be set to the address of a pointer to an array of bytes which, upon successful
return, will point to memory containing the byte array value of the property.
ddi_prop_lookup_int_array(9F)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_free()</td>
<td>Frees the resources associated with a property previously allocated using ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or ddi_prop_lookup_byte_array().</td>
</tr>
</tbody>
</table>

RETURN VALUES

The functions ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the following values:

- **DDI_PROP_SUCCESS**
  - Upon success.

- **DDI_PROP_INVAL_ARG**
  - If an attempt is made to look up a property with `match_dev` equal to DDI_DEV_T_NONE, `name` is NULL or `name` is the null string.

- **DDI_PROP_NOT_FOUND**
  - Property not found.

- **DDI_PROP_UNDEFINED**
  - Property explicitly not defined (see ddi_prop_undefine(9F)).

- **DDI_PROP_CANNOT_DECODE**
  - The value of the property cannot be decoded.

CONTEXT

These functions can be called from user or kernel context.

EXAMPLES

**EXAMPLE 1** Using ddi_prop_lookup_int_array()

The following example demonstrates the use of ddi_prop_lookup_int_array().

```c
int *options;
int noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0, "options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
    * Do "our thing" with the options data from the property
    */
    xx_process_options(options, noptions);

    /*
    * Free the memory allocated for the property data
    */
    ddi_prop_free(options);
}
```

SEE ALSO

execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F), ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)
Writing Device Drivers

kernel functions for drivers

ddi_prop_lookup_int_array(9F)
### Name

`ddi_prop_lookup`, `ddi_prop_lookup_int_array`, `ddi_prop_lookup_int64_array`, `ddi_prop_lookup_string_array`, `ddi_prop_lookup_string`, `ddi_prop_lookup_byte_array`, `ddi_prop_free` - look up property information

### Synopsis

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, uint_t flags, char *name, char **datap);

void ddi_prop_free(void *data);
```

### Parameters

- **match_dev**
  - Device number associated with property or `DDI_DEV_T_ANY`.
- **dip**
  - Pointer to the device info node of device whose property list should be searched.
- **flags**
  - Possible flag values are some combination of:
    ```markdown
    DDI_PROP_DONTPASS
    Do not pass request to parent device information node if the property is not found.
    
    DDI_PROP_NOTPROM
    Do not look at PROM properties (ignored on platforms that do not support PROM properties).
    ```
- **name**
  - String containing the name of the property.
- **nelementsp**
  - The address of an unsigned integer which, upon successful return, will contain the number of elements accounted for in the memory pointed at by `datap`. The elements are either integers, strings or bytes depending on the interface used.
- **datap**
  - The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.
**ddi_prop_lookup_int64_array()**
The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

**ddi_prop_lookup_string_array()**
The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to *execve*(2).

**ddi_prop_lookup_string()**
The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

**ddi_prop_lookup_byte_array()**
The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

Solaris DDI specific (Solaris DDI).
The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the *dip*, *name*, *match_dev*, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the *match_dev* argument should be set to the actual device number that this property is associated with. However, if the *match_dev* argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual *match_dev* the property was created with. If a property was created with *match_dev* set to DDI_DEV_T_NONE, then the only way to look up this property is with a *match_dev* set to DDI_DEV_T_ANY. PROM properties are always created with *match_dev* set to DDI_DEV_T_NONE.

*name* must always be set to the name of the property being looked up.
For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, `datap` is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case `*datap` points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. `nelementsp` is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by `*datap`.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if one called `ddi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an integer, `&my_int_ptr`, then the companion free call would be `ddi_prop_free(my_int_ptr)`.

**ddi_prop_lookup_int_array()**
This routine searches for and returns an array of integer property values. An array of integers is defined to `*nelementsp` number of 4 byte long integer elements. `datap` should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.

**ddi_prop_lookup_int64_array()**
This routine searches for and returns an array of 64-bit integer property values. The array is defined to be `*nelementsp` number of int64_t elements. `datap` should be set to the address of a pointer to an array of int64_t’s which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.

**ddi_prop_lookup_string_array()**
This routine searches for and returns a property that is an array of strings. `datap` should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the `argv` argument to `execve(2)`.

**ddi_prop_lookup_string()**
This routine searches for and returns a property that is a null-terminated string. `datap` should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.

**ddi_prop_lookup_byte_array()**
This routine searches for and returns a property that is an array of bytes. `datap` should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
ddi_prop_free()

Frees the resources associated with a property previously allocated using

- ddi_prop_lookup_int_array()
- ddi_prop_lookup_int64_array()
- ddi_prop_lookup_string_array()
- ddi_prop_lookup_string()
- ddi_prop_lookup_byte_array()

RETURN VALUES

The functions ddi_prop_lookup_int_array(),
ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),

and ddi_prop_lookup_string() and ddi_prop_lookup_byte_array() return the following values:

- DDI_PROP_SUCCESS: Upon success.
- DDI_PROP_INVAL_ARG: If an attempt is made to look up a property
  with match_dev equal to DDI_DEV_T_NONE,
  name is NULL or name is the null string.
- DDI_PROP_NOT_FOUND: Property not found.
- DDI_PROP_UNDEFINED: Property explicitly not defined (see
  ddi_prop_undefine(9F)).
- DDI_PROP_CANNOT DECODE: The value of the property cannot be
  decoded.

CONTEXT

These functions can be called from user or kernel context.

EXAMPLES

**EXAMPLE 1 Using ddi_prop_lookup_int_array()**

The following example demonstrates the use of ddi_prop_lookup_int_array().

```c
int *options;
int noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
 "options", &options, &noptions) == DDI_PROP_SUCCESS) {
  /*
   * Do "our thing" with the options data from the property
   */
  xx_process_options(options, noptions);
  /*
   * Free the memory allocated for the property data
   */
  ddi_prop_free(options);
}
```

SEE ALSO

execve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),

ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)
ddi_prop_lookup_string(9F)

Writing Device Drivers
ddi_prop_lookup_string_array(9F)

NAME
ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_int64_array,
ddi_prop_lookup_string_array, ddi_prop_lookup_string,
ddi_prop_lookup_byte_array, ddi_prop_free – look up property information

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, int **datap, uint_t *nelementsp);
int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);
int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, char ***datap, uint_t *nelementsp);
int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip,
   uint_t flags, char *name, char **datap);
void ddi_prop_free(void *data);

PARAMETERS

match_dev
Device number associated with property or DDI_DEV_T_ANY.
dip
Pointer to the device info node of device whose property list
   should be searched.
flags
Possible flag values are some combination of:
   DDI_PROP_DONTPASS
   Do not pass request to parent device information node if the
   property is not found.
   DDI_PROP_NOTPROM
   Do not look at PROM properties (ignored on platforms that do
   not support PROM properties).
name
String containing the name of the property.
nelementsp
The address of an unsigned integer which, upon successful return,
   will contain the number of elements accounted for in the memory
   pointed at by datap. The elements are either integers, strings or
   bytes depending on the interface used.
datap
   ddi_prop_lookup_int_array()
   The address of a pointer to an array of integers which, upon
   successful return, will point to memory containing the integer
   array property value.
ddi_prop_lookup_string_array(9F)

 ddii_prop_lookup_int64_array()
  The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

 ddii_prop_lookup_string_array()
  The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

 ddii_prop_lookup_string()
  The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

 ddii_prop_lookup_byte_array()
  The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

INTERFACE
DESCRIPTIO

Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the dip, name, match_dev, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return DDI_PROP_NOT_FOUND.

Usually, the match_dev argument should be set to the actual device number that this property is associated with. However, if the match_dev argument is DDI_DEV_T_ANY, the property look up routines will match the request regardless of the actual match_dev the property was created with. If a property was created with match_dev set to DDI_DEV_T_NONE, then the only way to look up this property is with a match_dev set to DDI_DEV_T_ANY. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.
For the routines ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array(), datap is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case *datap points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. nelementsp is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by *datap.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling ddi_prop_free(). ddi_prop_free() must be called with the address of the allocated property. For instance, if one called ddi_prop_lookup_int_array() with datap set to the address of a pointer to an integer, &my_int_ptr, then the companion free call would be ddi_prop_free(my_int_ptr).

ddi_prop_lookup_int_array()
This routine searches for and returns an array of integer property values. An array of integers is defined to *nelementsp number of 4 byte long integer elements. datap should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.

ddi_prop_lookup_int64_array()
This routine searches for and returns an array of 64-bit integer property values. The array is defined to be *nelementsp number of int64_t elements. datap should be set to the address of a pointer to an array of int64_t's which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.

ddi_prop_lookup_string_array()
This routine searches for and returns a property that is an array of strings. datap should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the argv argument to execve(2).

ddi_prop_lookup_string()
This routine searches for and returns a property that is a null-terminated string. datap should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.

ddi_prop_lookup_byte_array()
This routine searches for and returns a property that is an array of bytes. datap should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
ddi_prop_free()
Frees the resources associated with a property previously allocated using
ddi_prop_lookup_int_array(), ddi_prop_lookup_int64_array(),
ddi_prop_lookup_string_array(), ddi_prop_lookup_string(), or
ddi_prop_lookup_byte_array().

RETURN VALUES
The functions ddi_prop_lookup_int_array(),
ddi_prop_lookup_int64_array(), ddi_prop_lookup_string_array(),
ddi_prop_lookup_string(), and ddi_prop_lookup_byte_array() return the
following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>Upon success.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to look up a property with match_dev equal to DDI_DEV_T_NONE, name is NULL or name is the null string.</td>
</tr>
<tr>
<td>DDI_PROP_NOT_FOUND</td>
<td>Property not found.</td>
</tr>
<tr>
<td>DDI_PROP_UNDEFINED</td>
<td>Property explicitly not defined (see ddi_prop_undefine(9F)).</td>
</tr>
<tr>
<td>DDI_PROP_CANNOT_DECODE</td>
<td>The value of the property cannot be decoded.</td>
</tr>
</tbody>
</table>

CONTEXT
These functions can be called from user or kernel context.

EXAMPLES
EXAMPLE 1 Using ddi_prop_lookup_int_array()
The following example demonstrates the use of ddi_prop_lookup_int_array().

```c
int *options;
int noptions;
/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
"options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
    * Do "our thing" with the options data from the property
    */
    xx_process_options(options, noptions);
    /*
    * Free the memory allocated for the property data
    */
    ddi_prop_free(options);
}
```

SEE ALSO
eexecve(2), ddi_prop_exists(9F), ddi_prop_get_int(9F),
ddi_prop_remove(9F), ddi_prop_undefine(9F), ddi_prop_update(9F)
NAME  ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all, ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS  
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);
int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *name);
int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);
int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);
void ddi_prop_remove_all(dev_info_t *dip);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_prop_create() and ddi_prop_modify() functions are obsolete. Use ddi_prop_update(9F) instead of these functions.

PARAMETERS

ddi_prop_create()

dev  dev_t of the device.
dip  dev_info_t pointer of the device.
flags  flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.
name  name of property.
valuep  pointer to property value.
length  property length.

ddi_prop_undefine()

dev  dev_t of the device.
dip  dev_info_t pointer of the device.
flags  flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.
name  name of property.

ddi_prop_modify()

dev  dev_t of the device.
dip  dev_info_t pointer of the device.
Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses `ddi_getproplen(9F)` to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver's property list associated with `dip`.

Property definitions are stacked; they are added to the beginning of the driver's property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be return to the caller.

The individual functions are described as follows:

- **ddi_prop_create()**

  Adds a property to the device's property list. If the property is not associated with any particular `dev` but is associated with the physical device itself, then the argument `dev` should be the special device `DDI_DEV_T_NONE`. If you do not have a `dev` for your device (for example during `attach(9E)` time), you can create one using `makedevice(9F)` with a major number of `DDI_MAJOR_T_UNKNOWN`. `ddi_prop_create()` will then make the correct `dev` for your device.

  For boolean properties, you must set `length` to 0. For all other properties, the `length` argument must be set to the number of bytes used by the data structure representing the property being created.

  Note that creating a property involves allocating memory for the property list, the property name and the property value. If `flags` does not contain `DDI_PROP_CANSLEEP`, `ddi_prop_create()` returns `DDI_PROP_NO_MEMORY` on memory allocation failure or `DDI_PROP_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.
ddi_prop_modify(9F)

- **ddi_prop_undefine()**: This function undefines a property, setting its value to undefined. It terminates a property search at the current devinfo node, not allowing the search to proceed up to ancestor devinfo nodes. However, it will not terminate a search when `ddi_prop_get_int64(9F)` or `ddi_prop_lookup_int64_array(9F)` routines are used for lookup of 64-bit property value. See `ddi_prop_op(9F)`.

  Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.

- **ddi_prop_modify()**: This function modifies the length and value of a property. If `ddi_prop_modify()` finds the property in the driver's property list, allocates memory for the property value and returns `DDI_PROP_SUCCESS`. If the property was not found, the function returns `DDI_PROP_NOT_FOUND`.

  Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.

- **ddi_prop_remove()**: This function unlinks a property from the device's property list. If `ddi_prop_remove()` finds the property (an exact match of both `name` and `dev`), it unlinks the property, frees its memory, and returns `DDI_PROP_SUCCESS`; otherwise, it returns `DDI_PROP_NOT_FOUND`.

  `ddi_prop_remove_all()` removes the properties of all the `dev_t`s associated with the `dip`. It is called before unloading a driver.

<table>
<thead>
<tr>
<th>Function</th>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_prop_create()</code></td>
<td><code>DDI_PROP_SUCCESS</code></td>
<td>On success.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_NO_MEMORY</code></td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_INVAL_ARG</code></td>
<td>If an attempt is made to create a property with <code>dev</code> equal to <code>DDI_DEV_T_ANY</code> or if <code>name</code> is <code>NULL</code> or <code>name</code> is the <code>NULL</code> string.</td>
</tr>
<tr>
<td><code>ddi_prop_undefine()</code></td>
<td><code>DDI_PROP_SUCCESS</code></td>
<td>On success.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_NO_MEMORY</code></td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_INVAL_ARG</code></td>
<td>If an attempt is made to create a property with <code>dev</code> <code>DDI_DEV_T_ANY</code> or if <code>name</code> is <code>NULL</code> or <code>name</code> is the <code>NULL</code> string.</td>
</tr>
<tr>
<td><code>ddi_prop_modify()</code></td>
<td><code>DDI_PROP_SUCCESS</code></td>
<td>On success.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_NO_MEMORY</code></td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td></td>
<td><code>DDI_PROP_INVAL_ARG</code></td>
<td>If an attempt is made to create a property with <code>dev</code> equal to <code>DDI_DEV_T_ANY</code> or if <code>name</code> is <code>NULL</code> or <code>name</code> is the <code>NULL</code> string.</td>
</tr>
</tbody>
</table>
ddi_prop_remove()

DDI_PROP_NOT_FOUND On property search failure.
DDI_PROP_SUCCESS On success.
DDI_PROP_INVAL_ARG If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
DDI_PROP_NOT_FOUND On property search failure.

CONTEXT If DDI_PROP_CANSLEEP is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

EXAMPLES

EXAMPLE 1 Creating a Property

The following example creates a property called nblocks for each partition on a disk.

```c
int propval = 8192;
for (minor = 0; minor < 8; minor++) {
    (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
        dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t *)&propval,
        sizeof (int));
    ...
}
```

ATTRIBUTES See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_prop_create() and ddi_prop_modify() are Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

driver.conf(4), attributes(5), attach(9E), ddi_getproplen(9F), ddi_prop_create(9F), ddi_prop_update(9F), makedevice(9F)

Writing Device Drivers
ddi_prop_op(9F)

NAME

ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen –
get property information for leaf device drivers

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name,
int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *
name, int *lengthp);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI). The ddi_getlongprop(),
ddi_getlongprop_buf(), ddi_getprop(), and ddi_getproplen() functions
are obsolete. Use ddi_prop_lookup(9F) instead of ddi_getlongprop(),
ddi_getlongprop_buf(), and ddi_getproplen(). Use ddi_prop_get_int(9F)
instead of ddi_getprop().

PARAMETERS

dev  Device number associated with property or DDI_DEV_T_ANY as the
wildcard device number.

dip  Pointer to a device info node.

prop_op  Property operator.

flags  Possible flag values are some combination of:

  DDI_PROP_DONTPASS
      do not pass request to parent device information node if property not
found

  DDI_PROP_CANSLEEP
      the routine may sleep while allocating memory

  DDI_PROP_NOTPROM
      do not look at PROM properties (ignored on architectures that do not
support PROM properties)

name  String containing the name of the property.

valuep  If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the
users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the
address of a pointer.
On exit, *lengthp will contain the property length. If prop_op is PROP_LEN_AND_VAL_BUF then before calling ddi_prop_op(), lengthp should point to an int that contains the length of callers buffer.

defvalue The value that ddi_getprop() returns if the property is not found.

**DESCRIPTION**

ddi_prop_op() gets arbitrary-size properties for leaf devices. The routine searches the device’s property list. If it does not find the property at the device level, it examines the flags argument, and if DDI_PROP_DONTPASS is set, then ddi_prop_op() returns DDI_PROP_NOT_FOUND. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns DDI_PROP_UNDEFINED. Otherwise it returns either the property length, or both the length and value of the property to the caller via the valuep and lengthp pointers, depending on the value of prop_op, as described below, and returns DDI_PROP_SUCCESS. If a property cannot be found at all, DDI_PROP_NOT_FOUND is returned.

Usually, the dev argument should be set to the actual device number that this property applies to. However, if the dev argument is DDI_DEV_T_ANY, the wildcard dev, then ddi_prop_op() will match the request based on name only (regardless of the actual dev the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in last-in, first-out (LIFO) order, next search software properties created by the system in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the prop_op argument. If prop_op is PROP_LEN, then ddi_prop_op() just sets the callers length, *lengthp, to the property length and returns the value DDI_PROP_SUCCESS to the caller. The valuep argument is not used in this case. Property lengths are 0 for boolean properties, sizeof (int) for integer properties, and size in bytes for long (variable size) properties.

If prop_op is PROP_LEN_AND_VAL_BUF, then valuep should be a pointer to a user-supplied buffer whose length should be given in *lengthp by the caller. If the requested property exists, ddi_prop_op() first sets *lengthp to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns DDI_PROP_SUCCESS. If the named property exists but the buffer supplied is too small to hold it, it returns DDI_PROP_BUF_TOO_SMALL.

If prop_op is PROP_LEN_AND_VAL_ALLOC, and the property is found, ddi_prop_op() sets *lengthp to the property length. It then attempts to allocate a buffer to return to the caller using kmem_alloc(9F) routine, so that memory can be later recycled using kmem_free(9F). The driver is expected to call kmem_free() with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets *valuep to point to the allocated buffer, copies the property value into the buffer and returns DDI_PROP_SUCCESS. Otherwise, it returns
ddi_prop_op(9F)

DDI_PROP_NO_MEMORY. Note that the flags argument may affect the behavior of memory allocation in ddi_prop_op(). In particular, if DDI_PROP_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop() returns boolean and integer-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

RETURN VALUES
ddi_prop_op() ddi_getlongprop() ddi_getlongprop_buf()

ddi_getproplen() return:
DDI_PROP_SUCCESS Property found and returned.
DDI_PROP_NOT_FOUND Property not found.
DDI_PROP_UNDEFINED Property already explicitly undefined.
DDI_PROP_NO_MEMORY Property found, but unable to allocate memory. lengthp points to the correct property length.
DDI_PROP_BUF_TOO_SMALL Property found, but the supplied buffer is too small. lengthp points to the correct property length.

ddi_getprop() returns:
The value of the property or the value passed into the routine as defvalue if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

CONTEXT
These functions can be called from user or interrupt context, provided DDI_PROP_CANSLEEP is not set; if it is set, they can be called from user context only.

ATTRIBUTES
See attributes(5) for a description of the following attributes:
**SEE ALSO** attributes(5), ddi_prop_create(9F), ddi_prop_get_int(9F), ddi_prop_lookup(9F), kmem_alloc(9F), kmem_free(9F)

*Writing Device Drivers*
ddi_prop_remove(9F)

NAME  ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all,
      ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS  #include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *
      name, caddr_t valuep, int length);

int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *
      name);

int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *
      name, caddr_t valuep, int length);

int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

void ddi_prop_remove_all(dev_info_t *dip);

INTERFACE
Solaris DDI specific (Solaris DDI). The ddi_prop_create() and
      ddi_prop_modify() functions are obsolete. Use ddi_prop_update(9F) instead of
      these functions.

PARAMETERS

ddi_prop_create()

   dev     dev_t of the device.
   dip     dev_info_t pointer of the device.
   flags   flag modifiers. The only possible flag value is
            DDI_PROP_CANSLEEP: Memory allocation may sleep.
   name    name of property.
   valuep  pointer to property value.
   length  property length.

ddi_prop_undefine()

   dev     dev_t of the device.
   dip     dev_info_t pointer of the device.
   flags   flag modifiers. The only possible flag value is
            DDI_PROP_CANSLEEP: Memory allocation may sleep.
   name    name of property.

ddi_prop_modify()

   dev     dev_t of the device.
   dip     dev_info_t pointer of the device.
**ddi_prop_remove(9F)**

| **flags** | flag modifiers. The only possible flag value is
|           | DDI_PROP_CANSLEEP: Memory allocation may sleep.
| **name** | name of property.
| **valuep** | pointer to property value.
| **length** | property length.

`ddi_prop_remove()`

| **dev** | dev_t of the device.
| **dip** | dev_info_t pointer of the device.
| **name** | name of property.

`ddi_prop_remove_all()`

| **dip** | dev_info_t pointer of the device.

**DESCRIPTION**

Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses `ddi_getproplen(9F)` to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver’s property list associated with `dip`.

Property definitions are stacked; they are added to the beginning of the driver’s property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be return to the caller.

The individual functions are described as follows:

`ddi_prop_create()`

`ddi_prop_create()` adds a property to the device’s property list. If the property is not associated with any particular `dev` but is associated with the physical device itself, then the argument `dev` should be the special device DDI_DEV_T_NONE. If you do not have a `dev` for your device (for example during `attach(9E)` time), you can create one using `makedevice(9F)` with a major number of DDI_MAJOR_T_UNKNOWN. `ddi_prop_create()` will then make the correct `dev` for your device.

For boolean properties, you must set `length` to 0. For all other properties, the `length` argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If `flags` does not contain DDI_PROP_CANSLEEP, `ddi_prop_create()` returns DDI_PROP_NO_MEMORY on memory allocation failure or DDI_PROP_SUCCESS if the allocation succeeded. If DDI_PROP_CANSLEEP was set, the caller may sleep until memory becomes available.
ddi_prop_undefine() is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. However, ddi_prop_undefine() will not terminate a search when the ddi_prop_get_int64(9F) or ddi_prop_lookup_int64_array(9F) routines are used for lookup of 64-bit property value. See ddi_prop_op(9F).

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi_prop_create().

ddi_prop_modify() modifies the length and the value of a property. If ddi_prop_modify() finds the property in the driver’s property list, allocates memory for the property value and returns DDI_PROP_SUCCESS. If the property was not found, the function returns DDI_PROP_NOT_FOUND.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi_prop_create().

ddi_prop_remove() unlinks a property from the device’s property list. If ddi_prop_remove() finds the property (an exact match of both name and dev), it unlinks the property, free its memory, and returns DDI_PROP_SUCCESS, otherwise, it returns DDI_PROP_NOT_FOUND.

ddi_prop_remove_all() removes the properties of all the dev_t’s associated with the dip. It is called before unloading a driver.

**ddi_prop_create()**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>On success.</td>
</tr>
<tr>
<td>DDI_PROP_NO_MEMORY</td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.</td>
</tr>
</tbody>
</table>

**ddi_prop_undefine()**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>On success.</td>
</tr>
<tr>
<td>DDI_PROP_NO_MEMORY</td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.</td>
</tr>
</tbody>
</table>

**ddi_prop_modify()**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>On success.</td>
</tr>
<tr>
<td>DDI_PROP_NO_MEMORY</td>
<td>On memory allocation failure.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.</td>
</tr>
</tbody>
</table>
ddi_prop_remove(9F)

**ddi_prop_remove()**

- **DDI_PROP_NOT_FOUND**: On property search failure.
- **DDI_PROP_SUCCESS**: On success.
- **DDI_PROP_INVAL_ARG**: If an attempt is made to create a property with `dev` equal to **DDI_DEV_T_ANY** or if `name` is **NULL** or `name` is the **NULL** string.
- **DDI_PROP_NOT_FOUND**: On property search failure.

**CONTEXT**

If **DDI_PROP_CANSLEEP** is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

**EXAMPLES**

**EXAMPLE 1 Creating a Property**

The following example creates a property called `nblocks` for each partition on a disk.

```c
int propval = 8192;

for (minor = 0; minor < 8; minor++) {
    (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
        dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t) &propval,
        sizeof (int));
    ...
}
```

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_prop_create() and ddi_prop_modify() are Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`driver.conf(4), attributes(5), attach(9E), ddi_getproplen(9F), ddi_prop_op(9F), ddi_prop_update(9F), makedevice(9F)`

*Writing Device Drivers*
NAME

ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all,
ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS

#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int length);

int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *
name);

int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *
name, caddr_t valuep, int length);

int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

void ddi_prop_remove_all(dev_info_t *dip);

INTERFACE

Solaris DDI specific (Solaris DDI). The ddi_prop_create() and
ddi_prop_modify() functions are obsolete. Use ddi_prop_update(9F) instead of
these functions.

PARAMETERS

ddi_prop_create()

dev dev_t of the device.
dip dev_info_t pointer of the device.
flags flag modifiers. The only possible flag value is
DDI_PROP_CANSLEEP: Memory allocation may sleep.
name name of property.
valuep pointer to property value.
length property length.

ddi_prop_undefine()

dev dev_t of the device.
dip dev_info_t pointer of the device.
flags flag modifiers. The only possible flag value is
DDI_PROP_CANSLEEP: Memory allocation may sleep.
name name of property.

ddi_prop_modify()

dev dev_t of the device.
dip dev_info_t pointer of the device.
Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses `ddi_getproplen(9F)` to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver’s property list associated with `dip`.

Property definitions are stacked; they are added to the beginning of the driver’s property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be returned to the caller.

The individual functions are described as follows:

### ddi_prop_create()

`ddi_prop_create()` adds a property to the device’s property list. If the property is not associated with any particular `dev` but is associated with the physical device itself, then the argument `dev` should be the special device `DDI_DEV_T_NONE`. If you do not have a `dev` for your device (for example during `attach(9E)` time), you can create one using `makedevice(9F)` with a major number of `DDI_MAJOR_T_UNKNOWN`. `ddi_prop_create()` will then make the correct `dev` for your device.

For boolean properties, you must set `length` to 0. For all other properties, the `length` argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If `flags` does not contain `DDI_PROP_CANSLEEP`, `ddi_prop_create()` returns `DDI_PROP_NO_MEMORY` on memory allocation failure or `DDI_PROP_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.
ddi_prop_remove_all(9F)

    ddi_prop_undefined()

    ddi_prop_undefined() is a special case of property creation where the value of
    the property is set to undefined. This property has the effect of terminating a
    property search at the current devinfo node, rather than allowing the search to
    proceed up to ancestor devinfo nodes. However, ddi_prop_undefined() will not
    terminate a search when the ddi_prop_get_int64(9F) or
    ddi_prop_lookup_int64_array(9F) routines are used for lookup of 64-bit
    property value. See ddi_prop_op(9F).

    Note that undefining properties does involve memory allocation, and therefore, is
    subject to the same memory allocation constraints as ddi_prop_create().

    ddi_prop_modify()

    ddi_prop_modify() modifies the length and the value of a property. If
    ddi_prop_modify() finds the property in the driver’s property list, allocates
    memory for the property value and returns DDI_PROP_SUCCESS. If the property
    was not found, the function returns DDI_PROP_NOT_FOUND.

    Note that modifying properties does involve memory allocation, and therefore, is
    subject to the same memory allocation constraints as ddi_prop_create().

    ddi_prop_remove()

    ddi_prop_remove() unlinks a property from the device’s property list. If
    ddi_prop_remove() finds the property (an exact match of both name and dev), it
    unlinks the property, frees its memory, and returns DDI_PROP_SUCCESS,
    otherwise, it returns DDI_PROP_NOT_FOUND.

    ddi_prop_remove_all()

    ddi_prop_remove_all() removes the properties of all the dev_t’s associated
    with the dip. It is called before unloading a driver.

    ddi_prop_create()

    DDI_PROP_SUCCESS On success.
    DDI_PROP_NO_MEMORY On memory allocation failure.
    DDI_PROP_INVAL_ARG If an attempt is made to create a property
                        with dev equal to DDI_DEV_T_ANY or if
                        name is NULL or name is the NULL string.

    ddi_prop_undefined()

    DDI_PROP_SUCCESS On success.
    DDI_PROP_NO_MEMORY On memory allocation failure.
    DDI_PROP_INVAL_ARG If an attempt is made to create a property
                        with dev DDI_DEV_T_ANY or if name is
                        NULL or name is the NULL string.

    ddi_prop_modify()

    DDI_PROP_SUCCESS On success.
    DDI_PROP_NO_MEMORY On memory allocation failure.
    DDI_PROP_INVAL_ARG If an attempt is made to create a property
                        with dev equal to DDI_DEV_T_ANY or if
                        name is NULL or name is the NULL string.
ddi_prop_remove_all(9F)

**ddi_prop_remove()**

<table>
<thead>
<tr>
<th>DDI_PROP_NOT_FOUND</th>
<th>On property search failure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>On success.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.</td>
</tr>
<tr>
<td>DDI_PROP_NOT_FOUND</td>
<td>On property search failure.</td>
</tr>
</tbody>
</table>

**CONTEXT**

If DDI_PROP_CANSLEEP is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

**EXAMPLES**

**EXAMPLE 1 Creating a Property**

The following example creates a property called `nblocks` for each partition on a disk.

```c
int propval = 8192;

for (minor = 0; minor < 8; minor++) {
    (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
                        dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t *)&propval,
                        sizeof (int));
    ...
}
```

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_prop_create() and ddi_prop_modify() are Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

driver.conf(4), attributes(5), attach(9E), ddi_getproplen(9F), ddi_prop_op(9F), ddi_prop_update(9F), makedevice(9F)

Writing Device Drivers
NAME
  ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all, ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);

int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *name);

int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);

int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

void ddi_prop_remove_all(dev_info_t *dip);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). The ddi_prop_create() and ddi_prop_modify() functions are obsolete. Use ddi_prop_update(9F) instead of these functions.

PARAMETERS
  ddi_prop_create()
  dev        dev_t of the device.
  dip        dev_info_t pointer of the device.
  flags      flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.
  name       name of property.
  valuep     pointer to property value.
  length     property length.

  ddi_prop_undefine()
  dev        dev_t of the device.
  dip        dev_info_t pointer of the device.
  flags      flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.
  name       name of property.

  ddi_prop_modify()
  dev        dev_t of the device.
  dip        dev_info_t pointer of the device.
Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses `ddi_getproplen(9F)` to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver’s property list associated with `dip`. Property definitions are stacked; they are added to the beginning of the driver’s property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be returned to the caller.

The individual functions are described as follows:

```c

ddi_prop_create()

ddi_prop_create() adds a property to the device’s property list. If the property is not associated with any particular `dev` but is associated with the physical device itself, then the argument `dev` should be the special device `DDI_DEV_T_NONE`. If you do not have a `dev` for your device (for example during `attach(9E)` time), you can create one using `make_device(9F)` with a major number of `DDI_MAJOR_T_UNKNOWN`. `ddi_prop_create()` will then make the correct `dev` for your device.

For boolean properties, you must set `length` to 0. For all other properties, the `length` argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If `flags` does not contain `DDI_PROP_CANSLEEP`, `ddi_prop_create()` returns `DDI_PROP_NO_MEMORY` on memory allocation failure or `DDI_PROP_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.
```
ddi_prop_undefine(9F)

**ddi_prop_undefine()**

**ddi_prop_undefine()** is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. However, **ddi_prop_undefine()** will not terminate a search when the **ddi_prop_get_int64(9F)** or **ddi_prop_lookup_int64_array(9F)** routines are used for lookup of 64-bit property value. See **ddi_prop_op(9F)**.

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as **ddi_prop_create()**.

**ddi_prop_modify()**

**ddi_prop_modify()** modifies the length and the value of a property. If **ddi_prop_modify()** finds the property in the driver's property list, allocates memory for the property value and returns **DDI_PROP_SUCCESS**. If the property was not found, the function returns **DDI_PROP_NOT_FOUND**.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as **ddi_prop_create()**.

**ddi_prop_remove()**

**ddi_prop_remove()** unlinks a property from the device's property list. If **ddi_prop_remove()** finds the property (an exact match of both name and dev), it unlinks the property, frees its memory, and returns **DDI_PROP_SUCCESS**, otherwise, it returns **DDI_PROP_NOT_FOUND**.

**ddi_prop_remove_all()**

**ddi_prop_remove_all()** removes the properties of all the dev_t's associated with the dip. It is called before unloading a driver.

**ddi_prop_create()**

- **DDI_PROP_SUCCESS** On success.
- **DDI_PROP_NO_MEMORY** On memory allocation failure.
- **DDI_PROP_INVAL_ARG** If an attempt is made to create a property with dev equal to **DDI_DEV_T_ANY** or if name is NULL or name is the NULL string.

**ddi_prop_undefine()**

- **DDI_PROP_SUCCESS** On success.
- **DDI_PROP_NO_MEMORY** On memory allocation failure.
- **DDI_PROP_INVAL_ARG** If an attempt is made to create a property with dev **DDI_DEV_T_ANY** or if name is NULL or name is the NULL string.

**ddi_prop_modify()**

- **DDI_PROP_SUCCESS** On success.
- **DDI_PROP_NO_MEMORY** On memory allocation failure.
- **DDI_PROP_INVAL_ARG** If an attempt is made to create a property with dev equal to **DDI_DEV_T_ANY** or if name is NULL or name is the NULL string.
ddi_prop_unde
## ddii_prop_remove()

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_PROP_NOT_FOUND</td>
<td>On property search failure.</td>
</tr>
<tr>
<td>DDI_PROP_SUCCESS</td>
<td>On success.</td>
</tr>
<tr>
<td>DDI_PROP_INVAL_ARG</td>
<td>If an attempt is made to create a property with \texttt{dev} equal to DDI_DEV_T_ANY or if \texttt{name} is NULL or \texttt{name} is the NULL string.</td>
</tr>
<tr>
<td>DDI_PROP_NOT_FOUND</td>
<td>On property search failure.</td>
</tr>
</tbody>
</table>

### CONTEXT

If \texttt{DDI_PROP_CANSLEEP} is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

### EXAMPLES

#### EXAMPLE 1 Creating a Property

The following example creates a property called \texttt{nblocks} for each partition on a disk.

```c
int propval = 8192;

for (minor = 0; minor < 8; minor++) {
    // (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
    // dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t *)&propval,
    // sizeof (int));

    ...
}
```

### ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>ddi_prop_create() and ddi_prop_modify() are Obsolete</td>
</tr>
</tbody>
</table>

### SEE ALSO

driver.conf(4), attributes(5), attach(9E), ddi_getproplen(9F), ddi_prop_op(9F), ddi_prop_update(9F), makedevice(9F)

Writing Device Drivers
## NAME

ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int, ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array, ddi_prop_update_string, ddi_prop_update_byte_array

## SYNOPSIS

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char *name, int *data, uint_t nelements);
int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int data);
int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *name, int64_t *data, uint_t nelements);
int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name, int64_t data);
int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *name, char **data, uint_t nelements);
int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name, char *data);
int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *name, uchar_t *data, uint_t nelements);
```

## PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>Device number associated with the device.</td>
</tr>
<tr>
<td>dip</td>
<td>Pointer to the device info node of device whose property list should be updated.</td>
</tr>
<tr>
<td>name</td>
<td>String containing the name of the property to be updated.</td>
</tr>
<tr>
<td>nelements</td>
<td>The number of elements contained in the memory pointed at by data.</td>
</tr>
<tr>
<td>data</td>
<td>A pointer an integer array with which to update the property.</td>
</tr>
<tr>
<td>data</td>
<td>An integer value with which to update the property.</td>
</tr>
<tr>
<td>data</td>
<td>An pointer to a 64-bit integer array with which to update the property.</td>
</tr>
<tr>
<td>data</td>
<td>A 64-bit integer value with which to update the property.</td>
</tr>
</tbody>
</table>

## ddi_prop_update_int_array() (continued)

- **data**: A pointer an integer array with which to update the property.
- **name**: String containing the name of the property to be updated.
- **dip**: Pointer to the device info node of device whose property list should be updated.
- **dev**: Device number associated with the device.
- **nelements**: The number of elements contained in the memory pointed at by data.
- **ddi_prop_update_int()**: Function to update an integer property.
- **ddi_prop_update_int64()**: Function to update a 64-bit integer property.
- **ddi_prop_update_string()**: Function to update a string property.
- **ddi_prop_update_byte_array()**: Function to update a byte array property.

Last Revised 28 Aug 2001
The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the dip, name, dev, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit integer, you must use ddi_prop_update_int64_array() or ddi_prop_update_int64(). Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the dev argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular dev, then the argument dev should be set to DDI_DEV_T_NONE. This property will then match a look up request (see ddi_prop_lookup(9F)) with the match_dev argument set to DDI_DEV_T_ANY. If no dev is available for the device (for example during attach(9E) time), one can be created using makedevice(9F) with a major number of DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct dev when creating or updating the property.

name must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(), ddi_prop_update_string(), and ddi_prop_update_byte_array(), data is a pointer which points to memory containing the value of the property. In each case
*data points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. *nelements is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), *data is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined to be *nelements of 4 byte long integer elements. *data must be a pointer to an integer array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. *data must be an integer value with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be *nelements of int64_t integer elements. *data must be a pointer to a 64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. *data must be an int64_t value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. *data must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. *data must be a pointer to a string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. *data should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

**RETURN VALUES**

All of the property update routines return:
ddi_prop_update(9F)

DDI_PROP_SUCCESS  On success.

DDI_PROP_INVAL_ARG  If an attempt is made to update a property with name set to NULL or name set to the null string.

DDI_PROP_CANNOT_ENCODE  If the bytes of the property cannot be encoded.

CONTEXT  These functions can only be called from user or kernel context.

EXAMPLES  EXAMPLE 1 Updating Properties

The following example demonstrates the use of ddi_prop_update_int_array().

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
&options, sizeof (options) / sizeof (int));
```

SEE ALSO  execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)

Writing Device Drivers
ddi_prop_update_byte_array(9F)

NAME  ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
      ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
      ddi_prop_update_string, ddi_prop_update_byte_array

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char
*name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int
data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char
*name, int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name,
int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char
*name, char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name,
char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char
*name, uchar_t *data, uint_t nelements);

PARAMETERS

dev  Device number associated with the device.

dip  Pointer to the device info node of device whose property list
     should be updated.

name String containing the name of the property to be updated.

nelements The number of elements contained in the memory pointed at by
data.

data  A pointer an integer array with which to update the property.

ddi_prop_update_int_array()

data  An integer value with which to update the property.

ddi_prop_update_int()

data  An pointer to a 64-bit integer array with which to update the
     property.

ddi_prop_update_int64_array()

data  A 64-bit integer value with which to update the property.

ddi_prop_update_int64()

data  A pointer to a string array with which to update the property.

ddi_prop_update_string_array()
ddi_prop_update_byte_array(9F)

<table>
<thead>
<tr>
<th>data</th>
<th>A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the <code>argv</code> argument to <code>execve(2)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>A pointer to a string value with which to update the property.</td>
</tr>
<tr>
<td>data</td>
<td>A pointer to a byte array with which to update the property.</td>
</tr>
</tbody>
</table>

**Solaris DDI specific (Solaris DDI).**

The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the `dip`, `name`, `dev`, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either `ddi_prop_update_int_array()` or `ddi_prop_update_int()`. For a 64-bit integer, you must use `ddi_prop_update_int64_array()` or `ddi_prop_update_int64()`. Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the `dev` argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular `dev`, then the argument `dev` should be set to `DDI_DEV_T_NONE`. This property will then match a look up request (see `ddi_prop_lookup(9F)`) with the `match_dev` argument set to `DDI_DEV_T_ANY`. If no `dev` is available for the device (for example during `attach(9E)` time), one can be created using `makedevice(9F)` with a major number of `DDI MAJOR T UNKNOWN`. The update routines will then generate the correct `dev` when creating or updating the property.

`name` must always be set to the name of the property being updated.

For the routines `ddi_prop_update_int_array()`, `ddi_prop_update_int64_array()`, `ddi_prop_update_string_array()`, `ddi_prop_update_string()`, and `ddi_prop_update_byte_array()`, `data` is a pointer which points to memory containing the value of the property. In each case
*data points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. *nelements is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *data.

For the routines **ddi_prop_update_int()** and **ddi_prop_update_int64()**, *data is the new value of the property.

**ddi_prop_update_int_array()**

Updates or creates an array of integer property values. An array of integers is defined to be *nelements of 4 byte long integer elements. *data must be a pointer to an integer array with which to update the property.

**ddi_prop_update_int()**

Update or creates a single integer value of a property. *data must be an integer value with which to update the property.

**ddi_prop_update_int64_array()**

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be *nelements of int64_t integer elements. *data must be a pointer to a 64-bit integer array with which to update the property.

**ddi_prop_update_int64()**

Updates or creates a single 64-bit integer value of a property. *data must be an int64_t value with which to update the property.

**ddi_prop_update_string_array()**

Updates or creates a property that is an array of strings. *data must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULLterminated strings, much like the argv argument to execve(2).

**ddi_prop_update_string()**

Updates or creates a property that is a single string value. *data must be a pointer to a string with which to update the property.

**ddi_prop_update_byte_array()**

Updates or creates a property that is an array of bytes. *data should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

**RETURN VALUES**

All of the property update routines return:

---

774  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 28 Aug 2001
ddi_prop_update_byte_array(9F)

**CONTEXT**
These functions can only be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1 Updating Properties**

The following example demonstrates the use of `ddi_prop_update_int_array()`.

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
        &options, sizeof (options) / sizeof (int));
```

**SEE ALSO**
`execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)`

`Writing Device Drivers`
NAME

ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
ddi_prop_update_string, ddi_prop_update_byte_array - update properties

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char *name, int *data, uint_t nelements);
int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int data);
int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *name, int64_t *data, uint_t nelements);
int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name, int64_t data);
int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *name, char **data, uint_t nelements);
int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name, char *data);
int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *name, uchar_t *data, uint_t nelements);

PARAMETERS

dev Device number associated with the device.
dip Pointer to the device info node of device whose property list
      should be updated.
name String containing the name of the property to be updated.
nelements The number of elements contained in the memory pointed at by
data.
data A pointer an integer array with which to update the property.
data An integer value with which to update the property.
data A pointer to a 64-bit integer array with which to update the property.
data A 64-bit integer value with which to update the property.
ddi_prop_update_int64(9F)

<table>
<thead>
<tr>
<th>data</th>
<th>A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).</th>
</tr>
</thead>
</table>

ddi_prop_update_string()

<table>
<thead>
<tr>
<th>data</th>
<th>A pointer to a string value with which to update the property.</th>
</tr>
</thead>
</table>

ddi_prop_update_byte_array()

<table>
<thead>
<tr>
<th>data</th>
<th>A pointer to a byte array with which to update the property.</th>
</tr>
</thead>
</table>

Solaris DDI specific (Solaris DDI).

The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the\ dip, name, dev, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit integer, you must use ddi_prop_update_int64_array() or ddi_prop_update_int64(). Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the\ dev argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular\ dev, then the argument dev should be set to DDI_DEV_T_NONE. This property will then match a look up request (see ddi_prop_lookup(9F)) with the match_dev argument set to DDI_DEV_T_ANY. If no\ dev is available for the device (for example during attach(9E) time), one can be created using makedevice(9F) with a major number of DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct\ dev when creating or updating the property.

name must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(), ddi_prop_update_string(), and ddi_prop_update_byte_array(), data is a pointer which points to memory containing the value of the property. In each case
ddi_prop_update_int64(9F)

`*data` points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. `nelements` is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by `*data`.

For the routines `ddi_prop_update_int()` and `ddi_prop_update_int64()`, `data` is the new value of the property.

`ddi_prop_update_int_array()`

Updates or creates an array of integer property values. An array of integers is defined to be `nelements` of 4 byte long integer elements. `data` must be a pointer to an integer array with which to update the property.

`ddi_prop_update_int()`

Update or creates a single integer value of a property. `data` must be an integer value with which to update the property.

`ddi_prop_update_int64_array()`

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be `nelements` of `int64_t` integer elements. `data` must be a pointer to a 64-bit integer array with which to update the property.

`ddi_prop_update_int64()`

Updates or creates a single 64-bit integer value of a property. `data` must be an `int64_t` value with which to update the property.

`ddi_prop_update_string_array()`

Updates or creates a property that is an array of strings. `data` must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULLterminated strings, much like the `argv` argument to `execve(2)`.

`ddi_prop_update_string()`

Updates or creates a property that is a single string value. `data` must be a pointer to a string with which to update the property.

`ddi_prop_update_byte_array()`

Updates or creates a property that is an array of bytes. `data` should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

**RETURN VALUES**

All of the property update routines return:
DDI_PROP_SUCCESS        On success.
DDI_PROP_INVAL_ARG      If an attempt is made to update a property
                        with name set to NULL or name set to the
                        null string.
DDI_PROP_CANNOT_ENCODE  If the bytes of the property cannot be
                        encoded.

CONTEXT        These functions can only be called from user or kernel context.

EXAMPLES       EXAMPLE 1 Updating Properties

The following example demonstrates the use of ddi_prop_update_int_array().

```
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
                            &options, sizeof (options) / sizeof (int));
```

SEE ALSO  execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F),
          makedevice(9F)

Writing Device Drivers

Kernel Functions for Drivers  779
ddi_prop_update_int64_array(9F)

NAME  ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
       ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
       ddi_prop_update_string, ddi_prop_update_byte_array – update properties

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char *name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *name, int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name, int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *name, char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name, char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *name, uchar_t *data, uint_t nelements);

PARAMETERS

dev       Device number associated with the device.

dip       Pointer to the device info node of device whose property list
           should be updated.

name      String containing the name of the property to be updated.

nelements The number of elements contained in the memory pointed at by
           data.

data       A pointer an integer array with which to update the property.

data       An integer value with which to update the property.

data       An pointer to a 64-bit integer array with which to update the
           property.

data       A 64-bit integer value with which to update the property.
The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the dip, name, dev, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit integer, you must use ddi_prop_update_int64_array() or ddi_prop_update_int64(). Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the dev argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular dev, then the argument dev should be set to DDI_DEV_T_NONE. This property will then match a look up request (see ddi_prop_lookup(9F)) with the match_dev argument set to DDI_DEV_T_ANY. If no dev is available for the device (for example during attach(9E) time), one can be created using makedevice(9F) with a major number of DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct dev when creating or updating the property.

name must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(), ddi_prop_update_string(), and ddi_prop_update_byte_array(), data is a pointer which points to memory containing the value of the property. In each case
*data points to a different type of property value. See the individual descriptions of the
routines below for details concerning the different values. nelements is an unsigned
integer which contains the number of integer, string, or byte elements accounted for in
the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), data
is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined
to be nelements of 4 byte long integer elements. data must be a pointer to an integer
array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. data must be an integer value
with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is
defined to be nelements of int64_t integer elements. data must be a pointer to a
64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. data must be an int64_t
value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. data must be a pointer to a
string array with which to update the property. The array of strings is formatted as an
array of pointers to NULLterminated strings, much like the argv argument to
eexecve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. data must be a pointer to a
string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. data should be a pointer to a
byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value
of the property.

RETURN VALUES

All of the property update routines return:
### CONTEXT
These functions can only be called from user or kernel context.

### EXAMPLES

**EXAMPLE 1 Updating Properties**

The following example demonstrates the use of `ddi_prop_update_int_array()`.

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
   &options, sizeof (options) / sizeof (int));
```

### SEE ALSO
`execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)`

*Writing Device Drivers*
ddi_prop_update_int(9F)

NAME  ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
      ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
      ddi_prop_update_string, ddi_prop_update_byte_array – update properties

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char *
      name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *
      name, int data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *
      name, int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *
      name, int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *
      name, char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *
      name, char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *
      name, uchar_t *data, uint_t nelements);

PARAMETERS

dev     Device number associated with the device.
dip     Pointer to the device info node of device whose property list
         should be updated.
name    String containing the name of the property to be updated.
nelements The number of elements contained in the memory pointed at by
data.

ddi_prop_update_int_array()

data     A pointer an integer array with which to update the property.

ddi_prop_update_int()

data     An integer value with which to update the property.

ddi_prop_update_int64_array()

data     An pointer to a 64-bit integer array with which to update the
          property.

ddi_prop_update_int64()

data     A 64-bit integer value with which to update the property.

ddi_prop_update_string_array()
ddi_prop_update_int(9F)

data  A pointer to a string array with which to update the property. The
array of strings is formatted as an array of pointers to NULL
terminated strings, much like the argv argument to execve(2).

ddi_prop_update_string()

data  A pointer to a string value with which to update the property.

ddi_prop_update_byte_array()

data  A pointer to a byte array with which to update the property.

Solaris DDI specific (Solaris DDI).

The property update routines search for and, if found, modify the value of a given
property. Properties are searched for based on the dip, name, dev, and the type of the
data (integer, string, or byte). The driver software properties list is searched. If the
property is found, it is updated with the supplied value. If the property is not found
on this list, a new property is created with the value supplied. For example, if a driver
attempts to update the "foo" property, a property named "foo" is searched for on the
driver's software property list. If "foo" is found, the value is updated. If "foo" is not
found, a new property named "foo" is created on the driver's software property list
with the supplied value even if a "foo" property exists on another property list (such
as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A
property should be updated using a function with the same corresponding data type
as the property value. For example, an integer property must be updated using either
ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit
integer, you must use ddi_prop_update_int64_array() or
ddi_prop_update_int64(). Attempts to update a property with a function that
does not correspond to the property data type that was used to create it results in an
undefined state.

Usually, the dev argument should be set to the actual device number that this property
is associated with. If the property is not associated with any particular dev, then the
argument dev should be set to DDI_DEV_T_NONE. This property will then match a
lookup request (see ddi_prop_lookup(9F)) with the match_dev argument set to
DDI_DEV_T_ANY. If no dev is available for the device (for example during
attach(9E) time), one can be created using makedevice(9F) with a major number of
DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct dev
when creating or updating the property.

name must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(),
 ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(),
 ddi_prop_update_string(), and ddi_prop_update_byte_array(), data is a
pointer which points to memory containing the value of the property. In each case
*data points to a different type of property value. See the individual descriptions of the
routines below for details concerning the different values. nelements is an unsigned
integer which contains the number of integer, string, or byte elements accounted for in
the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), data
is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined
to be nelements of 4 byte long integer elements. data must be a pointer to an integer
array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. data must be an integer value
with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is
defined to be nelements of int64_t integer elements. data must be a pointer to a
64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. data must be an int64_t
value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. data must be a pointer to a
string array with which to update the property. The array of strings is formatted as an
array of pointers to NULLterminated strings, much like the argv argument to
execve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. data must be a pointer to a
string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. data should be a pointer to a
byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value
of the property.

RETURN VALUES

All of the property update routines return:
ddi_prop_update_int(9F)

**DDI_PROP_SUCCESS**
On success.

**DDI_PROP_INVAL_ARG**
If an attempt is made to update a property with *name* set to NULL or *name* set to the null string.

**DDI_PROP_CANNOT_ENCODE**
If the bytes of the property cannot be encoded.

**CONTEXT**
These functions can only be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1 Updating Properties**
The following example demonstrates the use of *ddi_prop_update_int_array()*.

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
                           &options, sizeof (options) / sizeof (int));
```

**SEE ALSO**
execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)

*Writing Device Drivers*
NAME
ddi_prop_update_int_array, ddi_prop_update_int_array, ddi_prop_update_int,
ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
ddi_prop_update_string, ddi_prop_update_byte_array - update properties

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char *
    name, int *data, uint_t nelements);
int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *
    name, int data);
int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *
    name, int64_t *data, uint_t nelements);
int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *
    name, int data);
int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *
    name, char **data, uint_t nelements);
int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *
    name, char data);
int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *
    name, uchar_t *data, uint_t nelements);

PARAMETERS

dev        Device number associated with the device.
dip        Pointer to the device info node of device whose property list
            should be updated.
name       String containing the name of the property to be updated.
nelements  The number of elements contained in the memory pointed at by
            data.

ddi_prop_update_int_array()
data        A pointer an integer array with which to update the property.
ddi_prop_update_int()
data        An integer value with which to update the property.
ddi_prop_update_int64_array()
data        An pointer to a 64-bit integer array with which to update the
            property.
ddi_prop_update_int64()
data        A 64-bit integer value with which to update the property.
ddi_prop_update_string_array()
ddi_prop_update_int_array(9F)

data A pointer to a string array with which to update the property. The
array of strings is formatted as an array of pointers to NULL
terminated strings, much like the argv argument to execve(2).

data A pointer to a string value with which to update the property.

data A pointer to a byte array with which to update the property.

Solaris DDI specific (Solaris DDI).

The property update routines search for and, if found, modify the value of a given
property. Properties are searched for based on the dip, name, dev, and the type of the
data (integer, string, or byte). The driver software properties list is searched. If the
property is found, it is updated with the supplied value. If the property is not found
on this list, a new property is created with the value supplied. For example, if a driver
attempts to update the "foo" property, a property named "foo" is searched for on the
driver's software property list. If "foo" is found, the value is updated. If "foo" is not
found, a new property named "foo" is created on the driver's software property list
with the supplied value even if a "foo" property exists on another property list (such
as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A
property should be updated using a function with the same corresponding data type
as the property value. For example, an integer property must be updated using either
ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit integer, you must use ddi_prop_update_int64_array() or
ddi_prop_update_int64(). Attempts to update a property with a function that
does not correspond to the property data type that was used to create it results in an
undefined state.

Usually, the dev argument should be set to the actual device number that this property
is associated with. If the property is not associated with any particular dev, then the
argument dev should be set to DDI_DEV_T_NONE. This property will then match a
look up request (see ddi_prop_lookup(9F)) with the match_dev argument set to
DDI_DEV_T_ANY. If no dev is available for the device (for example during
attach(9E) time), one can be created using madevice(9F) with a major number of
DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct dev
when creating or updating the property.

name must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(),
ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(),
ddi_prop_update_string(), and ddi_prop_update_byte_array(), data is a
pointer which points to memory containing the value of the property. In each case
ddi_prop_update_int_array(9F)

*data points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. nelements is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), data is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined to be nelements of 4 byte long integer elements. data must be a pointer to an integer array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. data must be an integer value with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be nelements of int64_t integer elements. data must be a pointer to a 64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. data must be an int64_t value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. data must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL-terminated strings, much like the argv argument to execve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. data must be a pointer to a string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. data should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

RETURN VALUES

All of the property update routines return:
ddi_prop_update_int_array(9F)

- **DDI_PROP_SUCCESS**: On success.
- **DDI_PROP_INVAL_ARG**: If an attempt is made to update a property with `name` set to NULL or `name` set to the null string.
- **DDI_PROP_CANNOT_ENCODE**: If the bytes of the property cannot be encoded.

**CONTEXT**

These functions can only be called from user or kernel context.

**EXAMPLES**

**EXAMPLE 1 Updating Properties**

The following example demonstrates the use of `ddi_prop_update_int_array()`.

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
    &options, sizeof (options) / sizeof (int));
```

**SEE ALSO**

execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)

*Writing Device Drivers*
ddi_prop_update_string(9F)

NAME | ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
      ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
      ddi_prop_update_string, ddi_prop_update_byte_array – update properties

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char
   *name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int
   data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char
   *name, int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name,
   int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char
   *name, char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name,
   char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char
   *name, uchar_t *data, uint_t nelements);

PARAMETERS

dev | Device number associated with the device.
dip | Pointer to the device info node of device whose property list
     should be updated.
name | String containing the name of the property to be updated.
exterms | The number of elements contained in the memory pointed at by
data.
data | A pointer an integer array with which to update the property.
data | An integer value with which to update the property.
data | An pointer to a 64-bit integer array with which to update the
     property.
data | A 64-bit integer value with which to update the property.
A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to *execve*(2).

**ddi_prop_update_string()**

*data* A pointer to a string value with which to update the property.

**ddi_prop_update_byte_array()**

*data* A pointer to a byte array with which to update the property.

**ddi_prop_update_int_array()**, **ddi_prop_lookup_int64_array()**, **ddi_prop_update_string_array()**, **ddi_prop_update_string()**, and **ddi_prop_update_byte_array()** are used to update properties with integer, string, or byte data types. The *data* parameter points to memory containing the property value.

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either **ddi_prop_update_int_array()** or **ddi_prop_update_int()**. For a 64-bit integer, you must use **ddi_prop_update_int64_array()** or **ddi_prop_update_int64()**. Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the *dev* argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular *dev*, then the argument *dev* should be set to **DDI_DEV_T_NONE**. This property will then match a look up request (see **ddi_prop_lookup(9F)**) with the *match_dev* argument set to **DDI_DEV_T_ANY**. If no *dev* is available for the device (for example during *attach*(9E) time), one can be created using **makedevice(9F)** with a major number of **DDI_MAJOR_T_UNKNOWN**. The update routines will then generate the correct *dev* when creating or updating the property.

$name$ must always be set to the name of the property being updated.

For the routines **ddi_prop_update_int_array()**, **ddi_prop_lookup_int64_array()**, **ddi_prop_update_string_array()**, **ddi_prop_update_string()**, and **ddi_prop_update_byte_array()**, *data* is a pointer which points to memory containing the value of the property. In each case...
ddi_prop_update_string(9F)

*data points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. nelements is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), data is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined to be nelements of 4 byte long integer elements. data must be a pointer to an integer array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. data must be an integer value with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be nelements of int64_t integer elements. data must be a pointer to a 64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. data must be an int64_t value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. data must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULLterminated strings, much like the argv argument to execve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. data must be a pointer to a string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. data should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

RETURN VALUES

All of the property update routines return:
DDI_PROP_SUCCESS  On success.

DDI_PROP_INVAL_ARG  If an attempt is made to update a property
                    with name set to NULL or name set to the
                    null string.

DDI_PROP_CANNOT_ENCODE  If the bytes of the property cannot be
                          encoded.

CONTEXT  These functions can only be called from user or kernel context.

EXAMPLES  EXAMPLE 1 Updating Properties

The following example demonstrates the use of ddi_prop_update_int_array().

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
                           &options, sizeof (options) / sizeof (int));
```

SEE ALSO  execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F),
makedevice(9F)

Writing Device Drivers
ddi_prop_update_string_array(9F)

NAME     ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
         ddi_prop_update_string_array, ddi_prop_update_int64, ddi_prop_update_int64_array,
         ddi_prop_update_string, ddi_prop_update_byte_array – update properties

SYNOPSIS #include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, char
     *name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name, int
     data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char
     *name, int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name,
     int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char
     *name, char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name,
     char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char
     *name, uchar_t *data, uint_t nelements);

PARAMETERS  

dev        Device number associated with the device.

dip        Pointer to the device info node of device whose property list
           should be updated.

name       String containing the name of the property to be updated.

nelements  The number of elements contained in the memory pointed at by
           data.

data       A pointer an integer array with which to update the property.

data       An integer value with which to update the property.

data       An pointer to a 64-bit integer array with which to update the
           property.

data       A 64-bit integer value with which to update the property.
ddi_prop_update_string_array(9F)

**INTERFACE**

**LEVEL**

**DESCRIPTION**

**Kernel Functions for Drivers**

## ddi_prop_update_string_array()

**data**  
A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to *execve*(2).

**ddi_prop_update_string()**

**data**  
A pointer to a string value with which to update the property.

**ddi_prop_update_byte_array()**

**data**  
A pointer to a byte array with which to update the property.

Solaris DDI specific (Solaris DDI).

The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the *dip*, *name*, *dev*, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver’s software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either ddi_prop_update_int_array() or ddi_prop_update_int(). For a 64-bit integer, you must use ddi_prop_update_int64_array() or ddi_prop_update_int64(). Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the *dev* argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular *dev*, then the argument *dev* should be set to DDI_DEV_T_NONE. This property will then match a look up request (see ddi_prop_lookup(9F)) with the *match_dev* argument set to DDI_DEV_T_ANY. If no *dev* is available for the device (for example during attach(9E) time), one can be created using makedevice(9F) with a major number of DDI_MAJOR_T_UNKNOWN. The update routines will then generate the correct *dev* when creating or updating the property.

*name* must always be set to the name of the property being updated.

For the routines ddi_prop_update_int_array(), ddi_prop_lookup_int64_array(), ddi_prop_update_string_array(), ddi_prop_update_string(), and ddi_prop_update_byte_array(), *data* is a pointer which points to memory containing the value of the property. In each case
ddi_prop_update_string_array(9F)

*data* points to a different type of property value. See the individual descriptions of the
routines below for details concerning the different values. *nelements* is an unsigned
integer which contains the number of integer, string, or byte elements accounted for in
the memory pointed at by *data.

For the routines ddi_prop_update_int() and ddi_prop_update_int64(), *data*
is the new value of the property.

ddi_prop_update_int_array()

Updates or creates an array of integer property values. An array of integers is defined
to be *nelements* of 4 byte long integer elements. *data* must be a pointer to an integer
array with which to update the property.

ddi_prop_update_int()

Update or creates a single integer value of a property. *data* must be an integer value
with which to update the property.

ddi_prop_update_int64_array()

Updates or creates an array of 64-bit integer property values. An array of integers is
defined to be *nelements* of int64_t integer elements. *data* must be a pointer to a
64-bit integer array with which to update the property.

ddi_prop_update_int64()

Updates or creates a single 64-bit integer value of a property. *data* must be an int64_t
value with which to update the property.

ddi_prop_update_string_array()

Updates or creates a property that is an array of strings. *data* must be a pointer to a
string array with which to update the property. The array of strings is formatted as an
array of pointers to NULLterminated strings, much like the argv argument to
execve(2).

ddi_prop_update_string()

Updates or creates a property that is a single string value. *data* must be a pointer to a
string with which to update the property.

ddi_prop_update_byte_array()

Updates or creates a property that is an array of bytes. *data* should be a pointer to a
byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value
of the property.

RETURN VALUES All of the property update routines return:
ddi_prop_update_string_array(9F)

**DDIPROP_SUCCESS**  
On success.

**DDIPROP_INVAL_ARG**  
If an attempt is made to update a property with `name` set to NULL or `name` set to the null string.

**DDIPROP_CANNOT_ENCODE**  
If the bytes of the property cannot be encoded.

**CONTEXT**  
These functions can only be called from user or kernel context.

**EXAMPLES**  
**EXAMPLE 1** Updating Properties

The following example demonstrates the use of `ddi_prop_update_int_array()`.

```c
int options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
 &options, sizeof (options) / sizeof (int));
```

**SEE ALSO**  
execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)

*Writing Device Drivers*
ddi_ptob(9F)

NAME       ddi_btop, ddi_btopr, ddi_ptob – page size conversions
SYNOPSIS   #include <sys/ddi.h>
            #include <sys/sunddi.h>
            
            unsigned long   ddi_btop    (dev_info_t *dip, unsigned long bytes);
            unsigned long   ddi_btopr   (dev_info_t *dip, unsigned long bytes);
            unsigned long   ddi_ptob    (dev_info_t *dip, unsigned long pages);

INTERFACE   Solaris DDI specific (Solaris DDI).
DESCRIPTION  This set of routines use the parent nexus driver to perform conversions in page size units.

            ddi_btop() converts the given number of bytes to the number of memory pages that it
            corresponds to, rounding down in the case that the byte count is not a page
            multiple.

            ddi_btopr() converts the given number of bytes to the number of memory pages that it
            corresponds to, rounding up in the case that the byte count is not a page
            multiple.

            ddi_ptob() converts the given number of pages to the number of bytes that it
            corresponds to.

            Because bus nexus may possess their own hardware address translation facilities,
            these routines should be used in preference to the corresponding DDI/DKI routines
            btop(9F), btopr(9F), and ptob(9F), which only deal in terms of the pagesize of the
            main system MMU.

RETURN VALUES  ddi_btop() and ddi_btopr() return the number of corresponding pages. ddi_ptob() returns the corresponding number of bytes. There are no error return values.

CONTEXT      This function can be called from user or interrupt context.

EXAMPLES     EXAMPLE 1 Find the size (in bytes) of one page
                
                pagesize = ddi_ptob(dip, 1L);

SEE ALSO      btop(9F), btopr(9F), ptob(9F)
                
                Writing Device Drivers
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw - write data to the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

value The data to be written to the device.

dev_addr Base device address.

These routines generate a write of various sizes to the mapped memory or device register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
ddi_put16(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw - write data to the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

These routines generate a write of various sizes to the mapped memory or device register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
### ddi_put32(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put64(9F)

NAME
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw –
write data to the mapped memory address, device register or allocated DMA memory
address

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8 (ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16 (ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32 (ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64 (ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

DESCRIPTION
These routines generate a write of various sizes to the mapped memory or device
register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device
address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb 8</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw 16</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl 32</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll 64</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw –
write data to the mapped memory address, device register or allocated DMA memory
address.

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

Solaris DDI specific (Solaris DDI).

handle              The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

value               The data to be written to the device.

devo_addr            Base device address.

These routines generate a write of various sizes to the mapped memory or device
register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device
address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

These functions can be called from user, kernel, or interrupt context.

SEE ALSO
  ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
  ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
## ddi_put8(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw – write data to the mapped memory address, device register or allocated DMA memory address

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);
```

SOLARIS DDI SPECIFIC (SOLARIS DDI).

**HANDLE**

The data access handle returned from setup calls, such as

```
#include <sys/ddi.h>
#include <sys/ddi_regs_map_setup(9F).
```

**VALUE**

The data to be written to the device.

**DEV_ADDR**

Base device address.

DESCRIPTION

These routines generate a write of various sizes to the mapped memory or device register. The `ddi_put8()`, `ddi_put16()`, `ddi_put32()`, and `ddi_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, `dev_addr`.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See `sysbus(4)`, `isa(4)`, `eisa(4)`, and `sbus(4)` for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See `pci(4)`.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

```
ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)
```

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw - write data to the mapped memory address, device register or allocated DMA memory address.

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

These routines generate a write of various sizes to the mapped memory or device register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents: 
### ddi_putl(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putchar, ddi_putl, ddi_putll, ddi_putw –
write data to the mapped memory address, device register or allocated DMA memory
address

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
handle The data access handle returned from setup calls, such as
        ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

DESCRIPTION
These routines generate a write of various sizes to the mapped memory or device
register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device
address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context.
These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and
sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI
functions from a high-interrupt context. See pci(4).

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
        ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

NOTES The functions described in this manual page previously used symbolic names which
specified their data access size; the function names have been changed so they now
specify a fixed-width data size. See the following table for the new name equivalents:
### ddi_putll(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_put64(9F), ddi_putb(9F), ddi_putl(9F), ddi_putll(9F), ddi_putw(9F)

NAME

ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw - write data to the mapped memory address, device register or allocated DMA memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

PARAMETERS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

DESCRIPTION

These routines generate a write of various sizes to the mapped memory or device register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F),
  ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
ddi_putw(9F)

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>
ddi_regs_map_free(9F)

**NAME**
ddi_regs_map_free – free a previously mapped register address space

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_regs_map_free(ddi_acc_handle_t *handle);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- `handle` Pointer to a data access handle previously allocated by a call to a setup routine such as `ddi_regs_map_setup(9F)`.

**DESCRIPTION**
`ddi_regs_map_free()` frees the mapping represented by the data access handle `handle`. This function is provided for drivers preparing to detach themselves from the system, allowing them to release allocated system resources represented in the handle.

**CONTEXT**
`ddi_regs_map_free()` must be called from user or kernel context.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus, SBus, ISA, EISA</td>
</tr>
</tbody>
</table>

**SEE ALSO**
attributes(5), `ddi_regs_map_setup(9F)`

*Writing Device Drivers*
ddi_regs_map_setup(9F)

NAME    ddi_regs_map_setup – set up a mapping for a register address space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_regs_map_setup(dev_info_t *dip, uint_t rnumber, caddr_t *addrp, offset_t offset, offset_t len, ddi_device_acc_attr_t *accattrp, ddi_acc_handle_t *handlep);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip</td>
<td>Pointer to the device’s dev_info structure.</td>
</tr>
<tr>
<td>rnumber</td>
<td>Index number to the register address space set.</td>
</tr>
<tr>
<td>addrp</td>
<td>A platform-dependent value that, when added to an offset that is less than or equal to the len parameter (see below), is used for the dev_addr argument to the ddi_get, ddi_mem_get, and ddi_io_get/put routines.</td>
</tr>
<tr>
<td>offset</td>
<td>Offset into the register address space.</td>
</tr>
<tr>
<td>len</td>
<td>Length to be mapped.</td>
</tr>
<tr>
<td>accattrp</td>
<td>Pointer to a device access attribute structure of this mapping (see ddi_device_acc_attr(9S)).</td>
</tr>
<tr>
<td>handlep</td>
<td>Pointer to a data access handle.</td>
</tr>
</tbody>
</table>

DESCRIPTION

ddi_regs_map_setup() maps in the register set given by rnumber. The register number determines which register set is mapped if more than one exists.

offset specifies the starting location within the register space and len indicates the size of the area to be mapped. If len is non-zero, it overrides the length given in the register set description. If both len and offset are 0, the entire space is mapped. The base of the mapped register space is returned in addrp.

The device access attributes are specified in the location pointed by the accattrp argument (see ddi_device_acc_attr(9S) for details).

The data access handle is returned in handlep. handlep is opaque; drivers should not attempt to interpret its value. The handle is used by the system to encode information for subsequent data access function calls to maintain a consistent view between the host and the device.

RETURN VALUES

ddi_regs_map_setup() returns:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_SUCCESS</td>
<td>Successfully set up the mapping for data access.</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
<td>Invalid register number rnumber, offset offset, or length len.</td>
</tr>
</tbody>
</table>

818 man pages section 9: DDI and DKI Kernel Functions • Last Revised 20 Nov 2001
ddi_regs_map_setup(9F)

<table>
<thead>
<tr>
<th>return value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_ME_RNUMBER_RANGE</td>
<td>Invalid register number <em>rnumber</em> or unable to find <em>reg</em> property.</td>
</tr>
<tr>
<td>DDI_REGS_ACC_CONFLICT</td>
<td>Cannot enable the register mapping due to access conflicts with other enabled mappings.</td>
</tr>
</tbody>
</table>

Note that the return value DDI_ME_RNUMBER_RANGE is not supported on all platforms. Also, there is potential overlap between DDI_ME_RNUMBER_RANGE and DDI_FAILURE. Drivers should check for !=DDI_SUCCESS rather than checking for a specific failure value.

**CONTEXT**

`ddi_regs_map_setup()` must be called from user or kernel context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus, SBus, ISA, EISA</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), ddi_regs_map_free(9F), ddi_device_acc_attr(9S)

*Writing Device Drivers*
NAME  
ddi_add_intr, ddi_get_iblock_cookie, ddi_remove_intr – hardware interrupt handling routines

SYNOPSIS  
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_iblock_cookie(dev_info_t *dip, uint_t inumber,
                          ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_intr(dev_info_t *dip, uint_t inumber,
                  ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *idevice_cookiep,
                  uint_t (*int_handler) (caddr_t), caddr_t int_handler_arg);

void ddi_remove_intr(dev_info_t *dip, uint_t inumber,
                     ddi_iblock_cookie_t iblock_cookie);

INTERFACE LEVEL PARAMETERS  
Solaris DDI specific (Solaris DDI).

For ddi_get_iblock_cookie():

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookie Pointer to an interrupt block cookie.

For ddi_add_intr():

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookie Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
idevice_cookie Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.
int_handler Pointer to interrupt handler.
int_handler_arg Argument for interrupt handler.

For ddi_remove_intr():

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookie Block cookie which identifies the interrupt handler to be removed.

ddi_get_iblock_cookie() retrieves the interrupt block cookie associated with a particular interrupt specification. This routine should be called before ddi_add_intr() to retrieve the interrupt block cookie needed to initialize locks.
On a successful return, `iblock_cookiep` contains information needed for initializing locks associated with the interrupt specification corresponding to `inumber` (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize locks acquired by the interrupt routine before calling `ddi_add_intr()` which prevents a possible race condition where the driver’s interrupt handler is called immediately after the driver has called `ddi_add_intr()` but before the driver has initialized the locks. This may happen when an interrupt for a different device occurs on the same interrupt level. If the interrupt routine acquires the lock before the lock has been initialized, undefined behavior may result.

The routine `intr_handler`, with its argument `int_handler_arg`, is called upon receipt of the appropriate interrupt. The interrupt handler should return `DDI_INTR_CLAIMED` if the interrupt was claimed, `DDI_INTR_UNCLAIMED` otherwise.

If successful, `ddi_add_intr()` will return `DDI_SUCCESS`. `DDI_INTR_NOTFOUND` is returned on i86pc and sun4m architectures if the interrupt information cannot be found. If the interrupt information cannot be found on the sun4u architecture, either `DDI_INTR_NOTFOUND` or `DDI_FAILURE` can be returned.

`ddi_remove_intr()` removes an interrupt handler from the system. Unloadable drivers should call this routine during their `detach(9E)` routine to remove their interrupt handler from the system.
The device interrupt routine for this instance of the device will not execute after `ddi_remove_intr()` returns. `ddi_remove_intr()` may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to `ddi_remove_intr()` or deadlock may result.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See `sysbus(4)`, `isa(4)`, `eisa(4)`, and `sbus(4)` for details.

### RETURN VALUES

`ddi_add_intr()` and `ddi_get_iblock_cookie()` return:

- **DDI_SUCCESS**: On success.
- **DDI_INTR_NOTFOUND**: On failure to find the interrupt.
- **DDI_FAILURE**: On failure. **DDI_FAILURE** can also be returned on failure to find interrupt (`sun4u`).

### CONTEXT

`ddi_add_intr()`, `ddi_remove_intr()`, and `ddi_get_iblock_cookie()` can be called from user or kernel context.

### SEE ALSO

- `driver.conf(4)`, `eisa(4)`, `isa(4)`, `sbus(4)`, `sysbus(4)`, `attach(9E)`, `detach(9E)`, `ddi_intr_hilevel(9F)`, `mutex(9F)`, `mutex_init(9F)`, `rw_init(9F)`, `rwlock(9F)`, `ddi_idevice_cookie(9S)`

### Writing Device Drivers

### NOTES

`ddi_get_iblock_cookie()` must not be called after the driver adds an interrupt handler for the interrupt specification corresponding to `inumber`.

All consumers of these interfaces, checking return codes, should verify `return_code != DDI_SUCCESS`. Checking for specific failure codes can result in inconsistent behaviors among platforms.

### BUGS

The `idevice_cookiep` should really point to a data structure that is specific to the bus architecture that the device operates on. Currently the SBus and PCI buses are supported and a single data structure is used to describe both.
ddi_remove_minor_node(9F)

NAME

ddi_remove_minor_node – remove a minor node for this dev_info

SYNOPSIS

void ddi_remove_minor_node(dev_info_t *dip, char *name);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dip A pointer to the device’s dev_info structure.

name The name of this minor device. If name is NULL, then remove all minor data structures from this dev_info.

DESCRIPTION

ddi_remove_minor_node() removes a data structure from the linked list of minor data structures that is pointed to by the dev_info structure for this driver.

EXAMPLES

**EXAMPLE 1** Removing a minor node

This will remove a data structure describing a minor device called dev1 which is linked into the dev_info structure pointed to by dip:

```c
    ddi_remove_minor_node(dip, "dev1");
```

SEE ALSO

attach(9E), detach(9E), ddi_create_minor_node(9F)

*Writing Device Drivers*
## NAME

`ddi_remove_softinatr(9F)`

### SYNOPSIS

```c
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t *dip, int preference,
                               ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t *idp,
                      ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *device_cookiep,
                      uint_t (*int_handler)(caddr_t int_handler_arg),
                      caddr_t int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);

void ddi_trigger_softintr(ddi_softintr_t id);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

### PARAMETERS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_get_soft_iblock_cookie()</code></td>
<td></td>
</tr>
<tr>
<td><code>dip</code></td>
<td>Pointer to a <code>dev_info</code> structure.</td>
</tr>
<tr>
<td><code>preference</code></td>
<td>The type of soft interrupt to retrieve the cookie for.</td>
</tr>
<tr>
<td><code>iblock_cookiep</code></td>
<td>Pointer to a location to store the interrupt block cookie.</td>
</tr>
<tr>
<td><code>ddi_add_softintr()</code></td>
<td></td>
</tr>
<tr>
<td><code>dip</code></td>
<td>Pointer to <code>dev_info</code> structure.</td>
</tr>
<tr>
<td><code>preference</code></td>
<td>A hint value describing the type of soft interrupt to generate.</td>
</tr>
<tr>
<td><code>idp</code></td>
<td>Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.</td>
</tr>
<tr>
<td><code>iblock_cookiep</code></td>
<td>Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.</td>
</tr>
<tr>
<td><code>device_cookiep</code></td>
<td>Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).</td>
</tr>
<tr>
<td><code>int_handler</code></td>
<td>Pointer to interrupt handler.</td>
</tr>
<tr>
<td><code>int_handler_arg</code></td>
<td>Argument for interrupt handler.</td>
</tr>
<tr>
<td><code>ddi_remove_softintr()</code></td>
<td></td>
</tr>
<tr>
<td><code>id</code></td>
<td>The identifier specifying which soft interrupt handler to remove.</td>
</tr>
<tr>
<td><code>ddi_trigger_softintr()</code></td>
<td></td>
</tr>
<tr>
<td><code>id</code></td>
<td>The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.</td>
</tr>
</tbody>
</table>
**DESCRIPTION**

For **ddi_get_soft_iblock_cookie()**:

```c
ddi_get_soft_iblock_cookie() retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before ddi_add_softintr() to retrieve the interrupt block cookie needed to initialize locks (mutex(9F), rwlock(9F)) used by the software interrupt routine. **preference** determines which type of soft interrupt to retrieve the cookie for. The possible values for **preference** are:

- **DDI_SOFTINT_LOW**
  - Low priority soft interrupt.
- **DDI_SOFTINT_MED**
  - Medium priority soft interrupt.
- **DDI_SOFTINT_HIGH**
  - High priority soft interrupt.
```

On a successful return, **iblock_cookiep** contains information needed for initializing locks associated with this soft interrupt (see mutex_init(9F) and rw_init(9F)). The driver can then initialize mutexes acquired by the interrupt routine before calling **ddi_add_softintr()** which prevents a possible race condition where the driver's soft interrupt handler is called immediately *after* the driver has called **ddi_add_softintr()** but *before* the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

For **ddi_add_softintr()**:

```c
ddi_add_softintr() adds a soft interrupt to the system. The user specified hint **preference** identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for **preference** should be the same as that used in the corresponding call to ddi_get_soft_iblock_cookie(). Refer to the description of ddi_get_soft_iblock_cookie() above.
```

The value returned in the location pointed at by **idp** is the soft interrupt identifier. This value is used in later calls to **ddi_remove_softintr()** and **ddi_trigger_softintr()** to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by **iblock_cookiep** is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see mutex_init(9F) and rw_init(9F)). Note that the interrupt block cookie is normally obtained using **ddi_get_soft_iblock_cookie()** to avoid the race conditions described above (refer to the description of **ddi_get_soft_iblock_cookie()** above). For this reason, **iblock_cookiep** is no longer useful and should be set to **NULL**.

**idevice_cookiep** is not used and should be set to **NULL**.

---

**ddi_remove_softintr(9F)**

**Kernel Functions for Drivers**  825
The routine `int_handler`, with its argument `int_handler_arg`, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine `int_handler` checks this flag, reachable through `int_handler_arg`, to determine if it should claim the interrupt and do its work.

The interrupt handler must return `DDI_INTR_CLAIMED` if the interrupt was claimed, `DDI_INTR_UNCLAIMED` otherwise.

If successful, `ddi_add_softintr()` will return `DDI_SUCCESS`; if the interrupt information cannot be found, it will return `DDI_FAILURE`.

For `ddi_remove_softintr()`:

`ddi_remove_softintr()` removes a soft interrupt from the system. The soft interrupt identifier `id`, which was returned from a call to `ddi_add_softintr()`, is used to determine which soft interrupt and which soft interrupt handler to remove.

Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For `ddi_trigger_softintr()`:

`ddi_trigger_softintr()` triggers a soft interrupt. The soft interrupt identifier `id` is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using `ddi_add_softintr()`.

## RETURN VALUES

`ddi_add_softintr()` and `ddi_get_soft_ipr_cookie()` return:

- `DDI_SUCCESS` on success
- `DDI_FAILURE` on failure

## CONTEXT

These functions can be called from user or kernel context. `ddi_trigger_softintr()` may be called from high-level interrupt context as well.

## EXAMPLES

**EXAMPLE 1** device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See `ddi_intr_hilevel(9F)`.
EXAMPLE 1 device using high-level interrupts (Continued)

In the example, the high-level interrupt routine minimally services the device, and
enqueue's the data for later processing by the soft interrupt handler. If the soft
interrupt handler is not currently running, the high-level interrupt routine triggers a
soft interrupt so the soft interrupt handler can process the data. Once running, the soft
interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data
shared between the high-level interrupt handler and the soft interrupt handler. The
low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

struct xxstate {
    ...
    ddi_softintr_t id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t low_mutex;
    int softint_running;
    ...
};
struct xxstate *xsp;
static uint_t xxsoftintr(caddr_t);
static uint_t xxhighintr(caddr_t);
...

EXAMPLE 2 sample attach() routine

The following code fragment would usually appear in the driver's attach() routine. ddi_add_intr is used to add the high-level interrupt handler and
ddi_add_softintr() is used to add the low-level interrupt routine.

static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    ...
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber,
        &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
        (void *)xsp->high_iblock_cookie);

    /* add high-level routine - xxhighintr() */
    if (ddi_add_intr(dip, inumber, NULL, NULL,
        xxhighintr, (caddr_t) xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }
EXAMPLE 2 sample attach() routine  (Continued)

} /* get soft iblock cookie */
if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
    &xsp->low_iblock_cookie) != DDI_SUCCESS) {
    /* clean up */
    return (DDI_FAILURE); /* fail attach */
}

/* initialize low-level mutex */
mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
            (void *)xsp->low_iblock_cookie);

/* add low level routine - xxsoftintr() */
if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
            NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
    /* cleanup */
    return (DDI_FAILURE); /* fail attach */
}

... 

EXAMPLE 3 High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, ddi_trigger_softintr() is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    ...
    mutex_enter(&xsp->high_mutex);
    /*
    * Verify this device generated the interrupt
    * and disable the device interrupt.
    * Enqueue data for xxsoftintr() processing.
    */
    /* is xxsoftintr() already running ? */
    if (xsp->softint_running)
        need_softint = 0;
    else
        need_softint = 1;
    mutex_exit(&xsp->high_mutex);
    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
EXAMPLE 3  High-level interrupt routine  (Continued)

    ddi_trigger_softintr(xsp->id);
    ...  
    return (DDI_INTR_CLAIMED);
}

static uint_t  
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    ...  
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);
    /* verify there is work to do */
    if (work queue empty || xsp->softint_running ) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }
    xsp->softint_running = 1;
    while ( data on queue ) {
        ASSERT(mutex owned(&xsp->high_mutex));
        /* de-queue data */
        mutex_exit(&xsp->high_mutex);
        /* Process data on queue */
        mutex_enter(&xsp->high_mutex);
    }
    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);
    return (DDI_INTR_CLAIMED);
}

SEE ALSO  ddi_add_intr(9F), ddi_in_panic(9F), ddi_intr_hilevel(9F),  
ddi_remove_intr(9F), mutex_init(9F)

Writing Device Drivers

NOTES  
ldi_add_softintr() may not be used to add the same software interrupt handler  
more than once. This is true even if a different value is used for int_handler_arg in each  
of the calls to ddi_add_softintr(). Instead, the argument passed to the interrupt  
handler should indicate what service(s) the interrupt handler should perform. For
example, the argument could be a pointer to the device’s soft state structure, which could contain a `which_service` field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_trigger_softintr()`.
ddi_removing_power – check whether DDI_SUSPEND might result in power being removed from a device

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_removing_power(dev_info_t *dip);
```

**DESCRIPTION**

The `ddi_removing_power()` function indicates whether a currently pending call into a driver's `detach()` entry point with a command of DDI_SUSPEND is likely to result in power being removed from the device.

`ddi_removing_power()` can return true and power still not be removed from the device due to a failure to suspend and power off the system.

**PARAMETERS**

The `ddi_removing_power()` function supports the following parameter:

- `dip`: pointer to the device's dev_info structure

**RETURN VALUES**

The `ddi_removing_power()` function returns:

- 1: Power might be removed by the framework as a result of the pending DDI_SUSPEND call.
- 0: Power will not be removed by the framework as a result of the pending DDI_SUSPEND call.

**EXAMPLES**

**EXAMPLE 1 Protecting a Tape from Abrupt Power Removal**

A tape driver that has hardware that would damage the tape if power is removed might include this code in its `detach()` code:

```c
int xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    ...
    case DDI_SUSPEND:
        /*
         * We do not allow DDI_SUSPEND if power will be removed and
         * we have a device that damages tape when power is removed
         * We do support DDI_SUSPEND for Device Reconfiguration, however.
         */
        if (ddi_removing_power(dip) && xxdamages_tape(dip))
            return (DDI_FAILURE);
    
    ...
```
Attributes

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

See attributes(5) for descriptions of the following attributes:

See Also

attributes(5), cpr(7), attach(9E), detach(9E)

Writing Device Drivers
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

These routines generate multiple reads from the mapped memory or device register. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
**ddi_rep_get16(9F)**

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, *dev_addr*, as a memory buffer location on the device and increment its address on the next input datum. However, when the *flags* argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**RETURN VALUES**

These functions return the value read from the mapped address.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_put8(9F)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_rep_getb</code></td>
<td><code>ddi_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_rep_getw</code></td>
<td><code>ddi_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_rep_getl</code></td>
<td><code>ddi_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_rep_getll</code></td>
<td><code>ddi_rep_get64</code></td>
</tr>
</tbody>
</table>
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw,
ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory
address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,
uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during
data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data
accesses.

These routines generate multiple reads from the mapped memory or device register.
repcount data is copied from the device address, dev_addr, to the host address,
host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(),
ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits,
and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and
host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
ddi_rep_get32(9F)

When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_put8(9F)

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
</tbody>
</table>
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw,
ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory
address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
                  uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
                   uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
                   uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,
                   uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as
        ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
            Automatically increment the device address, dev_addr, during
            data accesses.

DDI_DEV_NO_AUTOINCR
            Do not advance the device address, dev_addr, during data
            accesses.

DESCRIPTION These routines generate multiple reads from the mapped memory or device register.
repcount data is copied from the device address, dev_addr, to the host address,
host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(),
ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits,
and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and
host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

`ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_put8(9F)`

NOTES

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_rep_getb</code></td>
<td><code>ddi_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_rep_getw</code></td>
<td><code>ddi_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_rep_getl</code></td>
<td><code>ddi_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_rep_getll</code></td>
<td><code>ddi_rep_get64</code></td>
</tr>
</tbody>
</table>
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.
flags Device address flags:
    DDI_DEV_AUTOINCR
      Automatically increment the device address, dev_addr, during data accesses.

    DDI_DEV_NO_AUTOINCR
      Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION These routines generate multiple reads from the mapped memory or device register. 
repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
ddi_rep_get8(9F)

When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

RETURN VALUES
These functions return the value read from the mapped address.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_put8(9F)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
</tbody>
</table>
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION

These routines generate multiple reads from the mapped memory or device register. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**RETURN VALUES**
These functions return the value read from the mapped address.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_get8(9F)`, `ddi_put8(9F)`, `ddi_regs_map_free(9F)`, `ddi_regs_map_setup(9F)`, `ddi_rep_put8(9F)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_rep_getb</code></td>
<td><code>ddi_rep_get8</code></td>
</tr>
<tr>
<td><code>ddi_rep_getw</code></td>
<td><code>ddi_rep_get16</code></td>
</tr>
<tr>
<td><code>ddi_rep_getl</code></td>
<td><code>ddi_rep_get32</code></td>
</tr>
<tr>
<td><code>ddi_rep_getll</code></td>
<td><code>ddi_rep_get64</code></td>
</tr>
</tbody>
</table>
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

INTERFACE LEVEL

PARAMETERS

NAME

SYNOPSIS

INTERFACE LEVEL

PARAMETERS

DESCRIPTION

These routines generate multiple reads from the mapped memory or device register. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr, dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**RETURN VALUES**

These functions return the value read from the mapped address.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_get8(9F)`, `ddi_put8(9F)`, `ddi_regs_map_free(9F)`, `ddi_regs_map_setup(9F)`, `ddi_rep_put8(9F)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
</tbody>
</table>
**NAME**

ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb — read data from the mapped memory address, device register or allocated DMA memory address.

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_getll(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);
```

**INTERFACE LEVEL PARAMETERS**

Solaris DDI specific (Solaris DDI).

- **handle**: The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
- **host_addr**: Base host address.
- **dev_addr**: Base device address.
- **repcount**: Number of data accesses to perform.
- **flags**: Device address flags:
  - **DDI_DEV_AUTOINCR**: Automatically increment the device address, dev_addr, during data accesses.
  - **DDI_DEV_NO_AUTOINCR**: Do not advance the device address, dev_addr, during data accesses.

**DESCRIPTION**

These routines generate multiple reads from the mapped memory or device register. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**RETURN VALUES**

These functions return the value read from the mapped address.

**CONTEXT**

These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**

`ddi_get32(9F), ddi_put32(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_put8(9F)`

**NOTES**

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
</tbody>
</table>

---

846  man pages section 9: DDI and DKI Kernel Functions • Last Revised 10 Aug 1996
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
   Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
   Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION These routines generate multiple reads from the mapped memory or device register. repcount data is copied from the device address, dev_addr, to the host address, host_addr. For each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

These functions return the value read from the mapped address.

These functions can be called from user, kernel, or interrupt context.

ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_put8(9F)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
</tbody>
</table>
ddi_report_dev – announce a device

**SYNOPSIS**
```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_report_dev(dev_info_t *dip);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- `dip`: a pointer the device’s `dev_info` structure.

**DESCRIPTION**
`ddi_report_dev()` prints a banner at boot time, announcing the device pointed to by `dip`. The banner is always placed in the system logfile (displayed by `dmesg(1M)`), but is only displayed on the console if the system was booted with the verbose (`-v`) argument.

**CONTEXT**
`ddi_report_dev()` can be called from user context.

**SEE ALSO**
`dmesg(1M), kernel(1M)`

*Writing Device Drivers*
ddi_rep_put16(9F)

NAME

ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb, ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory address, device register or allocated DMA memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
  Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
  Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION

These routines generate multiple writes to the mapped memory or device register. repcount data is copied from the host address, host_addr, to the device address, dev_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO  ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
ddi_rep_put32(9F)

NAME
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_publ, ddi_rep_putw, ddi_rep_putl, ddi_rep_putl1 – write data to the mapped memory address, device register or allocated DMA memory address

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI).

handle
The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

host_addr
Base host address.

dev_addr
Base device address.

repcount
Number of data accesses to perform.

flags
Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION
These routines generate multiple writes to the mapped memory or device register. repcount data is copied from the host address, host_addr, to the device address, dev_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb, ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

**handle** The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

**host_addr** Base host address.

**dev_addr** Base device address.

**repcount** Number of data accesses to perform.

**flags** Device address flags:

- **DDI_DEV_AUTOINCR**
  - Automatically increment the device address, dev_addr, during data accesses.

- **DDI_DEV_NO_AUTOINCR**
  - Do not advance the device address, dev_addr, during data accesses.

These routines generate multiple writes to the mapped memory or device register. repcount data is copied from the host address, host_addr, to the device address, dev_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb,
ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory
address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
                  uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
                   uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
                   uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
                   uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

**handle**
The data access handle returned from setup calls, such as
  ddi_regs_map_setup(9F).

**host_addr**
Base host address.

**dev_addr**
Base device address.

**repcount**
Number of data accesses to perform.

**flags**
Device address flags:

- **DDI_DEV_AUTOINCR**
  Automatically increment the device address, dev_addr, during
data accesses.

- **DDI_DEV_NO_AUTOINCR**
  Do not advance the device address, dev_addr, during data
  accesses.

These routines generate multiple writes to the mapped memory or device register.
repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_rep_put8, ddi_rep_put16, ddi_rep_put32,
and ddi_rep_put64 functions write 8 bits, 16 bits, 32 bits,
and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr
must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, `dev_addr`, as a memory buffer location on the device and increment its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

**CONTEXT**
These functions can be called from user, kernel, or interrupt context.

**SEE ALSO**
`ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)`

**NOTES**
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_rep_putb</code></td>
<td><code>ddi_rep_put8</code></td>
</tr>
<tr>
<td><code>ddi_rep_putw</code></td>
<td><code>ddi_rep_put16</code></td>
</tr>
<tr>
<td><code>ddi_rep_putl</code></td>
<td><code>ddi_rep_put32</code></td>
</tr>
<tr>
<td><code>ddi_rep_putll</code></td>
<td><code>ddi_rep_put64</code></td>
</tr>
</tbody>
</table>
ddi_rep_putb(9F)

NAME

ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb,
ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory
address, device register or allocated DMA memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
        uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
        uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
        uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
        uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
        ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
        Automatically increment the device address, dev_addr, during
        data accesses.

DDI_DEV_NO_AUTOINCR
        Do not advance the device address, dev_addr, during data
        accesses.

DESCRIPTION

These routines generate multiple writes to the mapped memory or device register.
repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(),
ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits,
and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr
must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

CONTEXT
These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>

ddi_rep_putb(9F)
ddi_rep_putl(9F)

NAME
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_pubh,
ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory
address, device register or allocated DMA memory address

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
                   uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
                   uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
                   uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
                   uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
handle The data access handle returned from setup calls, such as
       ddi_regs_map_setup(9F).

host_addr Base host address.

devo_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

-DDI_DEV_AUTOINCR
   Automatically increment the device address, dev_addr, during
   data accesses.

-DDI_DEV_NO_AUTOINCR
   Do not advance the device address, dev_addr, during data
   accesses.

DESCRIPTION
These routines generate multiple writes to the mapped memory or device register.
repcount data is copied from the host address, host_addr, to the device address,
devo_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(),
_ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits,
and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and host_addr
must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

CONTEXT These functions can be called from user, kernel, or interrupt context.

SEE ALSO ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

NOTES The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb, ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory address, device register or allocated DMA memory address

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

Solaris DDI specific (Solaris DDI).

**PARAMETERS**

- **handle**
  - The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
- **host_addr**
  - Base host address.
- **dev_addr**
  - Base device address.
- **repcount**
  - Number of data accesses to perform.
- **flags**
  - Device address flags:
    - DDI_DEV_AUTOINCR
      - Automatically increment the device address, *dev_addr*, during data accesses.
    - DDI_DEV_NO_AUTOINCR
      - Do not advance the device address, *dev_addr*, during data accesses.

**DESCRIPTION**

These routines generate multiple writes to the mapped memory or device register. *repcount* data is copied from the host address, *host_addr*, to the device address, *dev_addr*. For each input datum, the ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address, *dev_addr*. *dev_addr* and *host_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F),
ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>

CONTEXT
These functions can be called from user, kernel, or interrupt context.
ddi_rep_putw(9F)

NAME

ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb,
  ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory
  address, device register or allocated DMA memory address

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
  uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
  uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
  uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
  uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

handle The data access handle returned from setup calls, such as
  ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

  DDI_DEV_AUTOINCR
  Automatically increment the device address, dev_addr, during
  data accesses.

  DDI_DEV_NO_AUTOINCR
  Do not advance the device address, dev_addr, during data
  accesses.

DESCRIPTION

These routines generate multiple writes to the mapped memory or device register.
repcount data is copied from the host address, host_addr, to the device address,
dev_addr. For each input datum, the ddi_rep_put8(), ddi_rep_put16(),
  ddi_rep_put32(), and ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits,
  and 64 bits of data, respectively, to the device address, dev_addr. dev_addr and
  host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

These functions can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
ddi_root_node(9F)

NAME  

ddi_root_node – get the root of the dev_info tree

SYNOPSIS

```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

dev_info_t *ddi_root_node(void);
```

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

DESCRIPTION  

ddi_root_node() returns a pointer to the root node of the device information tree.

RETURN VALUES  

ddi_root_node() returns a pointer to a device information structure.

CONTEXT  

ddi_root_node() can be called from user or interrupt context.

SEE ALSO  

Writing Device Drivers
ddi_segmap(9F)

NAME

ddi_segmap, ddi_segmap_setup – set up a user mapping using seg_dev

SYNOPSIS

#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_segmap(dev_t dev, off_t offset, struct asp *, caddr_t *addrp,
               off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp);

int ddi_segmap_setup(dev_t dev, off_t offset, struct asp *, caddr_t *addrp,
                      off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp,
                      ddi_device_acc_attr_t *accattrp, uint_t rnumber);

INTERFACE

LEVEL

These interfaces are obsolete. See devmap(9E) for an alternative to ddi_segmap().
Use devmap_setup(9F) instead of ddi_segmap_setup().

PARAMETERS

dev The device whose memory is to be mapped.

offset The offset within device memory at which the mapping begins.

asp An opaque pointer to the user address space into which the device memory
     should be mapped.

addrp Pointer to the starting address within the user address space to which the
        device memory should be mapped.

len Length (in bytes) of the memory to be mapped.

prot A bit field that specifies the protections. Some combinations of possible
      settings are:

      PROT_READ
      Read access is desired.

      PROT_WRITE
      Write access is desired.

      PROT_EXEC
      Execute access is desired.

      PROT_USER
      User-level access is desired (the mapping is being done as a result of a
      mmap(2) system call).

      PROT_ALL
      All access is desired.

maxprot Maximum protection flag possible for attempted mapping (the

      PROT_WRITE bit may be masked out if the user opened the special file
      read-only). If (maxprot & prot) != prot then there is an access
      violation.

flags Flags indicating type of mapping. Possible values are (other bits may be
        set):
**ddi_segmap(9F)**

| MAP_PRIVATE | Changes are private. |
| MAP_SHARED  | Changes should be shared. |
| MAP_FIXED   | The user specified an address in *addrp* rather than letting the system pick and address. |

*cred* Pointer to user credential structure.

**ddi_segmap_setup()**

| dev_acc_attr | Pointer to a ddi_device_acc_attr(9S) structure which contains the device access attributes to apply to this mapping. |
| rnumber      | Index number to the register address space set. |

**DESCRIPTION**

Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use ddi_devmap_segmap(9F) instead. See ddi_devmap_segmap(9F) for details.

ddi_segmap() and ddi_segmap_setup() set up user mappings to device space. When setting up the mapping, the ddi_segmap() and ddi_segmap_setup() routines call the mmap(9E) entry point to validate the range to be mapped. When a user process accesses the mapping, the drivers mmap(9E) entry point is again called to retrieve the page frame number that needs to be loaded. The mapping translations for that page are then loaded on behalf of the driver by the DDI framework.

ddi_segmap() is typically used as the segmap(9E) entry in the cb_ops(9S) structure for those devices that do not choose to provide their own segmap(9E) entry point. However, some drivers may have their own segmap(9E) entry point to do some initial processing on the parameters and then call ddi_segmap() to establish the default memory mapping.

ddi_segmap_setup() is used in the drivers segmap(9E) entry point to set up the mapping and assign device access attributes to that mapping. rnumber specifies the register set representing the range of device memory being mapped. See ddi_device_acc_attr(9S) for details regarding what device access attributes are available.

ddi_segmap_setup() cannot be used directly in the cb_ops(9S) structure and requires a driver to have a segmap(9E) entry point.

**RETURN VALUES**

ddi_segmap() and ddi_segmap_setup() return the following values:

| 0       | Successful completion. |
| Non-zero | An error occurred. In particular, they return ENXIO if the range to be mapped is invalid. |

**CONTEXT**

ddi_segmap() and ddi_segmap_setup() can be called from user or kernel context only.
ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

mmap(2), attributes(5), devmap(9E), mmap(9E), segmap(9E), devmap_setup(9F), ddi_mapdev(9F), ch_ops(9S), ddi_device_acc_attr(9S)

Writing Device Drivers

NOTES

If driver notification of user accesses to the mappings is required, the driver should use ddi_mapdev(9F) instead.
ddi_segmap_setup(9F)

NAME
  ddi_segmap, ddi_segmap_setup – set up a user mapping using seg_dev

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_segmap(dev_t dev, off_t offset, struct as *asp, caddr_t *addrp,
  off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp);

int ddi_segmap_setup(dev_t dev, off_t offset, struct as *asp, caddr_t *addrp,
  off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp,
  ddi_device_acc_attr_t *accattrp, uint_t rnumber);

INTERFACE
  These interfaces are obsolete. See devmap(9E) for an alternative to ddi_segmap().
  Use devmap_setup(9F) instead of ddi_segmap_setup().

PARAMETERS
  dev     The device whose memory is to be mapped.
  offset  The offset within device memory at which the mapping begins.
  asp     An opaque pointer to the user address space into which the device memory
          should be mapped.
  addrp   Pointer to the starting address within the user address space to which the
          device memory should be mapped.
  len     Length (in bytes) of the memory to be mapped.
  prot    A bit field that specifies the protections. Some combinations of possible
          settings are:
          PROT_READ
            Read access is desired.
          PROT_WRITE
            Write access is desired.
          PROT_EXEC
            Execute access is desired.
          PROT_USER
            User-level access is desired (the mapping is being done as a result of a
            mmap(2) system call).
          PROT_ALL
            All access is desired.
  maxprot Maximum protection flag possible for attempted mapping (the
            PROT_WRITE bit may be masked out if the user opened the special file
            read-only). If (maxprot & prot) != prot then there is an access
            violation.
  flags   Flags indicating type of mapping. Possible values are (other bits may be
          set):
**DESCRIPTION**
Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use `ddi_devmap_segmap(9F)` instead. See `ddi_devmap_segmap(9F)` for details.

`ddi_segmap()` and `ddi_segmap_setup()` set up user mappings to device space. When setting up the mapping, the `ddi_segmap()` and `ddi_segmap_setup()` routines call the `mmap(9E)` entry point to validate the range to be mapped. When a user process accesses the mapping, the drivers `mmap(9E)` entry point is again called to retrieve the page frame number that needs to be loaded. The mapping translations for that page are then loaded on behalf of the driver by the DDI framework.

`ddi_segmap()` is typically used as the `segmap(9E)` entry in the `cb_ops(9S)` structure for those devices that do not choose to provide their own `segmap(9E)` entry point. However, some drivers may have their own `segmap(9E)` entry point to do some initial processing on the parameters and then call `ddi_segmap()` to establish the default memory mapping.

`ddi_segmap_setup()` is used in the drivers `segmap(9E)` entry point to set up the mapping and assign device access attributes to that mapping. `rnumber` specifies the register set representing the range of device memory being mapped. See `ddi_device_acc_attr(9S)` for details regarding what device access attributes are available.

`ddi_segmap_setup()` cannot be used directly in the `cb_ops(9S)` structure and requires a driver to have a `segmap(9E)` entry point.

**RETURN VALUES**
`ddi_segmap()` and `ddi_segmap_setup()` return the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion</td>
</tr>
<tr>
<td>Non-zero</td>
<td>An error occurred. In particular, they return <code>ENXIO</code> if the range to be mapped is invalid.</td>
</tr>
</tbody>
</table>

**CONTEXT**
`ddi_segmap()` and `ddi_segmap_setup()` can be called from user or kernel context only.
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  

mmap(2), attributes(5), devmap(9E), mmap(9E), segmap(9E), devmap_setup(9F), ddi_mapdev(9F), cb_ops(9S), ddi_device_acc_attr(9S)

Writing Device Drivers

NOTES  

If driver notification of user accesses to the mappings is required, the driver should use ddi_mapdev(9F) instead.
ddi_set_driver_private(9F)

| NAME | ddi_get_driver_private, ddi_set_driver_private – get or set the address of the device’s private data area |
| SYNOPSIS | `#include <sys/conf.h>`<br>`#include <sys/ddi.h>`<br>`#include <sys/sunddi.h>`<br>`void ddi_set_driver_private(dev_info_t *dip, caddr_t data);`<br>`caddr_t ddi_get_driver_private(dev_info_t *dip);` |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| PARAMETERS | ddi_get_driver_private()<br>`dip` Pointer to device information structure to get from.<br>dip Pointer to device information structure to set.<br>`data` Data area address to set. |
| DESCRIPTION | ddi_get_driver_private() returns the address of the device’s private data area from the device information structure pointed to by `dip`. ddi_set_driver_private() sets the address of the device’s private data area in the device information structure pointed to by `dip` with the value of `data`. |
| RETURN VALUES | ddi_get_driver_private() returns the contents of devi_driver_data. If ddi_set_driver_private() has not been previously called with `dip`, an unpredictable value is returned. |
| CONTEXT | These functions can be called from user or interrupt context. |
| SEE ALSO | Writing Device Drivers |
ddi_slaveonly(9F)

NAME | ddi_slaveonly – tell if a device is installed in a slave access only location

SYNOPSIS
```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_slaveonly(dev_info_t *dip);
```

INTERFACE LEVEL | Solaris DDI specific (Solaris DDI).

PARAMETERS
| dip | A pointer to the device’s dev_info structure.

DESCRIPTION | ddi_slaveonly() tells the caller if the bus, or part of the bus that the device is installed on, does not permit the device to become a DMA master, that is, whether the device has been installed in a slave access only slot.

RETURN VALUES
| DDI_SUCCESS | The device has been installed in a slave access only location.
| DDI_FAILURE | The device has not been installed in a slave access only location.

CONTEXT | ddi_slaveonly() can be called from user or interrupt context.

SEE ALSO | Writing Device Drivers
ddi_soft_state(9F)

NAME
ddi_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free,
ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state, int item, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).
PARAMETERS
state_p Address of the opaque state pointer which will be initialized by
ddi_soft_state_init() to point to implementation dependent data.
size Size of the item which will be allocated by subsequent calls to
ddi_soft_state_zalloc()
n_items A hint of the number of items which will be preallocated; zero is allowed.
state An opaque pointer to implementation-dependent data that describes the
soft state.
item The item number for the state structure; usually the instance number of the
associated devinfo node.

DESCRIPTION
Most device drivers maintain state information with each instance of the device they
control; for example, a soft copy of a device control register, a mutex that must be held
while accessing a piece of hardware, a partition table, or a unit structure. These utility
routines are intended to help device drivers manage the space used by the driver to
hold such state information.

For example, if the driver holds the state of each instance in a single state structure,
these routines can be used to dynamically allocate and deallocate a separate structure
for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the
implementation uses as a place to hang a set of per-driver structures; everything else is
managed by these routines.

The routine ddi_soft_state_init() is usually called in the driver's _init(9E)
routine to initialize the state pointer, set the size of the soft state structure, and to allow
the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the driver's
attach(9E) routine. The routine is passed an item number which is used to refer to
the structure in subsequent calls to ddi_get_soft_state() and
ddi_soft_state_free(). The item number is usually just the instance number of
the devinfo node, obtained with ddi_get_instance(). The routine attempts to allocate space for the new structure, and if the space allocation was successful, DDI_SUCCESS is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling ddi_get_soft_state() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using ddi_soft_state_free(). This routine is usually called from the driver’s detach() entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using ddi_soft_state_fini(). This routine can be called from the driver’s _fini() routine.

The ddi_soft_state_zalloc(), ddi_soft_state_free() and ddi_get_soft_state() routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_get_soft_state()</td>
<td>NULL The requested state structure was not allocated at the time of the call.</td>
</tr>
<tr>
<td>pointer</td>
<td>The pointer to the state structure.</td>
</tr>
<tr>
<td>ddi_soft_state_init()</td>
<td>0 The allocation was successful.</td>
</tr>
<tr>
<td></td>
<td>EINVAL Either the size parameter was zero, or the state_p parameter was invalid.</td>
</tr>
<tr>
<td>ddi_soft_state_zalloc()</td>
<td>DDI_SUCCESS The allocation was successful.</td>
</tr>
<tr>
<td></td>
<td>DDI_FAILURE The routine failed to allocate the storage required; either the state parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.</td>
</tr>
</tbody>
</table>

**CONTEXT**

ddi_soft_state_init(), and ddi_soft_state_alloc() can be called from user context only, since they may internally call kmem_zalloc() with the KM_SLEEP flag.

The ddi_soft_state_fini(), ddi_soft_state_free() and ddi_get_soft_state() routines can be called from any driver context.
EXAMPLE 1 Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

typedef struct {
    volatile caddr_t *csr;  /* device registers */
    kmutex_t csr_mutex;     /* protects 'csr' field */
    unsigned int state;
    dev_info_t *dip;       /* back pointer to devinfo */
} devstate_t;

static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof(devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (error);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ... 
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
}
 static int
 xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
 {
  int instance;

  switch (cmd) {
    case DDI_DETACH:
      instance = ddi_get_instance(dip);
      ...
      ddi_soft_state_free(statep, instance);
      return (DDI_SUCCESS);
    default:
      return (DDI_FAILURE);
  }
}

 static int
 xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
 {
  devstate_t *softc;
  int instance;

  instance = getminor(*devp);
  if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
    return (ENXIO);
  ...
  softc->state |= XX_IN_USE;
  ...
  return (0);
}

SEE ALSO  _fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
           getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS  There is no attempt to validate the item parameter given to
          ddi_soft_state_zalloc() other than it must be a positive signed integer.
          Therefore very large item numbers may cause the driver to hang forever waiting for
          virtual memory resources that can never be satisfied.

NOTES    If necessary, a hierarchy of state structures can be constructed by embedding state
          pointers in higher order state structures.

DIAGNOSTICS  All of the messages described below usually indicate bugs in the driver and should
              not appear in normal operation of the system.
The implementation-dependent information kept in the state variable is corrupt.

The routine has been passed a null or corrupt state pointer. Check that ddi_soft_state_init() has been called.

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_soft_state_fini(9F)

NAME  ddi_soft_state, ddi_get_soft_state, ddi_soft_state_init, ddi_soft_state_free, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void *state);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);
```

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state_p</td>
<td>Address of the opaque state pointer which will be initialized by ddi_soft_state_init() to point to implementation dependent data.</td>
</tr>
<tr>
<td>size</td>
<td>Size of the item which will be allocated by subsequent calls to ddi_soft_state_zalloc().</td>
</tr>
<tr>
<td>n_items</td>
<td>A hint of the number of items which will be preallocated; zero is allowed.</td>
</tr>
<tr>
<td>state</td>
<td>An opaque pointer to implementation-dependent data that describes the soft state.</td>
</tr>
<tr>
<td>item</td>
<td>The item number for the state structure; usually the instance number of the associated devinfo node.</td>
</tr>
</tbody>
</table>

DESCRIPTION

Most device drivers maintain state information with each instance of the device they control; for example, a soft copy of a device control register, a mutex that must be held while accessing a piece of hardware, a partition table, or a unit structure. These utility routines are intended to help device drivers manage the space used by the driver to hold such state information.

For example, if the driver holds the state of each instance in a single state structure, these routines can be used to dynamically allocate and deallocate a separate structure for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.

The routine ddi_soft_state_init() is usually called in the driver’s _init(9E) routine to initialize the state pointer, set the size of the soft state structure, and to allow the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the driver’s attach(9E) routine. The routine is passed an item number which is used to refer to the structure in subsequent calls to ddi_get_soft_state() and ddi_soft_state_free(). The item number is usually just the instance number of
the devinfo node, obtained with ddi_get_instance(9F). The routine attempts to allocate space for the new structure, and if the space allocation was successful, DDI_SUCCESS is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling ddi_get_soft_state() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using ddi_soft_state_free(). This routine is usually called from the driver’s detach(9E) entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using ddi_soft_state_fini(). This routine can be called from the driver’s _fini(9E) routine.

The ddi_soft_state_zalloc(), ddi_soft_state_free() and ddi_get_soft_state() routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

RETURN VALUES

ddi_get_soft_state()

NULL    The requested state structure was not allocated at the time of the call.

pointer  The pointer to the state structure.

ddi_soft_state_init()

0      The allocation was successful.

EINVAL    Either the size parameter was zero, or the state_p parameter was invalid.

ddi_soft_state_zalloc()

DDI_SUCCESS    The allocation was successful.

DDI_FAILURE    The routine failed to allocate the storage required; either the state parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

CONTEXT

ddi_soft_state_init(), and ddi_soft_state_alloc() can be called from user context only, since they may internally call kmem_zalloc(9F) with the KM_SLEEP flag.

The ddi_soft_state_fini(), ddi_soft_state_free() and ddi_get_soft_state() routines can be called from any driver context.
The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

```c
typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex; /* protects 'csr' field */
    unsigned int state;
    dev_info_t *dip; /* back pointer to devinfo */
} devstate_t;
static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (error);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ...
        return (DDI_SUCCESS);
        default:
            return (DDI_FAILURE);
    }
}
```

EXAMPLE 1 Creating and Removing Data Structures
EXAMPLE 1 Creating and Removing Data Structures  (Continued)

static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {
        case DDI_DETACH:
            instance = ddi_get_instance(dip);
            ...
            ddi_soft_state_free(statep, instance);
            return (DDI_SUCCESS);
        default:
            return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
    ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}

SEE ALSO
    _fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
    getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS
    There is no attempt to validate the item parameter given to
    ddi_soft_state_zalloc() other than it must be a positive signed integer.
    Therefore very large item numbers may cause the driver to hang forever waiting for
    virtual memory resources that can never be satisfied.

NOTES
    If necessary, a hierarchy of state structures can be constructed by embedding state
    pointers in higher order state structures.

DIAGNOSTICS
    All of the messages described below usually indicate bugs in the driver and should
    not appear in normal operation of the system.
The implementation-dependent information kept in the state variable is corrupt.

The routine has been passed a null or corrupt state pointer. Check that `ddi_soft_state_init()` has been called.

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_soft_state_free(9F)

NAME

ddi_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free, ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state_p);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI).

state_p Address of the opaque state pointer which will be initialized by ddi_soft_state_init() to point to implementation dependent data.

size Size of the item which will be allocated by subsequent calls to ddi_soft_state_zalloc().

n_items A hint of the number of items which will be preallocated; zero is allowed.

state An opaque pointer to implementation-dependent data that describes the soft state.

item The item number for the state structure; usually the instance number of the associated devinfo node.

DESCRIPTION

Most device drivers maintain state information with each instance of the device they control; for example, a soft copy of a device control register, a mutex that must be held while accessing a piece of hardware, a partition table, or a unit structure. These utility routines are intended to help device drivers manage the space used by the driver to hold such state information.

For example, if the driver holds the state of each instance in a single state structure, these routines can be used to dynamically allocate and deallocate a separate structure for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.

The routine ddi_soft_state_init() is usually called in the driver’s _init(9E) routine to initialize the state pointer, set the size of the soft state structure, and to allow the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the driver’s attach(9E) routine. The routine is passed an item number which is used to refer to the structure in subsequent calls to ddi_get_soft_state() and ddi_soft_state_free(). The item number is usually just the instance number of
The devinfo node, obtained with ddi_get_instance(9F). The routine attempts to allocate space for the new structure, and if the space allocation was successful, DDI_SUCCESS is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling ddi_get_soft_state() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using ddi_soft_state_free(). This routine is usually called from the driver's detach(9E) entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using ddi_soft_state_fini(). This routine can be called from the driver's _fini(9E) routine.

The ddi_soft_state_zalloc(), ddi_soft_state_free() and ddi_get_soft_state() routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

**RETURN VALUES**

**ddi_get_soft_state()**

NULL The requested state structure was not allocated at the time of the call.

*pointer* The pointer to the state structure.

**ddi_soft_state_init()**

0 The allocation was successful.

EINVAL Either the *size* parameter was zero, or the *state_p* parameter was invalid.

**ddi_soft_state_zalloc()**

DDI_SUCCESS The allocation was successful.

DDI_FAILURE The routine failed to allocate the storage required; either the *state* parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

**CONTEXT**

ddi_soft_state_init() and ddi_soft_state_alloc() can be called from user context only, since they may internally call kmem_zalloc(9F) with the KM_SLEEP flag.

The ddi_soft_state_fini(), ddi_soft_state_free() and ddi_get_soft_state() routines can be called from any driver context.
EXAMPLE 1 Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex; /* protects 'csr' field */
    unsigned int state;
    dev_info_t *dip; /* back pointer to devinfo */
} devstate_t;

static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof(devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (0);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ...
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
}
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {

        case DDI_DETACH:
            instance = ddi_get_instance(dip);
            ...
            ddi_soft_state_free(statep, instance);
            return (DDI_SUCCESS);

        default:
            return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
    ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}

SEE ALSO
_ddi_soft_state_free(9F), _fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS
There is no attempt to validate the item parameter given to
ddi_soft_state_zalloc() other than it must be a positive signed integer.
Therefore very large item numbers may cause the driver to hang forever waiting for
virtual memory resources that can never be satisfied.

NOTES
If necessary, a hierarchy of state structures can be constructed by embedding state
pointers in higher order state structures.

DIAGNOSTICS
All of the messages described below usually indicate bugs in the driver and should
not appear in normal operation of the system.
The implementation-dependent information kept in the state variable is corrupt.

The routine has been passed a null or corrupt state pointer. Check that `ddi_soft_state_init()` has been called.

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_soft_state_init(9F)

NAME  ddi_soft_state, ddi_get_soft_state, ddi_soft_state_init, ddi_soft_state_free,
       ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
Parameters

PARAMETERS

state_p  Address of the opaque state pointer which will be initialized by
         ddi_soft_state_init() to point to implementation dependent data.

size  Size of the item which will be allocated by subsequent calls to
       ddi_soft_state_zalloc().

n_items  A hint of the number of items which will be preallocated; zero is allowed.

state  An opaque pointer to implementation-dependent data that describes the
       soft state.

item  The item number for the state structure; usually the instance number of the
       associated devinfo node.

DESCRIPTION  Most device drivers maintain state information with each instance of the device
            they control; for example, a soft copy of a device control register, a mutex that
            must be held while accessing a piece of hardware, a partition table, or a unit
            structure. These utility routines are intended to help device drivers manage
            the space used by the driver to hold such state information.

            For example, if the driver holds the state of each instance in a single state
            structure, these routines can be used to dynamically allocate and deallocate a
            separate structure for each instance of the driver as the instance is attached and
            detached.

            To use the routines, the driver writer needs to declare a state pointer, state_p,
            which the implementation uses as a place to hang a set of per-driver structures;
            everything else is managed by these routines.

            The routine ddi_soft_state_init() is usually called in the driver’s _init(9E)
            routine to initialize the state pointer, set the size of the soft state structure,
            and to allow the driver to pre-allocate a given number of such structures if
            required.

            The routine ddi_soft_state_zalloc() is usually called in the driver’s
            attach(9E) routine. The routine is passed an item number which is used to refer to
            the structure in subsequent calls to ddi_get_soft_state() and
            ddi_soft_state_free(). The item number is usually just the instance number of

890  man pages section 9: DDI and DKI Kernel Functions • Last Revised 7 Jun 1993
the devinfo node, obtained with ddi_get_instance(9F). The routine attempts to allocate space for the new structure, and if the space allocation was successful, DDI_SUCCESS is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling ddi_get_soft_state() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using ddi_soft_state_free(). This routine is usually called from the driver’s detach(9E) entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using ddi_soft_state_fini(). This routine can be called from the driver’s _fini(9E) routine.

The ddi_soft_state_zalloc(), ddi_soft_state_free() and ddi_get_soft_state() routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

RETURN VALUES

ddi_get_soft_state()

NULL The requested state structure was not allocated at the time of the call.

pointer The pointer to the state structure.

ddi_soft_state_init()

0 The allocation was successful.

EINVAL Either the size parameter was zero, or the state_p parameter was invalid.

ddi_soft_state_zalloc()

DDI_SUCCESS The allocation was successful.

DDI_FAILURE The routine failed to allocate the storage required; either the state parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

CONTEXT
ddi_soft_state_init(), and ddi_soft_state_alloc() can be called from user context only, since they may internally call kmem_zalloc(9F) with the KM_SLEEP flag.

The ddi_soft_state_fini(), ddi_soft_state_free() and ddi_get_soft_state() routines can be called from any driver context.
**EXAMPLE 1** Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

```c
typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex; /* protects 'csr' field */
    unsigned int state;
    dev_info_t *dip; /* back pointer to devinfo */
} devstate_t;

static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (error);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ...  
        return (DDI_SUCCESS);
        default:
            return (DDI_FAILURE);
    }
}
```
EXAMPLE 1 Creating and Removing Data Structures  (Continued)

    static int
    xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
    {
        int instance;

        switch (cmd) {
            case DDI_DETACH:
                instance = ddi_get_instance(dip);
                ...
                ddi_soft_state_free(statep, instance);
                return (DDI_SUCCESS);
            default:
                return (DDI_FAILURE);
        }
    }

    static int
    xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
    {
        devstate_t *softc;
        int instance;

        instance = getminor(*devp);
        if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
            return (ENXIO);
        ...
        softc->state |= XX_IN_USE;
        ...
        return (0);
    }

SEE ALSO  _fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
          getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS  There is no attempt to validate the item parameter given to
          ddi_soft_state_zalloc() other than it must be a positive signed integer.
          Therefore very large item numbers may cause the driver to hang forever waiting for
          virtual memory resources that can never be satisfied.

NOTES  If necessary, a hierarchy of state structures can be constructed by embedding state
        pointers in higher order state structures.

DIAGNOSTICS  All of the messages described below usually indicate bugs in the driver and should
             not appear in normal operation of the system.
The implementation-dependent information kept in the state variable is corrupt.

The routine has been passed a null or corrupt state pointer. Check that ddi_soft_state_init() has been called.

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_soft_state_zalloc(9F)

NAME
ddi_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free,
ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state_p);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
state_p Address of the opaque state pointer which will be initialized by
        ddi_soft_state_init() to point to implementation dependent data.
size Size of the item which will be allocated by subsequent calls to
        ddi_soft_state_zalloc().
n_items A hint of the number of items which will be preallocated; zero is allowed.
state An opaque pointer to implementation-dependent data that describes the
        soft state.
item The item number for the state structure; usually the instance number of the
        associated devinfo node.

DESCRIPTION
Most device drivers maintain state information with each instance of the device they
control; for example, a soft copy of a device control register, a mutex that must be held
while accessing a piece of hardware, a partition table, or a unit structure. These utility
routines are intended to help device drivers manage the space used by the driver to
hold such state information.

For example, if the driver holds the state of each instance in a single state structure,
these routines can be used to dynamically allocate and deallocate a separate structure
for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the
implementation uses as a place to hang a set of per-driver structures; everything else is
managed by these routines.

The routine ddi_soft_state_init() is usually called in the driver's _init(9E)
routine to initialize the state pointer, set the size of the soft state structure, and to allow
the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the driver's
attach(9E) routine. The routine is passed an item number which is used to refer to
the structure in subsequent calls to ddi_get_soft_state() and
iddi_soft_state_free(). The item number is usually just the instance number of
the devinfo node, obtained with ddi_get_instance(9F). The routine attempts to
allocate space for the new structure, and if the space allocation was successful,
DDI_SUCCESS is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by
calling ddi_get_soft_state() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using
ddi_soft_state_free(). This routine is usually called from the driver’s
detach(9E) entry point.

The space used by all the soft state structures allocated on a given state pointer,
共同 with the housekeeping information used by the implementation can be
returned to the system using ddi_soft_state_fini(). This routine can be called
from the driver’s _fini(9E) routine.

The ddi_soft_state_zalloc(), ddi_soft_state_free() and
ddi_get_soft_state() routines coordinate access to the underlying data
structures in an MT-safe fashion, thus no additional locks should be necessary.

RETURN VALUES

ddi_get_soft_state()

NULL The requested state structure was not allocated at the time of the
call.

pointer The pointer to the state structure.

ddi_soft_state_init()

0 The allocation was successful.

EINVAL Either the size parameter was zero, or the state_p parameter was
invalid.

ddi_soft_state_zalloc()

DDI_SUCCESS The allocation was successful.

DDI_FAILURE The routine failed to allocate the storage required; either the state
parameter was invalid, the item number was negative, or an
attempt was made to allocate an item number that was already
allocated.

CONTEXT
ddi_soft_state_init(), and ddi_soft_state_alloc() can be called from
user context only, since they may internally call kmem_zalloc(9F) with the
KM_SLEEP flag.

The ddi_soft_state_fini(), ddi_soft_state_free() and
ddi_get_soft_state() routines can be called from any driver context.
EXAMPLE 1 Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver’s data structures.

typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex; /* protects 'csr' field */
    unsigned int state;
    dev_info_t *dip; /* back pointer to devinfo */
} devstate_t;
static void *statep;

int _init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (0);
}

int _fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ... return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
EXAMPLE 1 Creating and Removing Data Structures  (Continued)

```c
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {
        case DDI_DETACH:
            instance = ddi_get_instance(dip);
            ...
            ddi_soft_state_free(statep, instance);
            return (DDI_SUCCESS);
        default:
            return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
    ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}
```

SEE ALSO
_fini(9E), _init(9E), attach(9E), detach(9E), ddi_get_instance(9F),
getminor(9F), kmem_zalloc(9F)

Writing Device Drivers

WARNINGS
There is no attempt to validate the item parameter given to
_ddi_soft_state_zalloc() other than it must be a positive signed integer.
Therefore very large item numbers may cause the driver to hang forever waiting for
virtual memory resources that can never be satisfied.

NOTES
If necessary, a hierarchy of state structures can be constructed by embedding state
pointers in higher order state structures.

DIAGNOSTICS
All of the messages described below usually indicate bugs in the driver and should
not appear in normal operation of the system.
ddi_soft_state_zalloc(9F)

WARNING: ddi_soft_state_zalloc: bad handle
WARNING: ddi_soft_state_free: bad handle
WARNING: ddi_soft_state_fini: bad handle

The implementation-dependent information kept in the state variable is corrupt.

WARNING: ddi_soft_state_free: null handle
WARNING: ddi_soft_state_fini: null handle

The routine has been passed a null or corrupt state pointer. Check that ddi_soft_state_init() has been called.

WARNING: ddi_soft_state_free: item %d not in range [0..%d]

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
ddi_trigger_softintr(9F)

NAME  ddi_add_softintr, ddi_get_soft_iblock_cookie, ddi_remove_softintr,
       ddi_trigger_softintr – software interrupt handling routines

SYNOPSIS  
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t *dip, int preference,
                                ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t
                      *idp, ddi_iblock_cookie_t *iblock_cookie, ddi_device_cookie_t
                      *device_cookie, uint_t(*int_handler) (caddr_t int_handler_arg),
                      caddr_t int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);
void ddi_trigger_softintr(ddi_softintr_t id);

INTERFACE
LEVEL
PARAMETERS  Solaris DDI specific (Solaris DDI).

ddi_get_soft_iblock_cookie()  
 dip  Pointer to a dev_info structure.
 preference  The type of soft interrupt to retrieve the cookie for.
 iblock_cookiep  Pointer to a location to store the interrupt block cookie.

ddi_add_softintr()  
 dip  Pointer to dev_info structure.
 preference  A hint value describing the type of soft interrupt to generate.
 idp  Pointer to a soft interrupt identifier where a returned soft interrupt
      identifier is stored.
 iblock_cookiep  Optional pointer to an interrupt block cookie where a returned
                  interrupt block cookie is stored.
 device_cookiep  Optional pointer to an interrupt device cookie where a returned
                  interrupt device cookie is stored (not used).
 int_handler  Pointer to interrupt handler.
 int_handler_arg  Argument for interrupt handler.

ddi_remove_softintr()  
 id  The identifier specifying which soft interrupt handler to remove.

ddi_trigger_softintr()  
 id  The identifier specifying which soft interrupt to trigger and which
     soft interrupt handler will be called.
**DESCRIPTION**

For `ddi_get_soft_iblock_cookie()`:

`ddi_get_soft_iblock_cookie()` retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before `ddi_add_softintr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the software interrupt routine. `preference` determines which type of soft interrupt to retrieve the cookie for. The possible values for `preference` are:

- **DDI_SOFTINT_LOW**
  - Low priority soft interrupt.

- **DDI_SOFTINT_MED**
  - Medium priority soft interrupt.

- **DDI_SOFTINT_HIGH**
  - High priority soft interrupt.

On a successful return, `iblock_cookiep` contains information needed for initializing locks associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize mutexes acquired by the interrupt routine before calling `ddi_add_softintr()` which prevents a possible race condition where the driver’s soft interrupt handler is called immediately after the driver has called `ddi_add_softintr()` but before the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

For `ddi_add_softintr()`:

`ddi_add_softintr()` adds a soft interrupt to the system. The user specified hint `preference` identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for `preference` should be the same as that used in the corresponding call to `ddi_get_soft_iblock_cookie()`. Refer to the description of `ddi_get_soft_iblock_cookie()` above.

The value returned in the location pointed at by `idp` is the soft interrupt identifier. This value is used in later calls to `ddi_remove_softintr()` and `ddi_trigger_softintr()` to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by `iblock_cookiep` is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). Note that the interrupt block cookie is normally obtained using `ddi_get_soft_iblock_cookie()` to avoid the race conditions described above (refer to the description of `ddi_get_soft_iblock_cookie()` above). For this reason, `iblock_cookiep` is no longer useful and should be set to NULL.

`idevice_cookiep` is not used and should be set to NULL.
The routine \texttt{int_handler}, with its argument \texttt{int_handler_arg}, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine \texttt{int_handler} checks this flag, reachable through \texttt{int_handler_arg}, to determine if it should claim the interrupt and do its work.

The interrupt handler must return \texttt{DDI_INTR_CLAIMED} if the interrupt was claimed, \texttt{DDI_INTR_UNCLAIMED} otherwise.

If successful, \texttt{ddi_add_softintr()} will return \texttt{DDI_SUCCESS}; if the interrupt information cannot be found, it will return \texttt{DDI_FAILURE}.

For \texttt{ddi_remove_softintr()}:

\texttt{ddi_remove_softintr()} removes a soft interrupt from the system. The soft interrupt identifier \texttt{id}, which was returned from a call to \texttt{ddi_add_softintr()}, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For \texttt{ddi_trigger_softintr()}:

\texttt{ddi_trigger_softintr()} triggers a soft interrupt. The soft interrupt identifier \texttt{id} is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using \texttt{ddi_add_softintr()}.

\textbf{RETURN VALUES} \texttt{ddi_add_softintr()} and \texttt{ddi_get_soft_iblock_cookie()} return:

\begin{itemize}
  \item \texttt{DDI_SUCCESS} on success
  \item \texttt{DDI_FAILURE} on failure
\end{itemize}

\textbf{CONTEXT} These functions can be called from user or kernel context. \texttt{ddi_trigger_softintr()} may be called from high-level interrupt context as well.

\textbf{EXAMPLES} \textbf{EXAMPLE 1} device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See \texttt{ddi_intr_hilevel(9F)}. 

\texttt{ddi_trigger_softintr(9F)}
EXAMPLE 1 device using high-level interrupts  (Continued)

In the example, the high-level interrupt routine minimally services the device, and
enqueues the data for later processing by the soft interrupt handler. If the soft
interrupt handler is not currently running, the high-level interrupt routine triggers a
soft interrupt so the soft interrupt handler can process the data. Once running, the soft
interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data
shared between the high-level interrupt handler and the soft interrupt handler. The
low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
    ...
    ddi_softintr_t id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t low_mutex;
    int softint_running;
    ...
};
struct xxstate *xsp;
static uint_t xxsoftintr(caddr_t);
static uint_t xxhighintr(caddr_t);
... 
```

EXAMPLE 2 sample attach() routine

The following code fragment would usually appear in the driver’s attach(9E)
routine. `ddi_add_intr(9F)` is used to add the high-level interrupt handler and
`ddi_add_softintr()` is used to add the low-level interrupt routine.

```
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    ...
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber,
                           &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
               (void *)&xsp->high_iblock_cookie);

    /* add high-level routine - xxhighintr() */
    if (ddi_add_intr(dip, inumber, NULL, NULL,
                     xxhighintr, (caddr_t)xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }
    ...
}```
EXAMPLE 2 sample attach() routine (Continued)

```c
} /* get soft iblock cookie */
if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
   &xsp->low_iblock_cookie) != DDI_SUCCESS) {
   /* clean up */
   return (DDI_FAILURE); /* fail attach */
}
/* initialize low-level mutex */
mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
   (void *)&xsp->low_iblock_cookie);
/* add low level routine - xxsoftintr() */
if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
    NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
   /* cleanup */
   return (DDI_FAILURE); /* fail attach */
}
```

EXAMPLE 3 High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, ddi_trigger_softintr() is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```c
static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    ...
    mutex_enter(&(xsp->high_mutex));
    /*
    * Verify this device generated the interrupt
    * and disable the device interrupt.
    * Enqueue data for xxsoftintr() processing.
    */
    /* is xxsoftintr() already running ? */
    if (xsp->softint_running) {
        need_softint = 0;
    } else {
        need_softint = 1;
        mutex_exit(&(xsp->high_mutex));
    }
    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
```
EXAMPLE 3 High-level interrupt routine (Continued)

    ddi_trigger_softintr(xsp->id);
    ...
    return (DDI_INTR_CLAIMED);
}

static uint_t
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    ...
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);
    /* verify there is work to do */
    if (work queue empty || xsp->softint_running ) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }
    xsp->softint_running = 1;
    while (data on queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));
        /* de-queue data */
        mutex_exit(&xsp->high_mutex);
        /* Process data on queue */
        mutex_enter(&xsp->high_mutex);
    }
    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);
    return (DDI_INTR_CLAIMED);
}

SEE ALSO ddi_add_intr(9F), ddi_in_panic(9F), ddi_intr_hilevel(9F),
ddi_remove_intr(9F), mutex_init(9F)

Writing Device Drivers

NOTES ddi_add_softintr() may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for int_handler_arg in each of the calls to ddi_add_softintr(). Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For
ddi_trigger_softintr(9F)

example, the argument could be a pointer to the device’s soft state structure, which could contain a ‘which_service’ field that the handler examines. The driver must set this field to the appropriate value before calling ddi_trigger_softintr().
ddi_umem_alloc(9F)

NAME
ddi_umem_alloc, ddi_umem_free - allocate and free page-aligned kernel memory

SYNOPSIS
#include <sys/types.h>
#include <sys/sunddi.h>

void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t *cookiep);

void ddi_umem_free(ddi_umem_cookie_t cookie);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
ddi_umem_alloc()

size Number of bytes to allocate.

fla Used to determine the sleep and pageable conditions.

Possible sleep flags are DDI_UMEM_SLEEP, which allows sleeping until
memory is available, and DDI_UMEM_NOSLEEP, which returns NULL
immediately if memory is not available.

The default condition is to allocate locked memory; this can be changed to
allocate pageable memory using the DDI_UMEM_PAGEABLE flag.

cookiep Pointer to a kernel memory cookie.

ddi_umem_free()

cookie A kernel memory cookie allocated in ddi_umem_alloc().

DESCRIPTION
ddi_umem_alloc() allocates page-aligned kernel memory and returns a pointer to
the allocated memory. The number of bytes allocated is a multiple of the system page
size (roundup of size). The allocated memory can be used in the kernel and can be
exported to user space. See devmap(9E) and devmap_umem_setup(9F) for further
information.

flag determines whether the caller can sleep for memory and whether the allocated
memory is locked or not. DDI_UMEM_SLEEP allocations may sleep but are guaranteed
to succeed. DDI_UMEM_NOSLEEP allocations do not sleep but may fail (return NULL) if
memory is currently unavailable. If DDI_UMEM_PAGEABLE is set, pageable memory
will be allocated. These pages can be swapped out to secondary memory devices. The
initial contents of memory allocated using ddi_umem_alloc() is zero-filled.

*cookiep is a pointer to the kernel memory cookie that describes the kernel memory
being allocated. A typical use of cookiep is in devmap_umem_setup(9F) when the
drivers want to export the kernel memory to a user application.

To free the allocated memory, a driver calls ddi_umem_free() with the cookie
obtained from ddi_umem_alloc(). ddi_umem_free() releases the entire buffer.

RETURN VALUES
Non-null Successful completion. ddi_umem_alloc() returns a pointer to
the allocated memory.

NULL Memory cannot be allocated by ddi_umem_alloc() because
DDI_UMEM_NOSLEEP is set and the system is out of resources.
ddi_umem_alloc(9F)

CONTEXT  
ddi_umem_alloc() can be called from any context if flag is set to DDI_UMEM_NOSLEEP. If DDI_UMEM_SLEEP is set, ddi_umem_alloc() can be called from user and kernel context only. ddi_umem_free() can be called from any context.

SEE ALSO  
devmap(9E), condvar(9F), devmap_umem_setup(9F), kmem_alloc(9F), mutex(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

WARNINGS  
Setting the DDI_UMEM_PAGEABLE flag in ddi_umem_alloc() will result in an allocation of pageable memory. Because these pages can be swapped out to secondary memory devices, drivers should use this flag with care. This memory must not be used for the following purposes:

- For synchronization objects such as locks and condition variables. See mutex(9F), semaphore(9F), rwlock(9F), and condvar(9F).
- For driver interrupt routines.

Memory allocated using ddi_umem_alloc() without setting DDI_UMEM_PAGEABLE flag cannot be paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.

Excessive use of kernel memory is likely to effect overall system performance. Over-commitment of kernel memory may cause unpredictable consequences.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing an invalid pointer, will cause the system to corrupt data or panic.

Do not call ddi_umem_alloc() within DDI_SUSPEND and DDI_RESUME operations. Memory acquired at these times is not reliable. In some cases, such a call can cause a system to hang.

NOTES  
ddi_umem_alloc(0, flag, cookiep) always returns NULL. ddi_umem_free(NULL) has no effects on system.
ddi_umem_free(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_umem_alloc, ddi_umem_free – allocate and free page-aligned kernel memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td></td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
<tr>
<td>ddi_umem_alloc()</td>
<td>size</td>
</tr>
<tr>
<td>flag</td>
<td>Used to determine the sleep and pageable conditions.</td>
</tr>
<tr>
<td>Possible sleep flags are DDI_UMEM_SLEEP, which allows sleeping until memory is available, and DDI_UMEM_NOSLEEP, which returns NULL immediately if memory is not available.</td>
<td></td>
</tr>
<tr>
<td>The default condition is to allocate locked memory; this can be changed to allocate pageable memory using the DDI_UMEM_PAGEABLE flag.</td>
<td></td>
</tr>
<tr>
<td>cookiep</td>
<td>Pointer to a kernel memory cookie.</td>
</tr>
<tr>
<td>ddi_umem_free()</td>
<td>cookie</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>ddi_umem_alloc() allocates page-aligned kernel memory and returns a pointer to the allocated memory. The number of bytes allocated is a multiple of the system page size (roundup of size). The allocated memory can be used in the kernel and can be exported to user space. See devmap(9E) and devmap_umem_setup(9F) for further information.</td>
</tr>
<tr>
<td>flag determines whether the caller can sleep for memory and whether the allocated memory is locked or not. DDI_UMEM_SLEEP allocations may sleep but are guaranteed to succeed. DDI_UMEM_NOSLEEP allocations do not sleep but may fail (return NULL) if memory is currently unavailable. If DDI_UMEM_PAGEABLE is set, pageable memory will be allocated. These pages can be swapped out to secondary memory devices. The initial contents of memory allocated using ddi_umem_alloc() is zero-filled.</td>
<td></td>
</tr>
<tr>
<td>*cookiep is a pointer to the kernel memory cookie that describes the kernel memory being allocated. A typical use of cookiep is in devmap_umem_setup(9F) when the drivers want to export the kernel memory to a user application.</td>
<td></td>
</tr>
<tr>
<td>To free the allocated memory, a driver calls ddi_umem_free() with the cookie obtained from ddi_umem_alloc(). ddi_umem_free() releases the entire buffer.</td>
<td></td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>Non-null</td>
</tr>
<tr>
<td>NULL</td>
<td>Memory cannot be allocated by ddi_umem_alloc() because DDI_UMEM_NOSLEEP is set and the system is out of resources.</td>
</tr>
</tbody>
</table>
ddi_umem_free(9F)

CONTEXT  ddi_umem_alloc() can be called from any context if flag is set to DDI_UMEM_NOSLEEP. If DDI_UMEM_SLEEP is set, ddi_umem_alloc() can be called from user and kernel context only. ddi_umem_free() can be called from any context.

SEE ALSO  dmap(9E), condvar(9F), dmap_umem_setup(9F), kmem_alloc(9F), mutex(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

WARNINGS  Setting the DDI_UMEM_PAGEABLE flag in ddi_umem_alloc() will result in an allocation of pageable memory. Because these pages can be swapped out to secondary memory devices, drivers should use this flag with care. This memory must not be used for the following purposes:

- For synchronization objects such as locks and condition variables. See mutex(9F), semaphore(9F), rwlock(9F), and condvar(9F).
- For driver interrupt routines.

Memory allocated using ddi_umem_alloc() without setting DDI_UMEM_PAGEABLE flag cannot be paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.

Excessive use of kernel memory is likely to effect overall system performance. Over-commitment of kernel memory may cause unpredictable consequences.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing an invalid pointer, will cause the system to corrupt data or panic.

Do not call ddi_umem_alloc() within DDI_SUSPEND and DDI_RESUME operations. Memory acquired at these times is not reliable. In some cases, such a call can cause a system to hang.

NOTES  ddi_umem_alloc(0, flag, cookiep) always returns NULL. ddi_umem_free(NULL) has no effects on system.
The `ddi_umem_iosetup` function is used by drivers to setup I/O requests to application memory which has been locked down using `ddi_umem_lock(9F)`. The `ddi_umem_iosetup(9F)` function returns a pointer to a `buf(9S)` structure corresponding to the memory cookie `cookie`. Drivers can setup multiple buffer structures simultaneously active using the same memory cookie. The `buf(9S)` structures can span all or part of the region represented by the cookie and can overlap each other. The `buf(9S)` structure can be passed to `ddi_dma_buf_bind_handle(9F)` to initiate DMA transfers to or from the locked down memory.

The `off` parameter specifies the offset from the start of the cookie. The `len` parameter represents the length of region to be mapped by the buffer. The `direction` parameter must be set to either `B_READ` or `B_WRITE`, to indicate the action that will be performed by the device. (Note that this direction is in the opposite sense of the VM system's direction of `DDI_UMEMLOCK_READ` and `DDI_UMEMLOCK_WRITE`.) The direction must be compatible with the flags used to create the memory cookie in `ddi_umem_lock(9F)`. For example, if `ddi_umem_lock()` is called with the `flags` parameter set to `DDI_UMEMLOCK_READ`, the `direction` parameter in `ddi_umem_iosetup()` should be set to `B_WRITE`.

The `cookie` parameter is the kernel memory cookie allocated by `ddi_umem_lock(9F)`. The `off` parameter specifies the offset from the start of the cookie. The `len` parameter represents the length of region to be mapped by the buffer. The `direction` parameter must be set to either `B_READ` or `B_WRITE`. The `dev` parameter is the device number. The `blkno` parameter is the block number on device. The `iodone` parameter is the specific biodone(9F) routine. The `sleepflag` parameter determines whether caller can sleep for memory. Possible flags are `DDI_UMEM_SLEEP` to allow sleeping until memory is available, or `DDI_UMEM_NOSLEEP` to return `NULL` immediately if memory is not available.

The `ddi_umem_iosetup(9F)` function is included in `<sys/ddi.h>` and `<sys/sunddi.h>`. The function prototype is:

```c
struct buf *ddi_umem_iosetup(ddi_umem_cookie_t cookie, off_t off, size_t len, int direction, dev_t dev, daddr_t blkno, int (*iodone)(struct buf *), int sleepflag);
```
The `dev` parameter specifies the device to which the buffer is to perform I/O. The `blkno` parameter represents the block number on the device. It will be assigned to the `b_blkno` field of the returned buffer structure. The `iodone` parameter enables the driver to identify a specific `iodone(9F)` routine to be called by the driver when the I/O is complete. The `sleepflag` parameter determines if the caller can sleep for memory. `DDI_UMEM_SLEEP` allocations may sleep but are guaranteed to succeed. `DDI_UMEM_NOSLEEP` allocations do not sleep but may fail (return `NULL`) if memory is currently not available.

After the I/O has completed and the buffer structure is no longer needed, the driver calls `freerbuf(9F)` to free the buffer structure.

### RETURN VALUES

The `ddi_umem_iosetup(9F)` function returns a pointer to the initialized buffer header, or `NULL` if no space is available.

### CONTEXT

The `ddi_umem_iosetup(9F)` function can be called from any context only if `flag` is set to `DDI_UMEM_NOSLEEP`. If `DDI_UMEM_SLEEP` is set, `ddi_umem_iosetup(9F)` can be called from user and kernel context only.

### SEE ALSO

`ddi_umem_lock(9F), ddi_dma_buf_bind_handle(9F), ddi_umem_unlock(9F), freerbuf(9F), physio(9F), buf(9S)`
ddi_umem_lock(9F)

NAME

ddi_umem_lock, ddi_umem_unlock – lock and unlock memory pages

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_umem_lock(caddr_t addr, size_t len, int flags,
                  ddi_umem_cookie_t *cookiep);

void ddi_umem_unlock(ddi_umem_cookie_t cookie);

INTERFACE

LEVEL

ddi_umem_lock

Virtual address of memory object

len

Length of memory object in bytes

flags

Valid flags include:

DDI_UMEMLOCK_READ
    Memory pages are locked to be read from. (Disk write or a network send.)

DDI_UMEMLOCK_WRITE
    Memory pages are locked to be written to. (Disk read or a network receive.)

cookiep

Pointer to a kernel memory cookie.

ddi_umem_unlock

cookie

Kernel memory cookie allocated by ddi_umem_lock().

DESCRIPTION

The ddi_umem_lock(9F) function locks down the physical pages (including I/O pages) that correspond to the current process’ virtual address range [addr, addr + size) and fills in a cookie representing the locked pages. This cookie can be used to create a buf(9S) structure that can be used to perform I/O (see ddi_umem_iosetup(9F) and ddi_dma_buf_bind_handle(9F)), or it can be used with devmap_umem_setup(9F) to export the memory to an application.

The virtual address and length specified must be at a page boundary and the mapping performed in terms of the system page size. See pagesize(1).

The flags argument indicates the intended use of the locked memory. Set flags to DDI_UMEMLOCK_READ if the memory pages will be read (for example, in a disk write or a network send.) Set flags to DDI_UMEMLOCK_WRITE if the memory pages will be written (for example, in a disk read or a network receive). You must choose one (and only one) of these values.

To unlock the locked pages, the drivers call ddi_umem_unlock(9F) with the cookie obtained from ddi_umem_lock(9F).

The process is not allowed to exec(2) or fork(2) while its physical pages are locked down by the device driver.
ddi_umem_lock(9F)

The device driver must ensure that the physical pages have been unlocked after the application has called close(2).

RETURN VALUES

On success, a 0 is returned. Otherwise, one of the following errno values is returned.

EFAULT
User process has no mapping at that address range or does not support locking

EACCES
User process does not have the required permission.

ENOMEM
The system does not have sufficient resources to lock memory.

EAGAIN
Could not allocate system resources required to lock the pages. The ddi_umem_lock() could succeed at a later time.

EINVAL
Requested memory is not aligned on a system page boundary.

CONTEXT
The ddi_umem_lock() function can only be called from user context; ddi_umem_unlock() from user, kernel, and interrupt contexts.

SEE ALSO
ddi_umem_iosetup(9F), ddi_dma_buf_bind_handle(9F), devmap_umem_setup(9F), ddi_umem_alloc(9F)

NOTES
The ddi_umem_lock(9F) function consumes physical memory. The driver is responsible for a speedy unlock to free up the resources.

ddi_umem_unlock() can defer unlocking of the pages to a later time depending on the implementation.
ddi_umem_lock(9F)

NAME
ddi_umem_lock, ddi_umem_unlock – lock and unlock memory pages

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_umem_lock(caddr_t addr, size_t len, int flags,
                   ddi_umem_cookie_t *cookiep);

void ddi_umem_unlock(ddi_umem_cookie_t cookie);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

ddi_umem_lock

addr
Virtual address of memory object

len
Length of memory object in bytes

flags
Valid flags include:

DDI_UMEMLOCK_READ
Memory pages are locked to be read from. (Disk write or a network send.)

DDI_UMEMLOCK_WRITE
Memory pages are locked to be written to. (Disk read or a network receive.)

cookiep
Pointer to a kernel memory cookie.

DESCRIPTION
The ddi_umem_lock(9F) function locks down the physical pages (including I/O
pages) that correspond to the current process’ virtual address range [addr, addr + size)
and fills in a cookie representing the locked pages. This cookie can be used to create a
buf(9S) structure that can be used to perform I/O (see ddi_umem_iostatup(9F) and
ddi_dma_buf_bind_handle(9F)), or it can be used with devmap_umem_setup(9F)
to export the memory to an application.

The virtual address and length specified must be at a page boundary and the mapping
performed in terms of the system page size. See pagesize(1).

The flags argument indicates the intended use of the locked memory. Set flags to
DDI_UMEMLOCK_READ if the memory pages will be read (for example, in a disk write
or a network send.) Set flags to DDI_UMEMLOCK_WRITE if the memory pages will be
written (for example, in a disk read or a network receive). You must choose one (and
only one) of these values.

To unlock the locked pages, the drivers call ddi_umem_unlock(9F) with the cookie
obtained from ddi_umem_lock(9F).

The process is not allowed to exec(2) or fork(2) while its physical pages are locked
down by the device driver.
ddi_umem_unlock(9F)

The device driver must ensure that the physical pages have been unlocked after the application has called close(2).

RETURN VALUES

On success, a 0 is returned. Otherwise, one of the following errno values is returned.

EFAULT
User process has no mapping at that address range or does not support locking

EACCES
User process does not have the required permission.

ENOMEM
The system does not have sufficient resources to lock memory.

EAGAIN
Could not allocate system resources required to lock the pages. The ddi_umem_lock() could succeed at a later time.

EINVAL
Requested memory is not aligned on a system page boundary.

CONTEXT
The ddi_umem_lock() function can only be called from user context; ddi_umem_unlock() from user, kernel, and interrupt contexts.

SEE ALSO
ddi_umem_iosetup(9F), ddi_dma_buf_bind_handle(9F), devmap_umem_setup(9F), ddi_umem_alloc(9F)

NOTES
The ddi_umem_lock(9F) function consumes physical memory. The driver is responsible for a speedy unlock to free up the resources.

ddi_umem_unlock() can defer unlocking of the pages to a later time depending on the implementation.
### NAME

ddi_map_regs, ddi_unmap_regs – map or unmap registers

### SYNOPSIS

```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_map_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp, off_t offset, off_t len);

void ddi_unmap_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp, off_t offset, off_t len);
```

### INTERFACE LEVEL

These interfaces are obsolete. Use ddi_regs_map_setup(9F) instead of ddi_map_regs(). Use ddi_regs_map_free(9F) instead of ddi_unmap_regs().

#### ddi_map_regs()
- **dip**: Pointer to the device’s dev_info structure.
- **rnumber**: Register set number.
- **kaddrp**: Pointer to the base kernel address of the mapped region (set on return).
- **offset**: Offset into register space.
- **len**: Length to be mapped.

#### ddi_unmap_regs()
- **dip**: Pointer to the device’s dev_info structure.
- **rnumber**: Register set number.
- **kaddrp**: Pointer to the base kernel address of the region to be unmapped.
- **offset**: Offset into register space.
- **len**: Length to be unmapped.

### DESCRIPTION

**ddi_map_regs()** maps in the register set given by **rnumber**. The register number determines which register set will be mapped if more than one exists. The base kernel virtual address of the mapped register set is returned in **kaddrp**. **offset** specifies an offset into the register space to start from and **len** indicates the size of the area to be mapped. If **len** is non-zero, it overrides the length given in the register set description. See the discussion of the **reg** property in **sbus(4)** and for more information on register set descriptions. If **len** and **offset** are 0, the entire space is mapped.

**ddi_unmap_regs()** undoes mappings set up by **ddi_map_regs()**. This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to **ddi_unmap_regs()** must correspond to a previous call to **ddi_map_regs()**). Releasing portions of previous mappings is not allowed. **rnumber** determines which register set will be unmapped if more than one exists. The **kaddrp**, **offset** and **len** specify the area to be unmapped. **kaddrp** is a pointer to the address returned from **ddi_map_regs()**; **offset** and **len** should match what **ddi_map_regs()** was called with.
ddi_unmap_regs(9F)

RETURR VALUES

ddi_map_regs() returns:

DDI_SUCCESS on success.

CONTEXT

These functions can be called from user or interrupt context.

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), sbus(4), ddi_regs_map_free(9F), ddi_regs_map_setup(9F)

Writing Device Drivers
delay(9F)

NAME    delay – delay execution for a specified number of clock ticks

SYNOPSIS #include <sys/ddi.h>

void delay(clock_t ticks);

INTERFACE

LEVEL

Architecture independent level 1 (DDI/DKI).

PARAMETERS

ticks The number of clock cycles to delay.

DESCRIPTION delay() provides a mechanism for a driver to delay its execution for a given period of time. Since the speed of the clock varies among systems, drivers should base their time values on microseconds and use drv_usecctohz(9F) to convert microseconds into clock ticks.

delay() uses timeout(9F) to schedule an internal function to be called after the specified amount of time has elapsed. delay() then waits until the function is called. Because timeout() is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use cv_timedwait(9F) rather than delay().

delay() does not busy-wait. If busy-waiting is required, use drv_usecwait(9F).

CONTEXT delay() can be called from user and kernel contexts.

EXAMPLES EXAMPLE 1 delay() Example

Before a driver I/O routine allocates buffers and stores any user data in them, it checks the status of the device (line 12). If the device needs manual intervention (such as, needing to be refilled with paper), a message is displayed on the system console (line 14). The driver waits an allotted time (line 17) before repeating the procedure.

1 struct device { /* layout of physical device registers */
2     int control; /* physical device control word */
3     int status; /* physical device status word */
4     short xmit_char; /* transmit character to device */
5   };
6
7 ...  
9     /* get device registers */
10     register struct device *rp = ...
11
12     while (rp->status & NOPAPER) { /* while printer is out of paper */
13         /* display message and ring bell */
14         /* on system console */
15         cmn_err(CE_WARN, "\007",
16                (getminor(dev) & 0xf));
17         /* wait one minute and try again */
18         delay(60 * drv_usecctohz(1000000));
19     }

Kernel Functions for Drivers 919
delay(9F)

EXAMPLE 1 delay() Example  (Continued)

SEE ALSO biodone(9F), biowait(9F), cv_timedwait(9F), ddi_in_panic(9F),
drv_hztousec(9F), drv_usectohz(9F), drv_usecwait(9F), timeout(9F),
untimout(9F)

Writing Device Drivers
devmap_default_access(9F)

NAME
devmap_default_access – default driver memory access function

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_default_access(devmap_cookie_t dhp, void *pvtp,
                           offset_t off, size_t len, uint_t type, uint_t rw);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
dhp An opaque mapping handle that the system uses to describe the mapping.
pvtp Driver private mapping data.
off User offset within the logical device memory at which the access begins.
len Length (in bytes) of the memory being accessed.
type Type of access operation.
rw Type of access.

description
devmap_default_access() is a function providing the semantics of
devmap_access(9E). The drivers call devmap_default_access() to handle the
mappings that do not support context switching. The drivers should call
devmap_do_ctxmgt(9F) for the mappings that support context management.

devmap_default_access() can either be called from devmap_access(9E) or be
used as the devmap_access(9E) entry point. The arguments dhp, pvtp, off, len, type,
and rw are provided by the devmap_access(9E) entry point and must not be
modified.

RETURN VALUES
0 Successful completion.
Non-zero An error occurred.

CONTEXT
devmap_default_access() must be called from the driver’s devmap_access(9E)
entry point.

EXAMPLES EXAMPLE 1 Using devmap_default_access in devmap_access.
The following shows an example of using devmap_default_access() in the
devmap_access(9E) entry point.
...
#define OFF_DO_CTXMGT  0x40000000
#define OFF_NORMAL  0x40100000
#define CTXSIZE  0x100000
#define NORMAL_SIZE  0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
 */
static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset,


EXAMPLE 1 Using devmap_default_access in devmap_access. (Continued)

```c
size_t length, uint_t type, uint_t rw)
{
    ......
    /*
    * see devmap_contextmgt(9E) for an example
    */
}
/*
* Driver devmap_access(9E) entry point
*/
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;
    /*
    * check if off is within the range that supports
    * context management.
    */
    if ((diff = off - OFF_DO_CTXMG) >= 0 && diff < CTXMGT_SIZE) {
        /*
        * calculates the length for context switching
        */
        if ((len + off) > (OFF_DO_CTXMG + CTXMGT_SIZE))
            return (-1);
        /*
        * perform context switching
        */
        err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
                        rw, xx_context_mgt);
        /*
        * check if off is within the range that does normal
        * memory mapping.
        */
    } else if (((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) {
        if ((len + off) > (OFF_NORMAL + NORMAL_SIZE))
            return (-1);
        err = devmap_default_access(dhp, pvtp, off, len, type, rw);
    } else
        return (-1);
    return (err);
}
```

SEE ALSO  
devmap_access(9E), devmap_do_ctxmgt(9F), devmap_callback_ctl(9S)

Writing Device Drivers
devmap_devmem_setup, devmap_umem_setup – set driver memory mapping parameters

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, uint_t rnumber, offset_t roff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);

int devmap_umem_setup(devmap_cookie_t dhp, dev_info_t *dip, struct
    devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie,
    offset_t koff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

Solaris DDI specific (Solaris DDI).

### INTERFACE LEVEL PARAMETERS

**devmap_devmem_setup()** parameters:

- **dhp**: An opaque mapping handle that the system uses to describe the mapping.
- **dip**: Pointer to the device’s dev_info structure.
- **callbackops**: Pointer to a devmap_callback_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping. The framework will copy the structure to the system private memory.
- **rnumber**: Index number to the register address space set.
- **roff**: Offset into the register address space.
- **len**: Length (in bytes) of the mapping to be mapped.
- **maxprot**: Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:
  - PROT_READ: Read access is allowed.
  - PROT_WRITE: Write access is allowed.
  - PROT_EXEC: Execute access is allowed.
  - PROT_USER: User-level access is allowed (the mapping is being done as a result of a mmap(2) system call).
  - PROT_ALL: All access is allowed.
- **flags**: Must be set to 0.
- **accattrp**: Pointer to a ddi_device_acc_attr(9S) structure. The structure contains the device access attributes to be applied to this range of memory.
**devmap_devmem_setup(9F)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| devmap_devmem_setup() | dhp: An opaque data structure that the system uses to describe the mapping.  
| | dip: Pointer to the device’s dev_info structure.  
| | callbacks: Pointer to a devmap_callback_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.  
| | cookie: A kernel memory cookie (see ddi_umem_alloc(9F)).  
| | koff: Offset into the kernel memory defined by cookie.  
| | len: Length (in bytes) of the mapping to be mapped.  
| | maxprot: Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:  
| | | PROT_READ: Read access is allowed.  
| | | PROT_WRITE: Write access is allowed.  
| | | PROT_EXEC: Execute access is allowed.  
| | | PROT_USER: User-level access is allowed (the mapping is being done as a result of a mmap(2) system call).  
| | | PROT_ALL: All access is allowed.  
| | flags: Must be set to 0.  
| | accattrp: Pointer to a ddi_device_acc_attr(9S) structure. Ignored in the current release. Reserved for future use.  

**DESCRIPTION**

devmap_devmem_setup() and devmap_umem_setup() are used in the devmap(9E) entry point to pass mapping parameters from the driver to the system.

**dhp** is a device mapping handle that the system uses to store all mapping parameters of a physical contiguous memory. The system copies the data pointed to by callbacks to a system private memory. This allows the driver to free the data after returning from either devmap_devmem_setup() or devmap_umem_setup(). The driver is notified of user events on the mappings via the entry points defined by devmap_callback_ctl(9S). The driver is notified of the following user events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
</table>
| Mapping Setup  | User has called mmap(2) to create a mapping to the device memory.  
| Access         | User has accessed an address in the mapping that has no translations.  
| Duplication    | User has duplicated the mapping. Mappings are duplicated when the process calls fork(2).  
| Unmapping      | User has called munmap(2) on the mapping or is exiting, exit(2).  

---

924 man pages section 9: DDI and DKI Kernel Functions • Last Revised 4 Jan 2002
By specifying a valid `callbackops` to the system, device drivers can manage events on a device mapping. For example, the `devmap_access(9E)` entry point allows the drivers to perform context switching by unloading the mappings of other processes and to load the mapping of the calling process. Device drivers may specify `NULL` to `callbackops` which means the drivers do not want to be notified by the system.

The maximum protection allowed for the mapping is specified in `maxprot`, `accattrp` defines the device access attributes. See `ddi_device_acc_attr(9S)` for more details.

`devmap_devmem_setup()` is used for device memory to map in the register set given by `rnumber` and the offset into the register address space given by `roff`. The system uses `rnumber` and `roff` to go up the device tree to get the physical address that corresponds to `roff`. The range to be affected is defined by `len` and `roff`. The range from `roff` to `roff + len` must be a physical contiguous memory and page aligned.

Drivers use `devmap_umem_setup()` for kernel memory to map in the kernel memory described by `cookie` and the offset into the kernel memory space given by `koff`. `cookie` is a kernel memory pointer obtained from `ddi_umem_alloc(9F)`. If `cookie` is `NULL`, `devmap_umem_setup()` returns `-1`. The range to be affected is defined by `len` and `koff`. The range from `koff` to `koff + len` must be within the limits of the kernel memory described by `koff + len` and must be page aligned.

Drivers use `devmap_umem_setup()` to export the kernel memory allocated by `ddi_umem_alloc(9F)` to user space. The system selects a user virtual address that is aligned with the kernel virtual address being mapped to avoid cache incoherence if the mapping is not `MAP_FIXED`.

**RETURN VALUES**

- **0**: Successful completion.
- **-1**: An error occurred.

**CONTEXT**

`devmap_devmem_setup()` and `devmap_umem_setup()` can be called from user, kernel, and interrupt context.

**SEE ALSO**

`exit(2), fork(2), mmap(2), munmap(2), devmap(9E), ddi_umem_alloc(9F), ddi_device_acc_attr(9S), devmap_callback_ctl(9S)`

Writing Device Drivers
**NAME**

devmap_do_ctxmgt(9F)

**SYNOPSIS**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_do_ctxmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
                      size_t len, uint_t type, uint_t rw, int (*devmap_contextmgt)(devmap_cookie_t, void *, offset_t, size_t,
                      uint_t, uint_t);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**PARAMETERS**

- **dhp**: An opaque mapping handle that the system uses to describe the mapping.
- **pvtp**: Driver private mapping data.
- **off**: User offset within the logical device memory at which the access begins.
- **len**: Length (in bytes) of the memory being accessed.
- **devmap_contextmgt**: The address of driver function that the system will call to perform context switching on a mapping. See devmap_contextmgt(9E) for details.
- **type**: Type of access operation. Provided by devmap_access(9E). Should not be modified.
- **rw**: Direction of access. Provided by devmap_access(9E). Should not be modified.

**DESCRIPTION**

Device drivers call devmap_do_ctxmgt() in the devmap_access(9E) entry point to perform device context switching on a mapping. devmap_do_ctxmgt() passes a pointer to a driver supplied callback function, devmap_contextmgt(9E), to the system that will perform the actual device context switching. If devmap_contextmgt(9E) is not a valid driver callback function, the system will fail the memory access operation which will result in a SIGSEGV or SIGBUS signal being delivered to the process.

devmap_do_ctxmgt() performs context switching on the mapping object identified by dhp and pvtp in the range specified by off and len. The arguments dhp, pvtp, type, and rw are provided by the devmap_access(9E) entry point and must not be modified. The range from off to off+len must support context switching.

The system will pass through dhp, pvtp, off, len, type, and rw to devmap_contextmgt(9E) in order to perform the actual device context switching. The return value from devmap_contextmgt(9E) will be returned directly to devmap_do_ctxmgt().

**RETURN VALUES**

- **0**: Successful completion.
- **Non-zero**: An error occurred.
devmap_do_ctxmgt(9F)

CONTEXT

devmap_do_ctxmgt() must be called from the driver's devmap_access(9E) entry point.

EXAMPLES

**EXAMPLE 1** Using devmap_do_ctxmgt in the devmap_access entry point.

The following shows an example of using devmap_do_ctxmgt() in the devmap_access(9E) entry point.

```c
#define OFF_DO_CTXMGT 0x40000000
#define OFF_NORMAL 0x40100000
#define CTXMGT_SIZE 0x100000
#define NORMAL_SIZE 0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
 */
static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset,
    size_t length, uint_t type, uint_t rw)
{
    ........
    /*
    * see devmap_contextmgt(9E) for an example
    */
}

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;
    /*
    * check if off is within the range that supports
    * context management.
    */
    if ((diff = off - OFF_DO_CTXMGT) >= 0 && diff < CTXMGT_SIZE) {
        /*
        * calculates the length for context switching
        */
        if ((len + off) > (OFF_DO_CTXMGT + CTXMGT_SIZE))
            return (-1);
        /*
        * perform context switching
        */
        err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
            rw, xx_context_mgt);
        /*
        * check if off is within the range that does normal
        * memory mapping.
        */
        ...
EXAMPLE 1 Using devmap_do_ctxmgt in the devmap_access entry point.  

    } else if ((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) { 
        if ((len + off) > (OFF_NORMAL + NORMAL_SIZE)) 
            return (-1); 
        err = devmap_default_access(dhp, pvtp, off, len, type, rw); 
    } else 
        return (-1); 

    return (err); 

SEE ALSO  

    devmap_access(9E), devmap_contextmgt(9E), devmap_default_access(9F) 

Writing Device Drivers
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_load(devmap_cookie_t dhp, offset_t off, size_t len, uint_t type, uint_t rw);

int devmap_unload(devmap_cookie_t dhp, offset_t off, size_t len);

Solaris DDI specific (Solaris DDI).

dhp  An opaque mapping handle that the system uses to describe the mapping.

off  User offset within the logical device memory at which the loading or unloading of the address translations begins.

len  Length (in bytes) of the range being affected.

type  Type of access operation.

rw  Direction of access.

devmap_unload() and devmap_load() are used to control the validation of the memory mapping described by dhp in the specified range. devmap_unload() invalidates the mapping translations and will generate calls to the devmap_access(9E) entry point next time the mapping is accessed. The drivers use devmap_load() to validate the mapping translations during memory access.

A typical use of devmap_unload() and devmap_load() is in the driver’s context management callback function, devmap_contextmgt(9E). To manage a device context, a device driver calls devmap_unload() on the context about to be switched out. It switches contexts, and then calls devmap_load() on the context switched in. devmap_unload() can be used to unload the mappings of other processes as well as the mappings of the calling process, but devmap_load() can only be used to load the mappings of the calling process. Attempting to load another process’s mappings with devmap_load() will result in a system panic.

For both routines, the range to be affected is defined by the off and len arguments. Requests affect the entire page containing the off and all pages up to and including the page containing the last byte as indicated by off + len. The arguments type and rw are provided by the system to the calling function (for example, devmap_contextmgt(9E)) and should not be modified.

Supplying a value of 0 for the len argument affects all addresses from the off to the end of the mapping. Supplying a value of 0 for the off argument and a value of 0 for len argument affect all addresses in the mapping.

A non-zero return value from either devmap_unload() or devmap_load() will cause the corresponding operation to fail. The failure may result in a SIGSEGV or SIGBUS signal being delivered to the process.

0  Successful completion.
EXAMPLE 1 Managing a One-Page Device Context

The following shows an example of managing a device context that is one page in length.

```c
struct xx_context cur_ctx;

static int xxdevmap_contextmgt(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len, uint_t type, uint_t rw)
{
    int err;
    devmap_cookie_t cur_dhp;
    struct xx_pvt *p;
    struct xx_pvt *pvtp = (struct xx_pvt *)pvtp;
    /* enable access callbacks for the current mapping */
    if (cur_ctx != NULL && cur_ctx != pvtp->ctx) {
        p = cur_ctx->pvt;
        /*
         * unload the region from off to the end of the mapping.
         */
        cur_dhp = p->dhp;
        if ((err = devmap_unload(cur_dhp, off, len)) != 0)
            return (err);
    }
    /* Switch device context - device dependent*/
    ...
    /* Make handle the new current mapping */
    cur_ctx = pvtp->ctx;
    /*
     * Disable callbacks and complete the access for the
     * mapping that generated this callback.
     */
    return (devmap_load(pvtp->dhp, off, len, type, rw));
}
```

SEE ALSO

`devmap_access(9E), devmap_contextmgt(9E)`

*Writing Device Drivers*
devmap_set_ctx_timeout(9F)

NAME

devmap_set_ctx_timeout – set the timeout value for the context management callback

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

void devmap_set_ctx_timeout(devmap_cookie_t dhp, clock_t ticks);

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dhp An opaque mapping handle that the system uses to describe the mapping.

ticks Number of clock ticks to wait between successive calls to the context

management callback function.

DESCRIPTION

devmap_set_ctx_timeout() specifies the time interval for the system to wait
between successive calls to the driver’s context management callback function,

devmap_contextmgt(9E).

Device drivers typically call devmap_set_ctx_timeout() in the devmap_map(9E)
routine. If the drivers do not call devmap_set_ctx_timeout() to set the timeout
value, the default timeout value of 0 will result in no delay between successive calls to
the driver’s devmap_contextmgt(9E) callback function.

CONTEXT

devmap_set_ctx_timeout() can be called from user or interrupt context.

SEE ALSO

devmap_contextmgt(9E), devmap_map(9E), timeout(9F)
devmap_setup(9F)

NAME
devmap_setup, ddi_devmap_segmap – set up a user mapping to device memory
using the devmap framework

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_setup(dev_t dev, offset_t off, ddi_as_handle_t as,
                 caddr_t *addrp, size_t len, uint_t prot, uint_t maxprot, uint_t
flags, cred_t *cred);

int ddi_devmap_segmap(dev_t dev, off_t off, ddi_as_handle_t as,
                       caddr_t *addrp, off_t len, uint_t prot, uint_t maxprot, uint_t
flags, cred_t *cred);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS

dev     Device whose memory is to be mapped.
off     User offset within the logical device memory at which the mapping begins.

as      An opaque data structure that describes the address space into which the
device memory should be mapped.

addrp   Pointer to the starting address in the address space into which the device
memory should be mapped.

len     Length (in bytes) of the memory to be mapped.

prot    A bit field that specifies the protections. Some possible settings
combinations are:

PROT_READ    Read access is desired.
PROT_WRITE   Write access is desired.
PROT_EXEC    Execute access is desired.
PROT_USER    User-level access is desired (the mapping is being done
             as a result of a mmap(2) system call).
PROT_ALL     All access is desired.

maxprot    Maximum protection flag possible for attempted mapping; the

            PROT_WRITE bit may be masked out if the user opened the special file
read-only.

flags     Flags indicating type of mapping. The following flags can be specified:

MAP_PRIVATE Changes are private.
MAP_SHARED  Changes should be shared.
MAP_FIXED   The user specified an address in *addrp rather than
letting the system choose an address.

cred      Pointer to the user credential structure.
**devmap_setup(9F)**

**DESCRIPTION**

devmap_setup() and ddi_devmap_segmap() allow device drivers to use the devmap framework to set up user mappings to device memory. The devmap framework provides several advantages over the default device mapping framework that is used by ddi_segmap(9F) or ddi_segmap_setup(9F). Device drivers should use the devmap framework, if the driver wants to:

- use an optimal MMU pagesize to minimize address translations,
- conserve kernel resources,
- receive callbacks to manage events on the mapping,
- export kernel memory to applications,
- set up device contexts for the user mapping if the device requires context switching,
- assign device access attributes to the user mapping, or
- change the maximum protection for the mapping.

devmap_setup() must be called in the segmap(9E) entry point to establish the mapping for the application. ddi_devmap_segmap() can be called in, or be used as, the segmap(9E) entry point. The differences between devmap_setup() and ddi_devmap_segmap() are in the data type used for off and len.

When setting up the mapping, devmap_setup() and ddi_devmap_segmap() call the devmap(9E) entry point to validate the range to be mapped. The devmap(9E) entry point also translates the logical offset (as seen by the application) to the corresponding physical offset within the device address space. If the driver does not provide its own devmap(9E) entry point, EINVAL will be returned to the mmap(2) system call.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>Non-zero</td>
<td>An error occurred. The return value of devmap_setup() and ddi_devmap_segmap() should be used directly in the segmap(9E) entry point.</td>
</tr>
</tbody>
</table>

**CONTEXT**

devmap_setup() and ddi_devmap_segmap() can be called from user or kernel context only.

**SEE ALSO**

mmap(2), devmap(9E), segmap(9E), ddi_segmap(9F), ddi_segmap_setup(9F), cb_ops(9S)

**Writing Device Drivers**
# solaris DDI specific (Solaris DDI).

development setup, development setup – set driver memory mapping parameters

## NAME

devmap_devmem_setup, devmap_umem_setup

## SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip, 
                        struct devmap_callback_ctl *callbackops, uint_t rnumber, offset_t roff, size_t len, uint_t maxprot, uint_t flags, 
                        ddi_device_acc_attr_t *accattrp);

int devmap_umem_setup(devmap_cookie_t dhp, dev_info_t *dip, struct devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie, 
                        offset_t koff, size_t len, uint_t maxprot, uint_t flags, 
                        ddi_device_acc_attr_t *accattrp);
```

## INTERFACE LEVEL PARAMETERS

### devmap_devmem_setup() parameters:

- **dhp**: An opaque mapping handle that the system uses to describe the mapping.
- **dip**: Pointer to the device's dev_info structure.
- **callbackops**: Pointer to a devmap_callback_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping. The framework will copy the structure to the system private memory.
- **rnumber**: Index number to the register address space set.
- **roff**: Offset into the register address space.
- **len**: Length (in bytes) of the mapping to be mapped.
- **maxprot**: Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:
  - **PROT_READ**: Read access is allowed.
  - **PROT_WRITE**: Write access is allowed.
  - **PROT_EXEC**: Execute access is allowed.
  - **PROT_USER**: User-level access is allowed (the mapping is being done as a result of a mmap(2) system call).
  - **PROT_ALL**: All access is allowed.
- **flags**: Must be set to 0.
- **accattrp**: Pointer to a ddi_device_acc_attr(9S) structure. The structure contains the device access attributes to be applied to this range of memory.
### devmap_umem_setup() Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dhp</code></td>
<td>An opaque data structure that the system uses to describe the mapping.</td>
</tr>
<tr>
<td><code>dip</code></td>
<td>Pointer to the device’s <code>dev_info</code> structure.</td>
</tr>
<tr>
<td><code>callbackops</code></td>
<td>Pointer to a <code>devmap_callback_ctl(9S)</code> structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.</td>
</tr>
<tr>
<td><code>cookie</code></td>
<td>A kernel memory <code>cookie</code> (see <code>ddi_umem_alloc(9F)</code>).</td>
</tr>
<tr>
<td><code>koff</code></td>
<td>Offset into the kernel memory defined by <code>cookie</code>.</td>
</tr>
<tr>
<td><code>len</code></td>
<td>Length (in bytes) of the mapping to be mapped.</td>
</tr>
<tr>
<td><code>maxprot</code></td>
<td>Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:</td>
</tr>
<tr>
<td></td>
<td>- <code>PROT_READ</code> Read access is allowed.</td>
</tr>
<tr>
<td></td>
<td>- <code>PROT_WRITE</code> Write access is allowed.</td>
</tr>
<tr>
<td></td>
<td>- <code>PROT_EXEC</code> Execute access is allowed.</td>
</tr>
<tr>
<td></td>
<td>- <code>PROT_USER</code> User-level access is allowed (the mapping is being done as a result of a <code>mmap(2)</code> system call).</td>
</tr>
<tr>
<td></td>
<td>- <code>PROT_ALL</code> All access is allowed.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Must be set to 0.</td>
</tr>
<tr>
<td><code>accattrp</code></td>
<td>Pointer to a <code>ddi_device_acc_attr(9S)</code> structure. Ignored in the current release. Reserved for future use.</td>
</tr>
</tbody>
</table>

### devmap_devmem_setup() and devmap_umem_setup() are used in the devmap(9E) entry point to pass mapping parameters from the driver to the system.

- `dhp` is a device mapping handle that the system uses to store all mapping parameters of a physical contiguous memory. The system copies the data pointed to by `callbackops` to a system private memory. This allows the driver to free the data after returning from either `devmap_devmem_setup()` or `devmap_umem_setup()`. The driver is notified of user events on the mappings via the entry points defined by `devmap_callback_ctl(9S)`. The driver is notified of the following user events:

  - **Mapping Setup** User has called `mmap(2)` to create a mapping to the device memory.
  - **Access** User has accessed an address in the mapping that has no translations.
  - **Duplication** User has duplicated the mapping. Mappings are duplicated when the process calls `fork(2)`.
  - **Unmapping** User has called `munmap(2)` on the mapping or is exiting, `exit(2)`. |
devmap_umem_setup(9F)

See devmap_map(9E), devmap_access(9E), devmap_dup(9E), and devmap_unmap(9E) for details on these entry points.

By specifying a valid callback to the system, device drivers can manage events on a device mapping. For example, the devmap_access(9E) entry point allows the drivers to perform context switching by unloading the mappings of other processes and to load the mapping of the calling process. Device drivers may specify NULL to callback which means the drivers do not want to be notified by the system.

The maximum protection allowed for the mapping is specified in maxprot. accattrp defines the device access attributes. See ddi_device_acc_attr(9S) for more details.

devmap_devmem_setup() is used for device memory to map in the register set given by rnumber and the offset into the register address space given by roff. The system uses rnumber and roff to go up the device tree to get the physical address that corresponds to roff. The range to be affected is defined by len and roff. The range from roff to roff + len must be a physical contiguous memory and page aligned.

Drivers use devmap_devmem_setup() for kernel memory to map in the kernel memory described by cookie and the offset into the kernel memory space given by koff. cookie is a kernel memory pointer obtained from ddi_umem_alloc(9F). If cookie is NULL, devmap_devmem_setup() returns -1. The range to be affected is defined by len and koff. The range from koff to koff + len must be within the limits of the kernel memory described by koff + len and must be page aligned.

Drivers use devmap_devmem_setup() to export the kernel memory allocated by ddi_umem_alloc(9F) to user space. The system selects a user virtual address that is aligned with the kernel virtual address being mapped to avoid cache incoherence if the mapping is not MAP_FIXED.

RETURN VALUES
0 Successful completion.
-1 An error occurred.

CONTEXT devmap_devmem_setup() and devmap_umem_setup() can be called from user, kernel, and interrupt context.

SEE ALSO exit(2), fork(2), mmap(2), munmap(2), devmap(9E), ddi umem_alloc(9F), ddi_device_acc_attr(9S), devmap_callback_ctl(9S)

Writing Device Drivers
# Syntax

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int devmap_load(devmap_cookie_t dhp, offset_t off, size_t len, uint_t type, uint_t rw);
int devmap_unload(devmap_cookie_t dhp, offset_t off, size_t len);
```

## Description

`devmap_unload()` and `devmap_load()` are used to control the validation of the memory mapping described by `dhp` in the specified range. `devmap_unload()` invalidates the mapping translations and will generate calls to the `devmap_access(9E)` entry point next time the mapping is accessed. The drivers use `devmap_load()` to validate the mapping translations during memory access.

A typical use of `devmap_unload()` and `devmap_load()` is in the driver's context management callback function, `devmap_contextmgt(9E)`. To manage a device context, a device driver calls `devmap_unload()` on the context about to be switched out. It switches contexts, and then calls `devmap_load()` on the context switched in. `devmap_unload()` can be used to unload the mappings of other processes as well as the mappings of the calling process, but `devmap_load()` can only be used to load the mappings of the calling process. Attempting to load another process's mappings with `devmap_load()` will result in a system panic.

For both routines, the range to be affected is defined by the `off` and `len` arguments. Requests affect the entire page containing the `off` and all pages up to and including the page containing the last byte as indicated by `off + len`. The arguments `type` and `rw` are provided by the system to the calling function (for example, `devmap_contextmgt(9E)`) and should not be modified.

Supplying a value of 0 for the `len` argument affects all addresses from the `off` to the end of the mapping. Supplying a value of 0 for the `off` argument and a value of 0 for `len` argument affect all addresses in the mapping.

A non-zero return value from either `devmap_unload()` or `devmap_load()` will cause the corresponding operation to fail. The failure may result in a `SIGSEGV` or `SIGBUS` signal being delivered to the process.

## Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
</tbody>
</table>
Non-zero An error occurred.

These routines can be called from user or kernel context only.

EXAMPLE 1 Managing a One-Page Device Context

The following shows an example of managing a device context that is one page in length.

```c
struct xx_context cur_ctx;

static int
xxdevmap_contextmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
                     size_t len, uint_t type, uint_t rw)
{
    int err;
    devmap_cookie_t cur_dhp;
    struct xx_pvt *p;
    struct xx_pvt *pvp = (struct xx_pvt *)pvtp;
    /* enable access callbacks for the current mapping */
    if (cur_ctx != NULL && cur_ctx != pvp->ctx) {
        p = cur_ctx->pvt;
        /*
         * unload the region from off to the end of the mapping.
         */
        cur_dhp = p->dhp;
        if ((err = devmap_unload(cur_dhp, off, len)) != 0)
            return (err);
    }
    /* Switch device context - device dependent*/
    ...
    /* Make handle the new current mapping */
    cur_ctx = pvp->ctx;
    /*
     * Disable callbacks and complete the access for the
     * mapping that generated this callback.
     */
    return (devmap_load(pvp->dhp, off, len, type, rw));
}
```

SEE ALSO

devmap_access(9E), devmap_contextmgt(9E)

Writing Device Drivers
disksort(9F)

NAME
disksort – single direction elevator seek sort for buffers

SYNOPSIS
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

void

disksort(struct diskhd *dp, struct buf *bp);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS

dp A pointer to a diskhd structure. A diskhd structure is essentially identical to head of a buffer structure (see buf(9S)). The only defined items of interest for this structure are the av_forw and av_back structure elements which are used to maintain the front and tail pointers of the forward linked I/O request queue.

bp A pointer to a buffer structure. Typically this is the I/O request that the driver receives in its strategy routine (see strategy(9E)). The driver is responsible for initializing the b_resid structure element to a meaningful sort key value prior to calling disksort().

DESCRIPTION
The function disksort() sorts a pointer to a buffer into a single forward linked list headed by the av_forw element of the argument *dp.

It uses a one-way elevator algorithm that sorts buffers into the queue in ascending order based upon a key value held in the argument buffer structure element b_resid.

This value can either be the driver calculated cylinder number for the I/O request described by the buffer argument, or simply the absolute logical block for the I/O request, depending on how fine grained the sort is desired to be or how applicable either quantity is to the device in question.

The head of the linked list is found by use of the av_forw structure element of the argument *dp. The tail of the linked list is found by use of the av_back structure element of the argument *dp. The av_forw element of the *bp argument is used by disksort() to maintain the forward linkage. The value at the head of the list presumably indicates the currently active disk area.

CONTEXT
This function can be called from user or interrupt context.

SEE ALSO
strategy(9E), buf(9S)

Writing Device Drivers

WARNINGS
disksort() does no locking. Therefore, any locking is completely the responsibility of the caller.
#include <sys/ddi.h>

int drv_getparm(unsigned int parm, void *value_p);

The kernel parameter to be obtained. Possible values are:

**LBOLT**
Read the value of lbolt. lbolt is a clock_t that is unconditionally incremented by one at each clock tick. No special treatment is applied when this value overflows the maximum value of the signed integral type clock_t. When this occurs, its value will be negative, and its magnitude will be decreasing until it again passes zero. It can therefore not be relied upon to provide an indication of the amount of time that passes since the last system reboot, nor should it be used to mark an absolute time in the system. Only the difference between two measurements of lbolt is significant. It is used in this way inside the system kernel for timing purposes.

**PPGRP**
Read the process group identification number. This number determines which processes should receive a HANGUP or BREAK signal when detected by a driver.

**UPROCP**
Read the process table token value.

**PPID**
Read process identification number.

**PSID**
Read process session identification number.

**TIME**
Read time in seconds.

**UCRED**
Return a pointer to the caller’s credential structure.

A pointer to the data space in which the value of the parameter is to be copied.
drv_getparm() returns 0 to indicate success, -1 to indicate failure. The value stored in the space pointed to by value_p is the value of the parameter if 0 is returned, or undefined if -1 is returned. -1 is returned if you specify a value other than LBOLT, PPGRP, PPID, PSID, TIME, UCRED, or UPROCP. Always check the return code when using this function.

CONTEXT  drv_getparm() can be called from user context only when using PPGRP, PPID, PSID, UCRED, or UPROCP. It can be called from user or interrupt context when using the LBOLT or TIME argument.

SEE ALSO  ddi_get_lbolt(9F), ddi_get_pid(9F), ddi_get_time(9F), buf(9S)

Writing Device Drivers
**drv_hztousec(9F)**

<table>
<thead>
<tr>
<th>NAME</th>
<th>drv_hztousec – convert clock ticks to microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/types.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>clock_t drv_hztousec(clock_t hertz);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td>hertz The number of clock ticks to convert.</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>drv_hztousec() converts into microseconds the time expressed by hertz, which is in system clock ticks.</td>
</tr>
<tr>
<td></td>
<td>The kernel variable lbolt, whose value should be retrieved by calling ddi_get_lbolt(9F), is the length of time the system has been up since boot and is expressed in clock ticks. Drivers often use the value of lbolt before and after an I/O request to measure the amount of time it took the device to process the request. drv_hztousec() can be used by the driver to convert the reading from clock ticks to a known unit of time.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The number of microseconds equivalent to the hertz parameter. No error value is returned. If the microsecond equivalent to hertz is too large to be represented as a clock_t, then the maximum clock_t value will be returned.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>drv_hztousec() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>ddi_get_lbolt(9F), drv_usectohz(9F), drv_usecwait(9F)</td>
</tr>
</tbody>
</table>

*Writing Device Drivers*
drv_priv (9F)

NAME

drv_priv – determine driver privilege

SYNOPSIS

```c
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/ddi.h>

int drv_priv(cred_t *cr);
```

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

PARAMETERS

cr

Pointer to the user credential structure.

DESCRIPTION

`drv_priv()` provides a general interface to the system privilege policy. It determines whether the credentials supplied by the user credential structure pointed to by `cr` identify a privileged process. This function should only be used when file access modes and special minor device numbers are insufficient to provide protection for the requested driver function. It is intended to replace all calls to `suser()` and any explicit checks for effective user ID = 0 in driver code.

RETURN VALUES

This routine returns 0 if it succeeds, `EPERM` if it fails.

CONTEXT

`drv_priv()` can be called from user or interrupt context.

SEE ALSO

`Writing Device Drivers`
### NAME
`drv_usectohz(9F)`

### SYNOPSIS
```
#include <sys/types.h>
#include <sys/ddi.h>

clock_t `drv_usectohz(clock_t microsecs);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- `microsecs` - The number of microseconds to convert.

### DESCRIPTION
`drv_usectohz()` converts a length of time expressed in microseconds to a number of system clock ticks. The time arguments to `timeout(9F)` and `delay(9F)` are expressed in clock ticks.

`drv_usectohz()` is a portable interface for drivers to make calls to `timeout(9F)` and `delay(9F)` and remain binary compatible should the driver object file be used on a system with a different clock speed (a different number of ticks in a second).

### RETURN VALUES
The value returned is the number of system clock ticks equivalent to the `microsecs` argument. No error value is returned. If the clock tick equivalent to `microsecs` is too large to be represented as a `clock_t`, then the maximum `clock_t` value will be returned.

### CONTEXT
`drv_usectohz()` can be called from user or interrupt context.

### SEE ALSO
`delay(9F), drv_hztosec(9F), timeout(9F)`

*Writing Device Drivers*
**NAME**
drv_usecwait – busy-wait for specified interval

**SYNOPSIS**
```
#include <sys/types.h>
#include <sys/ddi.h>

void drv_usecwait(clock_t microsecs);
```

**INTERFACE LEVEL PARAMETERS**
- `microsecs`: The number of microseconds to busy-wait.

**DESCRIPTION**
drv_usecwait() gives drivers a means of busy-waiting for a specified microsecond count. The amount of time spent busy-waiting may be greater than the microsecond count but will minimally be the number of microseconds specified.

delay(9F) can be used by a driver to delay for a specified number of system ticks, but it has two limitations. First, the granularity of the wait time is limited to one clock tick, which may be more time than is needed for the delay. Second, delay(9F) may only be invoked from user context and hence cannot be used at interrupt time or system initialization.

Often, drivers need to delay for only a few microseconds, waiting for a write to a device register to be picked up by the device. In this case, even in user context, delay(9F) produces too long a wait period.

**CONTEXT**
drv_usecwait() can be called from user or interrupt context.

**SEE ALSO**
delay(9F), timeout(9F), untimeout(9F)

**Writing Device Drivers**

**NOTES**
The driver wastes processor time by making this call since drv_usecwait() does not block but simply busy-waits. The driver should only make calls to drv_usecwait() as needed, and only for as much time as needed. drv_usecwait() does not mask out interrupts.
dupb(9F)

NAME
dupb – duplicate a message block descriptor

SYNOPSIS
#include <sys/stream.h>

mblk_t *dupb(mblk_t *bp);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

DESCRIPTION
dupb() creates a new mblk_t structure (see msgb(9S)) to reference the message block pointed to by bp.

Unlike copyb(9F), dupb() does not copy the information in the dblk_t structure (see datab(9S)), but creates a new mblk_t structure to point to it. The reference count in the dblk_t structure (db_ref) is incremented. The new mblk_t structure contains the same information as the original. Note that b_rptr and b_wptr are copied from the bp.

PARAMETERS
bp Pointer to the message block to be duplicated. mblk_t is an instance of the msgb(9S) structure.

RETURN VALUES
If successful, dupb() returns a pointer to the new message block. A NULL pointer is returned if dupb() cannot allocate a new message block descriptor or if the db_ref field of the data block structure (see datab(9S)) has reached a maximum value (255).

CONTEXT
dupb() can be called from user, kernel, or interrupt context.
EXAMPLE 1 Using dupb()

This srv(9E) (service) routine adds a header to all M_DATA messages before passing them along. dupb is used instead of copyb(9F) because the contents of the header block are not changed.

For each message on the queue, if it is a priority message, pass it along immediately (lines 10–11). Otherwise, if it is anything other than an M_DATA message (line 12), and if it can be sent along (line 13), then do so (line 14). Otherwise, put the message back on the queue and return (lines 16–17). For all M_DATA messages, first check to see if the stream is flow-controlled (line 20). If it is, put the message back on the queue and return (lines 37–38). If it is not, the header block is duplicated (line 21).

dupb() can fail either due to lack of resources or because the message block has already been duplicated 255 times. In order to handle the latter case, the example calls copyb(9F) (line 22). If copyb(9F) fails, it is due to buffer allocation failure. In this case, qbufcall(9F) is used to initiate a callback (lines 30-31) if one is not already pending (lines 26-27).

The callback function, xxxcallback(), clears the recorded qbufcall(9F) callback id and schedules the service procedure (lines 49-50). Note that the close routine, xxxclose(), must cancel any outstanding qbufcall(9F) callback requests (lines 58-59).

If dupb() or copyb(9F) succeed, link the M_DATA message to the new message block (line 34) and pass it along (line 35).

```c
1  xxxsrv(q)
2   queue_t *q;
3 {
4     struct xx *xx = (struct xx *)q->q_ptr;
5     mblk_t *mp;
6     mblk_t *bp;
7     extern mblk_t *hdr;
8     while ((mp = getq(q)) != NULL) {
9       if (mp->b_datap->db_type >= QPCTL) {
10          putnext(q, mp);
11       } else if (mp->b_datap->db_type != M_DATA) {
12          if (canputnext(q)) {
13              if ((bp = dupb(hdr)) == NULL)
14                  bp = copyb(hdr);
15              if (bp == NULL) {
16                  size_t size = msgdsize(mp);
17                  putbq(q, mp);
18                  return;
19              } else {
20                  /* M_DATA */
21                  if (canputnext(q)) {
22                      if ((bp = dupb(hdr)) == NULL)
23                          bp = copyb(hdr);
24                      if (bp == NULL) {
25                          size_t size = msgdsize(mp);
26                          putbq(q, mp);
27                          if (xx->xx_qbufcall_id) {
```
dupb(9F)

**EXAMPLE 1 Using dupb()**  (Continued)

```c
27     /* qbufcall pending */
28     return;
29 
30     xx->xx_qbufcall_id = qbufcall(q, size,
31          BPRI_MED, xxxcallback, (intptr_t)q);
32     return;
33 
34     } else {
35     linkb(bp, mp);
36     putnext(q, bp);
37     } else {
38     putbq(q, mp);
39     return;
40 
41     }
42 }
43 void
44 xxxcallback(q)
45     queue_t *q;
46 {
47     struct xx *xx = (struct xx *)q->q_ptr;
48     xx->xx_qbufcall_id = 0;
49     qenable(q);
50 }
51 
52 xxxclose(q, cflag, crp)
53     queue_t *q;
54     int cflag;
55     cred_t *crp;
56 {
57     struct xx *xx = (struct xx *)q->q_ptr;
58     ...
59     if (xx->xx_qbufcall_id)
60     qunbufcall(q, xx->xx_qbufcall_id);
61     ...
```

**SEE ALSO**  srv(9E), copyb(9F), qbufcall(9F), datab(9S), msgb(9S)

*Writing Device Drivers STREAMS Programming Guide*
dupmsg – duplicate a message

#include <sys/stream.h>

mblk_t *dupmsg(mblk_t *mp);

Architecture independent level 1 (DDI/DKI).

**Parameters**

- **mp** — Pointer to the message.

**Description**

dupmsg() forms a new message by copying the message block descriptors pointed to by mp and linking them. dupb(9F) is called for each message block. The data blocks themselves are not duplicated.

**Return Values**

If successful, dupmsg() returns a pointer to the new message block. Otherwise, it returns a NULL pointer. A return value of NULL indicates either memory depletion or the data block reference count, db_ref (see datab(9S)), has reached a limit (255). See dupb(9F).

**Context**

dupmsg() can be called from user, kernel, or interrupt context.

**Examples**

**Example 1 Using dupmsg()**

See copyb(9F) for an example using dupmsg().

**See Also**

copyb(9F), copymsg(9F), dupb(9F), datab(9S)

Writing Device Drivers

STREAMS Programming Guide
enableok(9F)

NAME    enableok - reschedule a queue for service
SYNOPSIS #include <sys/stream.h>
            #include <sys/ddi.h>
            
            void enableok(queue_t *q);

INTERFACE Architecture independent level 1 (DDI/DKI).
LEVEL
PARAMETERS q     A pointer to the queue to be rescheduled.
PARAMETERS
DESCRIPTION enableok() enables queue q to be rescheduled for service. It reverses the effect of a
previously call to noenable(9F) on q by turning off the QNOENB flag in the queue.
CONTEXT enableok() can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1 Using enableok()

The qrestart() routine uses two STREAMS functions to restart a queue that has
been disabled. The enableok() function turns off the QNOENB flag, allowing the
qenable(9F) to schedule the queue for immediate processing.

1    void
2    qrestart(rdwr_q)
3        register queue_t *rdwr_q;
4    {
5        enableok(rdwr_q);
6        /* re-enable a queue that has been disabled */
7        (void) qenable(rdwr_q);
8    }

SEE ALSO noenable(9F), qenable(9F)

Writing Device Drivers STREAMS Programming Guide
esballoc(9F)

**NAME**
esballoc – allocate a message block using a caller-supplied buffer

**SYNOPSIS**
```
#include <sys/stream.h>

mblk_t *esballoc(uchar *base, size_t size, uint_t pri, frtn_t *fr_rtnp);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base</code></td>
<td>Address of user supplied data buffer.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Number of bytes in data buffer.</td>
</tr>
<tr>
<td><code>pri</code></td>
<td>Priority of allocation request (to be used by allocb(9F) function, called by esballoc( )).</td>
</tr>
<tr>
<td><code>fr_rtnp</code></td>
<td>Free routine data structure.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**
esballoc() creates a STREAMS message and attaches a user-supplied data buffer in place of a STREAMS data buffer. It calls allocb(9F) to get a message and data block header only. The newly allocated message will have both the `b_wptr` and `b_rptr` set to the base of the buffer. As when using allocb(9F), the newly allocated message will have both `b_wptr` and `b_rptr` set to the base of the data buffer. The user-supplied data buffer, pointed to by `base`, is used as the data buffer for the message.

When freeb(9F) is called to free the message, the driver’s message freeing routine (referenced through the `free_rtn` structure) is called, with appropriate arguments, to free the data buffer.

The `free_rtn` structure includes the following members:

```c
void (*free_func)(); /* user's freeing routine */
char *free_arg; /* arguments to free_func() */
```

Instead of requiring a specific number of arguments, the `free_arg` field is defined of type `char *`. This way, the driver can pass a pointer to a structure if more than one argument is needed.

The method by which `free_func` is called is implementation-specific. The module writer must not assume that `free_func` will or will not be called directly from STREAMS utility routines like freeb(9F) which free a message block.

`free_func` must not call another modules put procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

`free_func` must not access any dynamically allocated data structure that might no longer exist when it runs.

**RETURN VALUES**
On success, a pointer to the newly allocated message block is returned. On failure, `NULL` is returned.
esballoc(9F)

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>esballoc() can be called from user or interrupt context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEE ALSO</td>
<td>allocb(9F), freeb(9F), datab(9S), free_rtn(9S)</td>
</tr>
<tr>
<td>WARNINGS</td>
<td>The free_func must be defined in kernel space, should be declared void and accept one argument. It has no user context and must not sleep.</td>
</tr>
</tbody>
</table>
NAME
 esbbcall – call function when buffer is available

SYNOPSIS
#include <sys/stream.h>

bufcall_id_t esbbcall(uint_t pri, void *func void *arg, void arg);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS

pri          Priority of allocation request (to be used by allocb(9F) function, called by esbbcall())

func          Function to be called when buffer becomes available.

arg          Argument to func.

DESCRIPTION
esbbcall(), like bufcall(9F), serves as a timeout(9F) call of indeterminate length. If
esballoc(9F) is unable to allocate a message and data block header to go with its
externally supplied data buffer, esbbcall() can be used to schedule the routine func,
to be called with the argument arg when a buffer becomes available. func may be a
routine that calls esballoc(9F) or it may be another kernel function.

RETURN VALUES
On success, a bufcall ID is returned. On failure, 0 is returned. The value returned
from a successful call should be saved for possible future use with unbufcall() should it become necessary to cancel the esbbcall() request (as at driver close
time).

CONTEXT
esbbcall() can be called from user or interrupt context.

SEE ALSO
allocb(9F), bufcall(9F), esballoc(9F), timeout(9F), datab(9S), unbufcall(9F)

Writing Device Drivers STREAMS Programming Guide
flushband(9F)

NAME | flushband – flush messages for a specified priority band
SYNOPSIS | #include <sys/stream.h>
            
            void flushband(queue_t *q, unsigned char pri, int flag);

INTERFACE LEVEL
PARAMETERS | Architecture independent level 1 (DDI/DKI).
            
            q Pointer to the queue.
            
            pri Priority of messages to be flushed.
            
            flag Valid flag values are:
            
            FLUSHDATA Flush only data messages (types M_DATA, M_DELAY, M_PROTO,
                        and M_PCPROTO).
            
            FLUSHALL Flush all messages.

DESCRIPTION | flushband() flushes messages associated with the priority band specified by pri. If
            pri is 0, only normal and high priority messages are flushed. Otherwise, messages are
            flushed from the band pri according to the value of flag.

CONTEXT | flushband() can be called from user or interrupt context.

SEE ALSO | flushq(9F)

Writing Device Drivers STREAMS Programming Guide
NAME
flushq – remove messages from a queue

SYNOPSIS
#include <sys/stream.h>

void flushq(queue_t *q, int flag);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
q Pointer to the queue to be flushed.

flag Valid flag values are:

FLUSHDATA Flush only data messages (types M_DATA M_DELAY M_PROTO and
M_PCPROTO).

FLUSHALL Flush all messages.

DESCRIPTION
flushq() frees messages and their associated data structures by calling
freemsg(9F). If the queue’s count falls below the low water mark and the queue was
blocking an upstream service procedure, the nearest upstream service procedure is
enabled.

CONTEXT
flushq() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using flushq()

This example depicts the canonical flushing code for STREAMS modules. The module
has a write service procedure and potentially has messages on the queue. If it receives
an M_FLUSH message, and if the FLUSRH bit is on in the first byte of the message (line
10), then the read queue is flushed (line 11). If the FLUSHW bit is on (line 12), then the
write queue is flushed (line 13). Then the message is passed along to the next entity in
the stream (line 14). See the example for qreply(9F) for the canonical flushing code
for drivers.

1 /*
2 * Module write-side put procedure.
3 */
4 xxxwput(q, mp)
5 queue_t *q;
6 mblk_t *mp;
7 {
8 switch(mp->b_datap->db_type) {
9 case M_FLUSH:
10 if (*mp->b_rptr & FLUSRH)
11 flushq(RD(q), FLUSHALL);
12 if (*mp->b_rptr & FLUSHW)
13 flushq(q, FLUSHALL);
14 putnext(q, mp);
15 break;
16 }
17 }

SEE ALSO
flushband(9F), freemsg(9F), putq(9F), qreply(9F)
freeb(9F)

NAME
freeb – free a message block

SYNOPSIS
#include <sys/stream.h>

void freeb(mblk_t *bp);

PARAMETERS
bp Pointer to the message block to be deallocated. mblk_t is an instance of
the msgb(9S) structure.

INTERFACE LEVEL DESCRIPTION
Architecture independent level 1 (DDI/DKI).

freeb() deallocates a message block. If the reference count of the db_ref member of
the datab(9S) structure is greater than 1, freeb() decrements the count. If db_ref
equals 1, it deallocates the message block and the corresponding data block and
buffer.

If the data buffer to be freed was allocated with the esballoc(9F), the buffer may be a
non-STREAMS resource. In that case, the driver must be notified that the attached
data buffer needs to be freed, and run its own freeing routine. To make this process
independent of the driver used in the stream, freeb() finds the free_rtn(9S)
structure associated with the buffer. The free_rtn structure contains a pointer to the
driver-dependent routine, which releases the buffer. Once this is accomplished,
freeb() releases the STREAMS resources associated with the buffer.

CONTEXT freeb() can be called from user or interrupt context.

EXAMPLES EXAMPLE 1 Using freeb()

See copyb(9F) for an example of using freeb().

SEE ALSO allocb(9F), copyb(9F), dupb(9F), esballoc(9F), free_rtn(9S)

Writing Device Drivers

STREAMS Programming Guide
freemsg(9F)

| NAME | freemsg – free all message blocks in a message |
| SYNOPSIS | #include <sys/stream.h>  
| | void freemsg(mblk_t *mp); |
| INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI). |
| PARAMETERS |  
| | mp Pointer to the message blocks to be deallocated. mblk_t is an instance of the msgb(9S) structure. If mp is NULL, freemsg() immediately returns. |
| DESCRIPTION | freemsg() calls freeb(9F) to free all message and data blocks associated with the message pointed to by mp. |
| CONTEXT | freemsg() can be called from user or interrupt context. |
| EXAMPLES | EXAMPLE 1 Using freemsg()  
| | See copymsg(9F). |
| SEE ALSO | copymsg(9F), freeb(9F), msgb(9S)  
| | Writing Device Drivers  
| | STREAMS Programming Guide |
| NOTES | The behavior of freemsg() when passed a NULL pointer is Solaris-specific. |
get_pktiopb, free_pktiopb – allocate/free a SCSI packet in the iopb map

#include <sys/scsi/scsi.h>

struct scsi_pkt *get_pktiopb(struct scsi_address *ap, caddr_t *datap, int cdblen, int statuslen, int datalen, int readflag, int (*callback);

void free_pktiopb(struct scsi_pkt *pkt, caddr_t datap, int datalen);

These interfaces are obsolete. Use scsi_alloc_consistent_buf(9F) instead of get_pktiopb(). Use scsi_free_consistent_buf(9F) instead of free_pktiopb().

PARAMETERS

ap Pointer to the target’s scsi_address structure.
datap Pointer to the address of the packet, set by this function.
cdblen Number of bytes required for the SCSI command descriptor block (CDB).
statuslen Number of bytes required for the SCSI status area.
datalen Number of bytes required for the data area of the SCSI command.
readflag If non-zero, data will be transferred from the SCSI target.
callback Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC
pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION

get_pktiopb() allocates a scsi_pkt structure that has a small data area allocated. It is used by some SCSI commands such as REQUEST_SENSE, which involve a small amount of data and require cache-consistent memory for proper operation. It uses ddi_iopb_alloc(9F) for allocating the data area and scsi_resalloc(9F) to allocate the packet and DMA resources.

callback indicates what get_pktiopb() should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC Wait indefinitely for resources.
Other Values callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

free_pktiopb() is used for freeing the packet and its associated resources.

RETURN VALUES

get_pktiopb() returns a pointer to the newly allocated scsi_pkt or a NULL pointer.
free_pktiopb(9F)

**CONTEXT**

If `callback` is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block.

`free_pktiopb()` can be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), ddi_iopb_alloc(9F), scsi_alloc_consistent_buf(9F), scsi_free_consistent_buf(9F), scsi_pktalloc(9F), scsi_resalloc(9F), scsi_pkt(9S)

_Writing Device Drivers_

**NOTES**

The `get_pktiopb()` and `free_pktiopb()` functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, `scsi_alloc_consistent_buf(9F)` and `scsi_free_consistent_buf(9F)`.

`get_pktiopb()` uses scarce resources. For this reason and its obsolescence (see above), its use is discouraged.
### NAME
freerbuf – free a raw buffer header

### SYNOPSIS
```c
#include <sys/buf.h>
#include <sys/ddi.h>

void freerbuf(struct buf *bp);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bp</code></td>
<td>Pointer to a previously allocated buffer header structure.</td>
</tr>
</tbody>
</table>

### DESCRIPTION
freerbuf() frees a raw buffer header previously allocated by getrbuf(9F). This function does not sleep and so may be called from an interrupt routine.

### CONTEXT
freerbuf() can be called from user or interrupt context.

### SEE ALSO
getrbuf(9F), kmem_alloc(9F), kmem_free(9F), kmem_zalloc(9F)
freeze(st, unfreezestr – freeze, thaw the state of a stream

NAME

SYNOPSIS

```
#include <sys/stream.h>
#include <sys/ddi.h>

void freezestr(queue_t *q);
void unfreezestr(queue_t *q);
```

INTERFACE

LEVEL

PARAMETERS

```
q       Pointer to the message queue to freeze/unfreeze.
```

DESCRIPTION

`freezestr()` freezes the state of the entire stream containing the queue pair `q`. A frozen stream blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the stream, and blocks any thread currently within the stream if it attempts to put messages onto or take messages off of any queue within the stream (with the sole exception of the caller). Threads blocked by this mechanism remain so until the stream is thawed by a call to `unfreezestr()`.

Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as `getq(9F)`, `putq(9F)`, `putbq(9F)`, etc.)

CONTEXT

These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.

SEE ALSO

*Writing Device Drivers*

*STREAMS Programming Guide*

NOTES

The `freezestr()` and `unfreezestr()` functions can have a serious impact on system performance. Their use should be very limited. In most cases, there is no need to use `freezestr()` and there are usually better ways to accomplish what you need to do than by freezing the stream.

Calling `freezestr()` to freeze a stream that is already frozen by the caller will result in a single-party deadlock.

The caller of `unfreezestr()` must be the thread who called `freezestr()`.

STREAMS utility functions such as `getq(9F)`, `putq(9F)`, `putbq(9F)`, and so forth, should not be called by the caller of `freezestr()` while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.
geterror(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>geterror – return I/O error</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/types.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>#include &lt;sys/buf.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>#include &lt;sys/ddi.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int geterror(struct buf *bp);</code></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td><code>bp</code> Pointer to a buf(9S) structure.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td><code>geterror()</code> returns the error number from the error field of the buffer header structure.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>An error number indicating the error condition of the I/O request is returned. If the I/O request completes successfully, 0 is returned.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td><code>geterror()</code> can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>buf(9S)</td>
</tr>
</tbody>
</table>

*Writing Device Drivers*
NAME  
gethrtime – get high resolution time

SYNOPSIS  
#include <sys/time.h>

hrtime_t gethrtime(void);

DESCRIPTION  
The gethrtime() function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of adjtime(2) or settimeofday(3C). The hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.

RETURN VALUES  
gethrtime() always returns the current high-resolution real time. There are no error conditions.

CONTEXT  
There are no restrictions on the context from which gethrtime() can be called.

SEE ALSO  
proc(1), gettimeofday(3C), settimeofday(3C), attributes(5)

NOTES  
Although the units of hi-res time are always the same (nanoseconds), the actual resolution is hardware dependent. Hi-res time is guaranteed to be monotonic (it does not go backward, it does not periodically wrap) and linear (it does not occasionally speed up or slow down for adjustment, as the time of day can), but not necessarily unique: two sufficiently proximate calls might return the same value.

The time base used for this function is the same as that for gettimeofday(3C). Values returned by both of these functions can be interleaved for comparison purposes.
getmajor – get major device number

SYNOPSIS
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

major_t getmajor(dev_t dev);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
dev  Device number.

DESCRIPTION
getmajor() extracts the major number from a device number.

RETURN VALUES
The major number.

CONTEXT
getmajor() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using getmajor()

The following example shows both the getmajor() and getminor(9F) functions used in a debug cmn_err(9F) statement to return the major and minor numbers for the device supported by the driver.

dev_t dev;

#ifndef DEBUG
    cmn_err(CR_NOTE, "Driver Started. Major# = %d, Minor# = %d", getmajor(dev), getminor(dev));
#endif

SEE ALSO
    cmn_err(9F), getminor(9F), makedevice(9F)
    Writing Device Drivers

WARNINGS
No validity checking is performed. If dev is invalid, an invalid number is returned.
getminor(9F)

NAME
  getminor – get minor device number

SYNOPSIS
  #include <sys/types.h>
  #include <sys/mkdev.h>
  #include <sys/ddi.h>

  minor_t getminor(dev_t dev);

INTERFACE LEVEL
  Architecture independent level 1 (DDI/DKI).

PARAMETERS
  dev     Device number.

DESCRIPTION
  getminor() extracts the minor number from a device number.

RETURN VALUES
  The minor number.

CONTEXT
  getminor() can be called from user or interrupt context.

EXAMPLES
  See the getmajor(9F) manual page for an example of how to use getminor().

SEE ALSO
  getmajor(9F), makedevice(9F)

          Writing Device Drivers

WARNINGS
  No validity checking is performed. If dev is invalid, an invalid number is returned.
get_pktiopb(9F)

NAME
get_pktiopb, free_pktiopb – allocate/free a SCSI packet in the iopb map

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *get_pktiopb(struct scsi_address *ap, caddr_t *datap, int *cdblen, int *statuslen, int *datalen, int *readflag, int (*callback);

void free_pktiopb(struct scsi_pkt *pkt, caddr_t datap, int datalen);

INTERFACE LEVEL
These interfaces are obsolete. Use scsi_alloc_consistent_buf(9F) instead of get_pktiopb(). Use scsi_free_consistent_buf(9F) instead of free_pktiopb().

PARAMETERS
ap Pointer to the target's scsi_address structure.
datap Pointer to the address of the packet, set by this function.
cdblen Number of bytes required for the SCSI command descriptor block (CDB).
statuslen Number of bytes required for the SCSI status area.
datalen Number of bytes required for the data area of the SCSI command.
readflag If non-zero, data will be transferred from the SCSI target.
callback Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC
pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION
get_pktiopb() allocates a scsi_pkt structure that has a small data area allocated. It is used by some SCSI commands such as REQUEST_SENSE, which involve a small amount of data and require cache-consistent memory for proper operation. It uses ddi_iopb_alloc(9F) for allocating the data area and scsi_resalloc(9F) to allocate the packet and DMA resources.

callback indicates what get_pktiopb() should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC Wait indefinitely for resources.
Other Values callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

free_pktiopb() is used for freeing the packet and its associated resources.

RETURN VALUES
get_pktiopb() returns a pointer to the newly allocated scsi_pkt or a NULL pointer.
If *callback* is `SLEEP_FUNC`, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

`free_pktiopb()` can be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), ddi_iopb_alloc(9F), scsi_alloc_consistent_buf(9F), scsi_free_consistent_buf(9F), scsi_pktalloc(9F), scsi_resalloc(9F), scsi_pt(9S)

**Writing Device Drivers**

The `get_pktiopb()` and `free_pktiopb()` functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, `scsi_alloc_consistent_buf(9F)` and `scsi_free_consistent_buf(9F)`.

`get_pktiopb()` uses scarce resources. For this reason and its obsolescence (see above), its use is discouraged.
getq – get the next message from a queue

SYNOPSIS

`#include <sys/stream.h>`

```c
mblk_t *getq(queue_t *q);
```

INTERFACE

Architecture independent level 1 (DDI/DKI).

PARAMETERS

`q`  
Pointer to the queue from which the message is to be retrieved.

DESCRIPTION

`getq()` is used by a service (`srv(9E)`) routine to retrieve its enqueued messages.

A module or driver may include a service routine to process enqueued messages. Once the STREAMS scheduler calls `srv()` it must process all enqueued messages, unless prevented by flow control. `getq()` obtains the next available message from the top of the queue pointed to by `q`. It should be called in a while loop that is exited only when there are no more messages or flow control prevents further processing.

If an attempt was made to write to the queue while it was blocked by flow control, `getq()` back-enables (restarts) the service routine once it falls below the low water mark.

RETURN VALUES

If there is a message to retrieve, `getq()` returns a pointer to it. If no message is queued, `getq()` returns a NULL pointer.

EXAMPLES

`getq()` can be called from user or interrupt context.

SEE ALSO

`srv(9E), bcanput(9F), canput(9F), dupb(9F), putbq(9F), putq(9F), qenable(9F)`

Writing Device Drivers

STREAMS Programming Guide
getrbuf(9F)

NAME | getrbuf – get a raw buffer header

SYNOPSIS

```
#include <sys/buf.h>
#include <sys/kmem.h>
#include <sys/ddi.h>

struct buf *getrbuf(int sleepflag);
```

INTERFACE LEVEL
Architectures independent level 1 (DDI/DKI).

PARAMETERS

sleepflag | Indicates whether driver should sleep for free space.

DESCRIPTION

getrbuf() allocates the space for a buffer header to the caller. It is used in cases where a block driver is performing raw (character interface) I/O and needs to set up a buffer header that is not associated with the buffer cache.

getrbuf() calls kmem_alloc(9F) to perform the memory allocation. kmem_alloc() requires the information included in the sleepflag argument. If sleepflag is set to KM_SLEEP, the driver may sleep until the space is freed up. If sleepflag is set to KM_NOSLEEP, the driver will not sleep. In either case, a pointer to the allocated space is returned or NULL to indicate that no space was available.

RETURN VALUES

getrbuf() returns a pointer to the allocated buffer header, or NULL if no space is available.

CONTEXT

getrbuf() can be called from user or interrupt context. (Drivers must not allow getrbuf() to sleep if called from an interrupt routine.)

SEE ALSO

bioinit(9F), freerbuf(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
gld(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched,
gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>

gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);

int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);

int gld_unregister(gld_mac_info_t *macinfo);

void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);

void gld_sched(gld_mac_info_t *macinfo);

uint_t gld_intr(caddr_t);

INTERFACE
LEVEL
PARAMETERS

Solaris architecture specific (Solaris DDI).

macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a
pointer to it. Some of the GLD-private elements of the structure may be initialized
before gld_mac_alloc() returns; all other elements are initialized to zero. The
device driver must initialize some structure members, as described in
gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by

gld_mac_alloc().

gld_register() is called from the device driver’s attach(9E) routine, and is used
to link the GLD-based device driver with the GLD framework. Before calling
gld_register() the device driver’s attach(9E) routine must first use
gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of
its structure elements. See gld_mac_info(9S) for more information. A successful call
to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver’s private data pointer (using
ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name
of the driver module as it exists in the filesystem.
The driver's attach(9E) routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach(9E) routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver's detach(9E) function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver's detach(9E) routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver's gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's gldm_send() routine in a timely way. (If the driver's gldm_stop() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
call the device driver’s gldm_intr() function, passing that pointer to the
gld_mac_info(9S) structure. However, if the driver uses a high-level interrupt, it
must provide its own high-level interrupt handler, and trigger a soft interrupt from
within that. In this case, gld_intr() may be specified as the soft interrupt handler in
the call to ddi_add_softintr().

RETURN VALUES

- **gld_mac_alloc()** returns a pointer to a new gld_mac_info(9S) structure.
- gld_register() and gld_unregister() return:
  - DDI_SUCCESS on success.
  - DDI_FAILURE on failure.
- gld_intr() returns a value appropriate for an interrupt handler.

SEE ALSO

gld(7D), gld(9E), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E),
ddi_add_intr(9F).

*Writing Device Drivers*
gld_intr(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched,
gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
int gld_unregister(gld_mac_info_t *macinfo);
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);
uint_t gld_intr(caddr_t);

INTERFACE LEVEL PARAMETERS
Solaris architecture specific (Solaris DDI).

macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a
pointer to it. Some of the GLD-private elements of the structure may be initialized
before gld_mac_alloc() returns; all other elements are initialized to zero. The
device driver must initialize some structure members, as described in
gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by
gld_mac_alloc().

gld_register() is called from the device driver's attach(9E) routine, and is used
to link the GLD-based device driver with the GLD framework. Before calling
gld_register() the device driver's attach(9E) routine must first use
gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of
its structure elements. See gld_mac_info(9S) for more information. A successful call
to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver's private data pointer (using
ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name
of the driver module as it exists in the filesystem.
The driver’s attach(9E) routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach(9E) routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver’s detach(9E) function, and if successful, performs the following tasks:

- ensures the device’s interrupts are stopped, calling the driver’s gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver’s detach(9E) routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver’s interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver’s gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver’s gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver’s gldm_send() routine in a timely way. (If the driver’s gldm_stop() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD’s main interrupt handler. Normally it is specified as the interrupt routine in the device driver’s call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
call the device driver’s `gldm_intr()` function, passing that pointer to the `gld_mac_info(9S)` structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, `gld_intr()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

**RETURN VALUES**

- `gld_mac_alloc()` returns a pointer to a new `gld_mac_info(9S)` structure.
- `gld_register()` and `gld_unregister()` return:
  - `DDI_SUCCESS` on success.
  - `DDI_FAILURE` on failure.
- `gld_intr()` returns a value appropriate for an interrupt handler.

**SEE ALSO**

- `gld(7D)`, `gld(9E)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `dlpi(7P)`, `attach(9E)`, `ddi_add_intr(9F)`.

*Writing Device Drivers*
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched, gld_intr – Generic LAN Driver service routines

```
#include <sys/gld.h>

gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
int gld_unregister(gld_mac_info_t *macinfo);
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);
uint_t gld_intr(caddr_t);
```

Solaris architecture specific (Solaris DDI).

**macinfo** Pointer to a gld_mac_info(9S) structure.

**dip** Pointer to dev_info structure.

**name** Device interface name.

**mp** Pointer to a message block containing a received packet.

**DESCRIPTION**

gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a pointer to it. Some of the GLD-private elements of the structure may be initialized before gld_mac_alloc() returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by gld_mac_alloc().

gld_register() is called from the device driver’s attach(9E) routine, and is used to link the GLD-based device driver with the GLD framework. Before calling gld_register() the device driver’s attach(9E) routine must first use gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of its structure elements. See gld_mac_info(9S) for more information. A successful call to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver’s private data pointer (using ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name of the driver module as it exists in the filesystem.
The driver's attach(9E) routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach(9E) routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver's detach(9E) function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver's detach(9E) routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver's gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's gldm_send() routine in a timely way. (If the driver's gldm_stop() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
call the device driver’s gldm_intr() function, passing that pointer to the

gld_mac_info(9S) structure. However, if the driver uses a high-level interrupt, it
must provide its own high-level interrupt handler, and trigger a soft interrupt from
within that. In this case, gld_intr() may be specified as the soft interrupt handler in
the call to ddi_add_softintr().

**RETURN VALUES**

gld_mac_alloc() returns a pointer to a new gld_mac_info(9S) structure.

gld_register() and gld_unregister() return:

- DDI_SUCCESS on success.
- DDI_FAILURE on failure.

gld_intr() returns a value appropriate for an interrupt handler.

**SEE ALSO**

gld(7D), gld(9E), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E),
ddi_add_intr(9F).

*Writing Device Drivers*
gld_mac_alloc(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched, gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>

gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);

void gld_mac_free(gld_mac_info_t *macinfo);

int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);

int gld_unregister(gld_mac_info_t *macinfo);

void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);

void gld_sched(gld_mac_info_t *macinfo);

uint_t gld_intr(caddr_t);

INTERFACE LEVEL PARAMETERS

Solaris architecture specific (Solaris DDI).

macinfo Pointer to a gld_mac_info(9S) structure.

dip Pointer to dev_info structure.

name Device interface name.

mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a pointer to it. Some of the GLD-private elements of the structure may be initialized before gld_mac_alloc() returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by gld_mac_alloc().

gld_register() is called from the device driver's attach(9E) routine, and is used to link the GLD-based device driver with the GLD framework. Before calling gld_register() the device driver's attach(9E) routine must first use gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of its structure elements. See gld_mac_info(9S) for more information. A successful call to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver's private data pointer (using ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name of the driver module as it exists in the filesystem.
The driver's `attach` routine should return `DDI_SUCCESS` if `gld_register()` succeeds. If `gld_register()` returns `DDI_FAILURE`, the `attach` routine should deallocate any resources it allocated before calling `gld_register()` and then also return `DDI_FAILURE`.

`gld_unregister()` is called by the device driver's `detach` function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's `gldm_stop()` routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If `gld_unregister()` returns `DDI_SUCCESS`, the `detach` routine should deallocate any data structures allocated in the `attach` routine, using `gld_mac_free()` to deallocate the `macinfo` structure, and return `DDI_SUCCESS`. If `gld_unregister()` returns `DDI_FAILURE`, the driver's `detach` routine must leave the device operational and return `DDI_FAILURE`.

`gld_recv()` is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS `M_DATA` message containing the raw packet. `gld_recv()` determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a `DL_UNITDATA_IND` message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to `gld_recv()`. In particular, locks that could be taken by a transmit thread may not be held during a call to `gld_recv()`: the interrupt thread that calls `gld_recv()` may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's `gldm_send()` routine. If the `gldm_send()` routine were to try to acquire a mutex being held by the `gldm_intr()` routine at the time it calls `gld_recv()`, this could result in a panic due to recursive mutex entry.

`gld_sched()` is called by the device driver to reschedule stalled outbound packets. Whenever the driver's `gldm_send()` routine has returned `GLD_NORESOURCES`, the driver must later call `gld_sched()` to inform the GLD framework that it should retry the packets that previously could not be sent. `gld_sched()` should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's `gldm_send()` routine in a timely way. (If the driver's `gldm_stop()` routine is called, the driver is absolved from this obligation until it later again returns `GLD_NORESOURCES` from its `gldm_send()` routine; however, extra calls to `gld_sched()` will not cause incorrect operation.)

`gld_intr()` is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to `ddi_add_intr()`. The argument to the interrupt handler (specified as `int_handler_arg` in the call to `ddi_add_intr()`) must be a pointer to the `gld_mac_info` structure. `gld_intr()` will, when appropriate,
call the device driver's `gldm_intr()` function, passing that pointer to the `gld_mac_info(9S)` structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, `gld_intr()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

**RETURN VALUES**

- `gld_mac_alloc()` returns a pointer to a new `gld_mac_info(9S)` structure.
- `gld_register()` and `gld_unregister()` return:
  - `DDI_SUCCESS` on success.
  - `DDI_FAILURE` on failure.
- `gld_intr()` returns a value appropriate for an interrupt handler.

**SEE ALSO**

- `gld(7D)`, `gld(9E)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `dlpi(7P)`, `attach(9E)`, `ddi_add_intr(9F)`.

Writing Device Drivers
gld_recv(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched,
gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
int gld_unregister(gld_mac_info_t *macinfo);
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);
uint_t gld_intr(caddr_t);

INTERFACE
Solaris architecture specific (Solaris DDI).

PARAMETERS
macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a
pointer to it. Some of the GLD-private elements of the structure may be initialized
before gld_mac_alloc() returns; all other elements are initialized to zero. The
device driver must initialize some structure members, as described in
gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by
gld_mac_alloc().

gld_register() is called from the device driver’s attach(9E) routine, and is used
to link the GLD-based device driver with the GLD framework. Before calling
gld_register() the device driver’s attach(9E) routine must first use
gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of
its structure elements. See gld_mac_info(9S) for more information. A successful call
to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver’s private data pointer (using
ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name
of the driver module as it exists in the filesystem.
The driver’s attach(9E) routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach(9E) routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver’s detach(9E) function, and if successful, performs the following tasks:
- ensures the device’s interrupts are stopped, calling the driver’s gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver’s detach(9E) routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver’s interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(); the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver’s gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver’s gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver’s gldm_send() routine in a timely way. (If the driver’s gldm_stop() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD’s main interrupt handler. Normally it is specified as the interrupt routine in the device driver’s call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
call the device driver’s gldm_intr() function, passing that pointer to the gld_mac_info(9S) structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, gld_intr() may be specified as the soft interrupt handler in the call to ddi_add_softintr().

**RETURN VALUES**

- **gld_mac_alloc()** returns a pointer to a new gld_mac_info(9S) structure.
- **gld_register()** and **gld_unregister()** return:
  - **DDI_SUCCESS** on success.
  - **DDI_FAILURE** on failure.
- **gld_intr()** returns a value appropriate for an interrupt handler.

**SEE ALSO**

gld(7D), gld(9E), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E), ddi_add_intr(9F).

*Writing Device Drivers*
gld_register(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched, gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
int gld_unregister(gld_mac_info_t *macinfo);
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);
uint_t gld_intr(caddr_t);

INTERFACE
Solaris architecture specific (Solaris DDI).

LEVEL

PARAMETERS
macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a pointer to it. Some of the GLD-private elements of the structure may be initialized before gld_mac_alloc() returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in gld_mac_info(9S), before passing the mac_info pointer to gld_register().
gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by gld_mac_alloc().
gld_register() is called from the device driver's attach(9E) routine, and is used to link the GLD-based device driver with the GLD framework. Before calling gld_register() the device driver's attach(9E) routine must first use gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of its structure elements. See gld_mac_info(9S) for more information. A successful call to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver's private data pointer (using ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name of the driver module as it exists in the filesystem.
The driver’s attach routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

If gld_unregister() returns DDI_SUCCESS, the detach routine should deallocate any data structures allocated in the attach routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver’s detach routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver’s interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver’s gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver’s gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver’s gldm_send() routine in a timely way. (If the driver’s gldm_send() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD’s main interrupt handler. Normally it is specified as the interrupt routine in the device driver’s call to ddi_add_intr(). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr()) must be a pointer to the gld_mac_info() structure. gld_intr() will, when appropriate,
call the device driver’s `gldm_intr()` function, passing that pointer to the `gld_mac_info(9S)` structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, `gld_intr()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

**RETURN VALUES**

`gld_mac_alloc()` returns a pointer to a new `gld_mac_info(9S)` structure.

`gld_register()` and `gld_unregister()` return:

- `DDI_SUCCESS` on success.
- `DDI_FAILURE` on failure.

`gld_intr()` returns a value appropriate for an interrupt handler.

**SEE ALSO**

`gld(7D), gld(9E), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E),
 ddi_add_intr(9F).`
SYNOPSIS
#include <sys/gld.h>

gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);

void gld_mac_free(gld_mac_info_t *macinfo);

int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);

int gld_unregister(gld_mac_info_t *macinfo);

void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);

void gld_sched(gld_mac_info_t *macinfo);

uint_t gld_intr(caddr_t);

INTERFACE LEVEL PARAMETERS

Solaris architecture specific (Solaris DDI).

macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a
pointer to it. Some of the GLD-private elements of the structure may be initialized
before gld_mac_alloc() returns; all other elements are initialized to zero. The
device driver must initialize some structure members, as described in
gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by
gld_mac_alloc().

gld_register() is called from the device driver’s attach(9E) routine, and is used
to link the GLD-based device driver with the GLD framework. Before calling
gld_register() the device driver’s attach(9E) routine must first use
gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of
its structure elements. See gld_mac_info(9S) for more information. A successful call
to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver’s private data pointer (using
ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name
of the driver module as it exists in the filesystem.
The driver's attach() routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach() routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver's detach() function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach() routine should deallocate any data structures allocated in the attach() routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver's detach() routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver's gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's gldm_send() routine in a timely way. (If the driver's gldm_stop() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
call the device driver’s `gldm_intr()` function, passing that pointer to the `gld_mac_info` structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, `gld_intr()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

**RETURN VALUES**

- `gld_mac_alloc()` returns a pointer to a new `gld_mac_info` structure.
- `gld_register()` and `gld_unregister()` return:
  - `DDI_SUCCESS` on success.
  - `DDI_FAILURE` on failure.
- `gld_intr()` returns a value appropriate for an interrupt handler.

**SEE ALSO**

- `gld(7D)`, `gld(9E)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `dlpi(7P)`, `attach(9E)`, `ddi_add_intr(9F)`.

*Writing Device Drivers*
gld_unregister(9F)

NAME
gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched, gld_intr – Generic LAN Driver service routines

SYNOPSIS
#include <sys/gld.h>

gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);

int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);

int gld_unregister(gld_mac_info_t *macinfo);

void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);

uint_t gld_intr(caddr_t);

INTERFACE
LEVEL
PARAMETERS
Solaris architecture specific (Solaris DDI).

macinfo Pointer to a gld_mac_info(9S) structure.
dip Pointer to dev_info structure.
name Device interface name.
mp Pointer to a message block containing a received packet.

DESCRIPTION
gld_mac_alloc() allocates a new gld_mac_info(9S) structure and returns a pointer to it. Some of the GLD-private elements of the structure may be initialized before gld_mac_alloc() returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in gld_mac_info(9S), before passing the mac_info pointer to gld_register().

gld_mac_free() frees a gld_mac_info(9S) structure previously allocated by gld_mac_alloc().

gld_register() is called from the device driver's attach(9E) routine, and is used to link the GLD-based device driver with the GLD framework. Before calling gld_register() the device driver's attach(9E) routine must first use gld_mac_alloc() to allocate a gld_mac_info(9S) structure, and initialize several of its structure elements. See gld_mac_info(9S) for more information. A successful call to gld_register() performs the following actions:

- links the device-specific driver with the GLD system;
- sets the device-specific driver's private data pointer (using ddi_set_driver_private(9F)) to point to the macinfo structure;
- creates the minor device node.

The device interface name passed to gld_register() must exactly match the name of the driver module as it exists in the filesystem.
The driver's attach(9E) routine should return DDI_SUCCESS if gld_register() succeeds. If gld_register() returns DDI_FAILURE, the attach(9E) routine should deallocate any resources it allocated before calling gld_register() and then also return DDI_FAILURE.

gld_unregister() is called by the device driver's detach(9E) function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's gldm_stop() routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If gld_unregister() returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free() to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister() returns DDI_FAILURE, the driver's detach(9E) routine must leave the device operational and return DDI_FAILURE.

gld_recv() is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread may not be held during a call to gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to recursive mutex entry.

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver's gldm_send() routine has returned GLD_NORESOURCES, the driver must later call gld_sched() to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched() should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's gldm_send() routine in a timely way. (If the driver's gldm_send() routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send() routine; however, extra calls to gld_sched() will not cause incorrect operation.)

gld_intr() is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to ddi_add_intr(9F). The argument to the interrupt handler (specified as int_handler_arg in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr() will, when appropriate,
gld_unregister(9F)

call the device driver’s gldm_intr() function, passing that pointer to the
gld_mac_info(9S) structure. However, if the driver uses a high-level interrupt, it
must provide its own high-level interrupt handler, and trigger a soft interrupt from
within that. In this case, gld_intr() may be specified as the soft interrupt handler in
the call to ddi_add_softintr().

RETURN VALUES

gld_mac_alloc() returns a pointer to a new gld_mac_info(9S) structure.

gld_register() and gld_unregister() return:
DDI_SUCCESS on success.
DDI_FAILURE on failure.

gld_intr() returns a value appropriate for an interrupt handler.

SEE ALSO

gld(7D), gld(9E), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E),
 ddi_add_intr(9F).

Writing Device Drivers
hat_getkpfnum(9F)

NAME
hat_getkpfnum – get page frame number for kernel address

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

pfnt_t hat_getkpfnum(caddr_t addr);

INTERFACE LEVEL
Architecture independent level 2 (DKI only).

PARAMETERS
addr The kernel virtual address for which the page frame number is to be
returned.

DESCRIPTION
hat_getkpfnum() returns the page frame number corresponding to the kernel
virtual address, addr.

addr must be a kernel virtual address which maps to device memory.
ddi_map_regs(9F) can be used to obtain this address. For example,
ddi_map_regs(9F) can be called in the driver’s attach(9E) routine. The resulting
kernel virtual address can be saved by the driver (see ddi_soft_state(9F)) and
used in mmap(9E). The corresponding ddi_unmap_regs(9F) call can be made in the
driver’s detach(9E) routine. Refer to mmap(9E) for more information.

RETURN VALUES
The page frame number corresponding to the valid virtual address addr. Otherwise the
return value is undefined.

CONTEXT
hat_getkpfnum() can be called only from user or kernel context.

SEE ALSO
attach(9E), detach(9E), mmap(9E), ddi_map_regs(9F), ddi_soft_state(9F),
ddi_unmap_regs(9F)

Writing Device Drivers

NOTES
For some devices, mapping device memory in the driver’s attach(9E) routine and
unmapping device memory in the driver’s detach(9E) routine is a sizeable drain on
system resources. This is especially true for devices with a large amount of physical
address space. Refer to mmap(9E) for alternative methods.
id32_alloc(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>id32_alloc, id32_free, id32_lookup – 32-bit driver ID management routines</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/ddi.h>
#include <sys/id32.h>

uint32_t id32_alloc(void *ptr, int flag);
void id32_free(uint32_t token);
void *id32_lookup(uint32_t token);

| INTERFACE LEVEL | Solaris architecture specific (Solaris DDI). |
| PARAMETERS | |
| ptr | any valid 32- or 64-bit pointer |
| flag | determines whether caller can sleep for memory (see kmem_alloc(9F) for a description) |

| DESCRIPTION | These routines were originally developed so that device drivers could manage 64-bit pointers on devices that save space only for 32-bit pointers. |
| Many device drivers need to pass a 32-bit value to the hardware when attempting I/O. Later, when that I/O completes, the only way the driver has to identify the request that generated that I/O is via a "token". When the I/O is initiated, the driver passes this token to the hardware. When the I/O completes the hardware passes back this 32-bit token. |
| Before Solaris supported 64-bit pointers, device drivers just passed a raw 32-bit pointer to the hardware. When pointers grew to be 64 bits this was no longer possible. The id32_*() routines were created to help drivers translate between 64-bit pointers and a 32-bit token. |
| Given a 32- or 64-bit pointer, the routine id32_alloc() allocates a 32-bit token, returning 0 if KM_NOSLEEP was specified and memory could not be allocated. The allocated token is passed back to id32_lookup() to obtain the original 32- or 64-bit pointer. |
| The routine id32_free() is used to free an allocated token. Once id32_free() is called, the supplied token is no longer valid. |
| Note that these routines have some degree of error checking. This is done so that an invalid token passed to id32_lookup() will not be accepted as valid. When id32_lookup() detects an invalid token it returns NULL. Calling routines should check for this return value so that they do not try to dereference a NULL pointer. |

| CONTEXT | These functions can be called from user or interrupt context. The routine id32_alloc() should not be called from interrupt context when the KM_SLEEP flag is passed in. All other routines can be called from interrupt or kernel context. |

| SEE ALSO | kmem_alloc(9F) |
| Writing Device Drivers |
id32_free(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>id32_alloc, id32_free, id32_lookup – 32-bit driver ID management routines</th>
</tr>
</thead>
</table>
| SYNOPSIS       | `#include <sys/ddi.h>`
                 | `#include <sys/id32.h>`
                 | `uint32_t id32_alloc(void *ptr, int flag);`
                 | `void id32_free(uint32_t token);`
                 | `void *id32_lookup(uint32_t token);` |
| INTERFACE      | Solaris architecture specific (Solaris DDI). |
| LEVEL          | LEVEL | PARAMETERS | |
|                |      | `ptr`      | any valid 32- or 64-bit pointer |
|                |      | `flag`     | determines whether caller can sleep for memory (see kmem_alloc(9F) for a description) |
|    DESCRIPTION |      |  |
|                |      | These routines were originally developed so that device drivers could manage 64-bit pointers on devices that save space only for 32-bit pointers. |
|                |      | Many device drivers need to pass a 32-bit value to the hardware when attempting I/O. Later, when that I/O completes, the only way the driver has to identify the request that generated that I/O is via a “token”. When the I/O is initiated, the driver passes this token to the hardware. When the I/O completes the hardware passes back this 32-bit token. |
|                |      | Before Solaris supported 64-bit pointers, device drivers just passed a raw 32-bit pointer to the hardware. When pointers grew to be 64 bits this was no longer possible. The id32_*() routines were created to help drivers translate between 64-bit pointers and a 32-bit token. |
|                |      | Given a 32- or 64-bit pointer, the routine id32_alloc() allocates a 32-bit token, returning 0 if KM_NOSLEEP was specified and memory could not be allocated. The allocated token is passed back to id32_lookup() to obtain the original 32- or 64-bit pointer. |
|                |      | The routine id32_free() is used to free an allocated token. Once id32_free() is called, the supplied token is no longer valid. |
|                |      | Note that these routines have some degree of error checking. This is done so that an invalid token passed to id32_lookup() will not be accepted as valid. When id32_lookup() detects an invalid token it returns NULL. Calling routines should check for this return value so that they do not try to dereference a NULL pointer. |
| CONTEXT        |      | These functions can be called from user or interrupt context. The routine id32_alloc() should not be called from interrupt context when the KM_SLEEP flag is passed in. All other routines can be called from interrupt or kernel context. |
|                |      | SEE ALSO | kmem_alloc(9F) |
|                |      | Writing Device Drivers |
id32_alloc, id32_free, id32_lookup – 32-bit driver ID management routines

#include <sys/ddi.h>
#include <sys/id32.h>

uint32_t id32_alloc(void *ptr, int flag);
void id32_free(uint32_t token);
void *id32_lookup(uint32_t token);

Solaris architecture specific (Solaris DDI).

ptr any valid 32- or 64-bit pointer
flag determines whether caller can sleep for memory (see kmem_alloc(9F) for a description)

These routines were originally developed so that device drivers could manage 64-bit pointers on devices that save space only for 32-bit pointers.

Many device drivers need to pass a 32-bit value to the hardware when attempting I/O. Later, when that I/O completes, the only way the driver has to identify the request that generated that I/O is via a "token". When the I/O is initiated, the driver passes this token to the hardware. When the I/O completes the hardware passes back this 32-bit token.

Before Solaris supported 64-bit pointers, device drivers just passed a raw 32-bit pointer to the hardware. When pointers grew to be 64 bits this was no longer possible. The id32_*() routines were created to help drivers translate between 64-bit pointers and a 32-bit token.

Given a 32- or 64-bit pointer, the routine id32_alloc() allocates a 32-bit token, returning 0 if KM_NOSLEEP was specified and memory could not be allocated. The allocated token is passed back to id32_lookup() to obtain the original 32- or 64-bit pointer.

The routine id32_free() is used to free an allocated token. Once id32_free() is called, the supplied token is no longer valid.

Note that these routines have some degree of error checking. This is done so that an invalid token passed to id32_lookup() will not be accepted as valid. When id32_lookup() detects an invalid token it returns NULL. Calling routines should check for this return value so that they do not try to dereference a NULL pointer.

These functions can be called from user or interrupt context. The routine id32_alloc() should not be called from interrupt context when the KM_SLEEP flag is passed in. All other routines can be called from interrupt or kernel context.

SEE ALSO
kmem_alloc(9F)

Writing Device Drivers
The functions described here are obsolete. For the inb(), inw(), and inl() functions, use, respectively, ddi_get8(9F), ddi_get16(9F), and ddi_get32(9F) instead. For repinsb(), repinsw(), and repinsd(), use, respectively, ddi_rep_get8(9F), ddi_rep_get16(9F), and ddi_rep_get32(9F) instead.

**PARAMETERS**

- **port**
  A valid I/O port address.

- **addr**
  The address of a buffer where the values will be stored.

- **count**
  The number of values to be read from the I/O port.

**DESCRIPTION**

These routines read data of various sizes from the I/O port with the address specified by `port`.

The inb(), inw(), and inl() functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The repinsb(), repinsw(), and repinsd() functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. `count` specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold count values of the requested size.

**RETURN VALUES**

inb(), inw(), and inl() return the value that was read from the I/O port.

**CONTEXT**

These functions may be called from user or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**
eisa(4), isa(4), attributes(5), ddi_get8(9F), ddi_get16(9F), ddi_get32(9F), ddi_rep_get8(9F), ddi_rep_get16(9F), ddi_rep_get32(9F), outb(9F)
inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port

#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned char inb(int port);
unsigned short inw(int port);
unsigned long inl(int port);

void repinsb(int port, unsigned char *addr, int count);
void repinsw(int port, unsigned short *addr, int count);
void repinsd(int port, unsigned long *addr, int count);

These routines read data of various sizes from the I/O port with the address specified by port.

The inb (), inw (), and inl () functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The repinsb (), repinsw (), and repinsd () functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold count values of the requested size.

RETURN VALUES inb (), inw (), and inl () return the value that was read from the I/O port.

CONTEXT These functions may be called from user or interrupt context.

ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO eisa(4), isa(4), attributes(5), ddi_get8(9F), ddi_get16(9F), ddi_get32(9F), ddi_rep_get8(9F), ddi_rep_get16(9F), ddi_rep_get32(9F), outb(9F)
```c
#include <sys/stream.h>

int insq(queue_t *q, mblk_t *emp, mblk_t *nmp);
```

Interface Level

PARAMETERS

- q: Pointer to the queue containing message `emp`.
- emp: Enqueued message before which the new message is to be inserted.
- nmp: Message to be inserted.

DESCRIPTION

`insq()` inserts a message into a queue. The message to be inserted, `nmp`, is placed in `q` immediately before the message `emp`. If `emp` is `NULL`, the new message is placed at the end of the queue. The queue class of the new message is ignored. All flow control parameters are updated. The service procedure is enabled unless `QNOENB` is set.

RETURN VALUES

`insq()` returns `1` on success, and `0` on failure.

CONTEXT

`insq()` can be called from user or interrupt context.

EXAMPLES

This routine illustrates the steps a transport provider may take to place expedited data ahead of normal data on a queue (assume all `M_DATA` messages are converted into `M_PROTO T_DATA_REQ` messages). Normal `T_DATA_REQ` messages are just placed on the end of the queue (line 16). However, expedited `T_EXDATA_REQ` messages are inserted before any normal messages already on the queue (line 25). If there are no normal messages on the queue, `bp` will be `NULL` and we fall out of the for loop (line 21). `insq` acts like `putq(9F)` in this case.

```c
#include
#include

static int xxxwput(queue_t *q, mblk_t *mp) {
    union T_primitives *tp;
    mblk_t *bp;
    union T_primitives *ntp;

    switch (mp->b_datap->db_type) {
    case M_PROTO:
        tp = (union T_primitives *)mp->b_rptr;
        switch (tp->type) {
        case T_DATA_REQ:
            putq(q, mp);
            break;
        case T_EXDATA_REQ:
            /* Insert code here to protect queue and message block */
            for (bp = q->q_first; bp; bp = bp->b_next) {
                if (bp->b_datap->db_type == M_PROTO) {
                    ntp = (union T_primitives *)bp->b_rptr;
                }
            }
        }
    
    case MPROTO:
        switch (mp->type) {
            case T_DATA_REQ:
                putq(q, mp);
                break;
            case T_EXDATA_REQ:
                /* Insert code here to protect queue and message block */
                for (bp = q->q_first; bp; bp = bp->b_next) {
                    if (bp->b_datap->db_type == M_PROTO) {
                        ntp = (union T_primitives *)bp->b_rptr;
                    }
                }
        }
    
    return 1;
}
```
When using `insq()`, you must ensure that the queue and the message block is not modified by another thread at the same time. You can achieve this either by using STREAMS functions or by implementing your own locking.

**SEE ALSO**

`putq(9F), rmvq(9F), msgb(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**WARNINGS**

If `emp` is non-NULL, it must point to a message on `q` or a system panic could result.
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned char inb(int port);
unsigned short inw(int port);
unsigned long inl(int port);
void repinsb(int port, unsigned char *addr, int count);
void repinsw(int port, unsigned short *addr, int count);
void repinsd(int port, unsigned long *addr, int count);

The functions described here are obsolete. For the inb(), inw(), and inl() functions, use, respectively, ddi_get8(9F), ddi_get16(9F), and ddi_get32(9F) instead. For repinsb(), repinsw(), and repinsd(), use, respectively, ddi_rep_get8(9F), ddi_rep_get16(9F), and ddi_rep_get32(9F) instead.

port A valid I/O port address.
addr The address of a buffer where the values will be stored.
count The number of values to be read from the I/O port.

These routines read data of various sizes from the I/O port with the address specified by port.

The inb(), inw(), and inl() functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The repinsb(), repinsw(), and repinsd() functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold count values of the requested size.

inb(), inw(), and inl() return the value that was read from the I/O port.

These functions may be called from user or interrupt context.

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE | ATTRIBUTE VALUE
--- | ---
Architecture | IA
Stability Level | Obsolete

See also eisa(4), isa(4), attributes(5), ddi_get8(9F), ddi_get16(9F), ddi_get32(9F), ddi_rep_get8(9F), ddi_rep_get16(9F), ddi_rep_get32(9F), outb(9F)
Writing Device Drivers
### NAME
IOC_CONVERT_FROM – determine if there is a need to translate M_IOCTL contents.

### SYNOPSIS
```c
#include <sys/stream.h>

uint_t IOC_CONVERT_FROM(struct iocblk *iocp);
```

### INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

### PARAMETERS
- `iocp` A pointer to the M_IOCTL control structure.

### DESCRIPTION
The `IOC_CONVERT_FROM` macro is used to see if the contents of the current M_IOCTL message had its origin in a different C Language Type Model.

### RETURN VALUES
- `IOC_ILP32` This is an LP64 kernel and the M_IOCTL originated in an ILP32 user process.
- `IOC_NONE` The M_IOCTL message uses the same C Language Type Model as this calling module or driver.

### CONTEXT
`IOC_CONVERT_FROM()` can be called from user or interrupt context.

### SEE ALSO
- `ddi_model_convert_from(9F)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
### NAME
kmem_alloc, kmem_zalloc, kmem_free – allocate kernel memory

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/kmem.h>

void *kmem_alloc(size_t size, int flag);
void *kmem_zalloc(size_t size, int flag);
void kmem_free(void* buf, size_t size);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- `size`: Number of bytes to allocate.
- `flag`: Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.
- `buf`: Pointer to allocated memory.

### DESCRIPTION
- **kmem_alloc()**: allocates `size` bytes of kernel memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can hold any C data structure. No greater alignment can be assumed. `flag` determines whether the caller can sleep for memory. `KM_SLEEP` allocations may sleep but are guaranteed to succeed. `KM_NOSLEEP` allocations are guaranteed not to sleep but may fail (return `NULL`) if no memory is currently available. The initial contents of memory allocated using `kmem_alloc()` are random garbage.
- **kmem_zalloc()**: is like `kmem_alloc()` but returns zero-filled memory.
- **kmem_free()**: frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.

### RETURN VALUES
- If successful, `kmem_alloc()` and `kmem_zalloc()` return a pointer to the allocated memory. If `KM_NOSLEEP` is set and memory cannot be allocated without sleeping, `kmem_alloc()` and `kmem_zalloc()` return `NULL`.

### CONTEXT
- `kmem_alloc()` and `kmem_zalloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid `flag`.
- `kmem_free()` can be called from user or interrupt context.

### SEE ALSO
- `copyout(9F)`, `freerbuf(9F)`, `getrbuf(9F)`
- `Writing Device Drivers`

### WARNINGS
Memory allocated using `kmem_alloc()` is not paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.
Excessive use of kernel memory is likely to affect overall system performance. Overcommitment of kernel memory will cause the system to hang or panic.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing a null or invalid pointer, will corrupt the kernel heap and may cause the system to corrupt data or panic.

The initial contents of memory allocated using `kmem_alloc()` are random garbage. This random garbage may include secure kernel data. Therefore, uninitialized kernel memory should be handled carefully. For example, never `copyout(9F)` a potentially uninitialized buffer.

**NOTES**

- `kmem_alloc(0, flag)` always returns `NULL`. `kmem_free(NULL, 0)` is legal.
### NAME
kmem_cache_create, kmem_cache_alloc, kmem_cache_free, kmem_cache_destroy – kernel memory cache allocator operations

### SYNOPSIS
```
#include <sys/types.h>
#include <sys/kmem.h>

kmem_cache_t *kmem_cache_create(char *name, size_t bufsize, size_t align, int (*constructor)(void *, void *, int), void (*destructor)(void *, void *), void (*reclaim)(void *), void *private, void *vmp, int cflags);

void kmem_cache_destroy(kmem_cache_t *cp);

void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);

void kmem_cache_free(kmem_cache_t *cp, void *obj);
[Synopsis for callback functions:]
int (*constructor)(void *buf, void *un, int kmflags);
void (*destructor)(void *buf, void *un);
```

### INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI)

The parameters for the kmem_cache_* functions are as follows:

- **name**
  Descriptive name of a kstat(9S) structure of class kmem_cache. Only alphanumeric characters can be used in `name`.

- **bufsize**
  Size of the objects it manages.

- **align**
  Required object alignment.

- **constructor**
  Pointer to an object constructor function. Parameters are defined below.

- **destructor**
  Pointer to an object destructor function. Parameters are defined below.

- **reclaim**
  Drivers should pass NULL.

- **private**
  Pass-through argument for constructor/destructor.

- **vmp**
  Drivers should pass NULL.

- **cflags**
 Drivers must pass 0.
**kmflag**

Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.

*obj

Pointer to the object allocated by `kmem_cache_alloc()`.

The parameters for the callback constructor function are as follows:

`void *buf`

Pointer to the object to be constructed.

`void *un`

The private parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

`int kmflags`

Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.

The parameters for the callback destructor function are as follows:

`void *buf`

Pointer to the object to be deconstructed.

`void *un`

The private parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

---

**DESCRIPTION**

In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object’s initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of `mutex_destroy()` and `mutex_init()` each time. An object’s embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state. The essential requirement is that the client must free the object (using `kmem_cache_free()`) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.
A driver should call `kmem_cache_create()` at the time of `_init(9E)` or `attach(9E)`, and call the corresponding `kmem_cache_destroy()` at the time of `_fini(9E)` or `detach(9E).

`kmem_cache_create()` creates a cache of objects, each of size `size` bytes, aligned on an `align` boundary. Drivers not requiring a specific alignment can pass 0. `name` identifies the cache for statistics and debugging. `constructor` and `destructor` convert plain memory into objects and back again; `constructor` can fail if it needs to allocate memory but cannot. `private` is a parameter passed to the constructor and destructor callbacks to support parameterized caches (for example, a pointer to an instance of the driver’s soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat(9S)` structure of class `kmem_cache` and name `name`. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. `kmflag` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it is acceptable to wait for memory if none is currently available.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

**CONTEXT**

Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

`kmem_cache_free()` can be called from user, kernel, or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Object Caching

Consider the following data structure:

```c
struct foo {
    kmutex_t foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barlist;
    int foo_refcnt;
};
```
EXAMPLE 1 Object Caching  (Continued)

Assume that a foo structure cannot be freed until there are no outstanding references to it (foo_refcnt == 0) and all of its pending bar events (whatever they are) have completed (foo_barlist == NULL). The life cycle of a dynamically allocated foo would be something like this:

foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;
use foo;
ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more expensive overhead. All of this overhead (that is, everything other than use foo above) can be eliminated by object caching.

int foo_constructor(void *buf, void *arg, int size) {
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}

void foo_destructor(void *buf, void *arg, int size) {
    struct foo *foo = buf;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}

un = ddi_get_softc(foo_softc, instance);
(void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache", 
ddi_get_instance(dip));
foo_cache = kmem_cache_create(buf,
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor,
    NULL, un, 0);

To allocate, use, and free a foo object:

foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
use foo;
kmem_cache_free(foo_cache, foo);
EXAMPLE 1  Object Caching  (Continued)

This makes foo allocation fast, because the allocator will usually do nothing more
than fetch an already-constructed foo from the cache. foo_constructor and
foo_destructor will be invoked only to populate and drain the cache, respectively.

RETURN VALUES

If successful, the constructor function must return a pointer to the allocated object. If
KM_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor
must return NULL.

kmem_cache_create() returns a pointer to the allocated cache. If the name
parameter contains non-alphanumeric characters, kmem_cache_create() returns
NULL.

If successful, kmem_cache_alloc() returns a pointer to the allocated object. If
KM_NOSLEEP is set and memory cannot be allocated without sleeping,
kmem_cache_alloc() returns NULL.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

condvar(9F), kmem_alloc(9F), mutex(9F), kstat(9S)

Writing Device Drivers

The Slab Allocator: An Object-Caching Kernel Memory Allocator, Bonwick, J.; USENIX
NAME
kmem_cache_create, kmem_cache_alloc, kmem_cache_free, kmem_cache_destroy –
kernel memory cache allocator operations

SYNOPSIS
#include <sys/types.h>
#include <sys/kmem.h>

kmem_cache_t *kmem_cache_create(char *name, size_t bufsize, size_t align, int (*constructor)(void *, void *, int), void (*destructor)(void *, void *), void (*reclaim)(void *), void *private, void *vmp, int cflags);

void kmem_cache_destroy(kmem_cache_t *cp);

void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);

void kmem_cache_free(kmem_cache_t *cp, void *obj);

[Synopsis for callback functions:]
int (*constructor)(void *buf, void *un, int kmflags);

void (*destructor)(void *buf, void *un);

PARAMETERS
The parameters for the kmem_cache_* functions are as follows:

name
Descriptive name of a kstat(9S) structure of class kmem_cache. Only
alphanumeric characters can be used in name.

bufsize
Size of the objects it manages.

align
Required object alignment.

constructor
Pointer to an object constructor function. Parameters are defined below.

destructor
Pointer to an object destructor function. Parameters are defined below.

reclaim
Drivers should pass NULL.

private
Pass-through argument for constructor/destructor.

vmp
Drivers should pass NULL.

cflags
Drivers must pass 0.
Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

*obj
  Pointer to the object allocated by kmem_cache_alloc().

The parameters for the callback constructor function are as follows:

void *buf
  Pointer to the object to be constructed.

void *un
  The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

int kmflags
  Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

The parameters for the callback destructor function are as follows:

void *buf
  Pointer to the object to be deconstructed.

void *un
  The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

**DESCRIPTION**

In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object’s initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have mutex_init() applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of mutex_destroy() and mutex_init() each time. An object’s embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state. The essential requirement is that the client must free the object (using kmem_cache_free()) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.
A driver should call `kmem_cache_create()` at the time of `_init`(9E) or `attach`(9E), and call the corresponding `kmem_cache_destroy()` at the time of `_fini`(9E) or `detach`(9E).

`kmem_cache_create()` creates a cache of objects, each of size `size` bytes, aligned on an `align` boundary. Drivers not requiring a specific alignment can pass 0. `name` identifies the cache for statistics and debugging. `constructor` and `destructor` convert plain memory into objects and back again; `constructor` can fail if it needs to allocate memory but cannot. `private` is a parameter passed to the `constructor` and `destructor` callbacks to support parameterized caches (for example, a pointer to an instance of the driver’s soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat`(9S) structure of class `kmem_cache` and name `name`. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. `kmflag` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it is acceptable to wait for memory if none is currently available.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

**CONTEXT**

Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

`kmem_cache_free()` can be called from user, kernel, or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Object Caching

Consider the following data structure:

```c
struct foo {
    kmutex_t foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barlist;
    int foo_refcnt;
};
```

```
EXAMPLE 1 Object Caching (Continued)

Assume that a foo structure cannot be freed until there are no outstanding references to it (foo_refcnt == 0) and all of its pending bar events (whatever they are) have completed (foo_barlist == NULL). The life cycle of a dynamically allocated foo would be something like this:

```c
foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;
use foo;
ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more expensive overhead. All of this overhead (that is, everything other than use foo above) can be eliminated by object caching.

```c
int foo_constructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}

void foo_destructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}
```

```c
un = ddi_get_soft_state(foo_softc, instance);
(void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache",
    ddi_get_instance(dip));
foo_cache = kmem_cache_create(buf,
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor,
    NULL, un, 0);
```

To allocate, use, and free a foo object:

```c
foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
use foo;
kmem_cache_free(foo_cache, foo);
```
EXAMPLE 1 Object Caching (Continued)

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. foo_constructor and foo_destructor will be invoked only to populate and drain the cache, respectively.

RETURN VALUES

If successful, the constructor function must return a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor must return NULL.

kmem_cache_create() returns a pointer to the allocated cache. If the name parameter contains non-alphanumeric characters, kmem_cache_create() returns NULL.

If successful, kmem_cache_alloc() returns a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, kmem_cache_alloc() returns NULL.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

cndvar(9F), kmem_alloc(9F), mutex(9F), kstat(9S)

Writing Device Drivers

The parameters for the `kmem_cache_*` functions are as follows:

- **name**
  - Descriptive name of a `kstat(9S)` structure of class `kmem_cache`. Only alphanumeric characters can be used in `name`.

- **bufsize**
  - Size of the objects it manages.

- **align**
  - Required object alignment.

- **constructor**
  - Pointer to an object constructor function. Parameters are defined below.

- **destructor**
  - Pointer to an object destructor function. Parameters are defined below.

- **reclaim**
  - Drivers should pass `NULL`.

- **private**
  - Pass-through argument for constructor/destructor.

- **vmp**
  - Drivers should pass `NULL`.

- **cflags**
  - Drivers must pass 0.
Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

*obj
Pointer to the object allocated by kmem_cache_alloc().

The parameters for the callback constructor function are as follows:

void *buf
Pointer to the object to be constructed.

void *un
The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

int kmflags
Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

The parameters for the callback destructor function are as follows:

void *buf
Pointer to the object to be deconstructed.

void *un
The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

DESCRIPTION
In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object’s initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have mutex_init() applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of mutex_destroy() and mutex_init() each time. An object’s embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state. The essential requirement is that the client must free the object (using kmem_cache_free()) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.
A driver should call `kmem_cache_create()` at the time of `_init(9E)` or `attach(9E)`, and call the corresponding `kmem_cache_destroy()` at the time of `_fini(9E)` or `detach(9E)`.

`kmem_cache_create()` creates a cache of objects, each of size `size` bytes, aligned on an `align` boundary. Drivers not requiring a specific alignment can pass 0. `name` identifies the cache for statistics and debugging. `constructor` and `destructor` convert plain memory into objects and back again; `constructor` can fail if it needs to allocate memory but cannot. `private` is a parameter passed to the constructor and destructor callbacks to support parameterized caches (for example, a pointer to an instance of the driver’s soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat(9S)` structure of class `kmem_cache` and name `name`. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. `kmflag` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it is acceptable to wait for memory if none is currently available.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

`kmem_cache_free()` can be called from user, kernel, or interrupt context.

**EXAMPLE 1** Object Caching

Consider the following data structure:

```c
struct foo {
    kmutex_t foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barrlist;
    int foo_refcnt;
};
```
EXAMPLE 1 Object Caching  (Continued)

Assume that a foo structure cannot be freed until there are no outstanding references to it (foo_refcnt == 0) and all of its pending bar events (whatever they are) have completed (foo_barlist == NULL). The life cycle of a dynamically allocated foo would be something like this:

```c
foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
CV_Init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;
use foo;
ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more expensive overhead. All of this overhead (that is, everything other than use foo above) can be eliminated by object caching.

```c
int foo_constructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    CV_Init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}
```

```c
void foo_destructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}
```

```c
un = ddi_get_soft_state(foo_softc, instance);
(void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache",
    ddi_get_instance(dip));
foo_cache = kmem_cache_create(buf,
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor,
    NULL, un, 0);
```

To allocate, use, and free a foo object:

```c
foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
use foo;
kmem_cache_free(foo_cache, foo);
```
EXAMPLE 1 Object Caching (Continued)

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. foo_constructor and foo_destructor will be invoked only to populate and drain the cache, respectively.

RETURN VALUES

If successful, the constructor function must return a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor must return NULL.

kmem_cache_create() returns a pointer to the allocated cache. If the name parameter contains non-alphanumeric characters, kmem_cache_create() returns NULL.

If successful, kmem_cache_alloc() returns a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, kmem_cache_alloc() returns NULL.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

condvar(9F), kmem_alloc(9F), mutex(9F), kstat(9S)

Writing Device Drivers

NAME

kmem_cache_create, kmem_cache_alloc, kmem_cache_free, kmem_cache_destroy –
kernel memory cache allocator operations.

SYNOPSIS

#include <sys/types.h>
#include <sys/kmem.h>

kmem_cache_t *kmem_cache_create(char *name, size_t bufsize, size_t align, int (*constructor)(void *, void *, int), void (*destructor)(void *, void *), void (*reclaim)(void *), void *private, void *vmp, int cflags);

void kmem_cache_destroy(kmem_cache_t *cp);

void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);

void kmem_cache_free(kmem_cache_t *cp, void *obj);

[Synopsis for callback functions:]

int (*constructor)(void *buf, void *un, int kmflags);

void (*destructor)(void *buf, void *un);

INTERFACE

Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

The parameters for the kmem_cache_* functions are as follows:

name

Descriptive name of a kstat(9S) structure of class kmem_cache. Only
alphanumeric characters can be used in name.

bufsize

Size of the objects it manages.

align

Required object alignment.

destructor

Pointer to an object destructor function. Parameters are defined below.

reclaim

Drivers should pass NULL.

private

Pass-through argument for constructor/destructor.

vmp

Drivers should pass NULL.

cflags

Drivers must pass 0.
Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

*obj
    Pointer to the object allocated by kmem_cache_alloc().

The parameters for the callback constructor function are as follows:

void *buf
    Pointer to the object to be constructed.

void *un
    The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

int kmflags
    Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

The parameters for the callback destructor function are as follows:

void *buf
    Pointer to the object to be deconstructed.

void *un
    The private parameter from the call to kmem_cache_create(); it is typically a pointer to the soft-state structure.

DESCRIPTION

In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object’s initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have mutex_init() applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of mutex_destroy() and mutex_init() each time. An object’s embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state. The essential requirement is that the client must free the object (using kmem_cache_free()) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.
A driver should call `kmem_cache_create()` at the time of `_init(9E)` or `attach(9E)`, and call the corresponding `kmem_cache_destroy()` at the time of `_fini(9E)` or `detach(9E).

`kmem_cache_create()` creates a cache of objects, each of size `size` bytes, aligned on an `align` boundary. Drivers not requiring a specific alignment can pass 0. `name` identifies the cache for statistics and debugging. `constructor` and `destructor` convert plain memory into objects and back again; `constructor` can fail if it needs to allocate memory but cannot. `private` is a parameter passed to the constructor and destructor callbacks to support parameterized caches (for example, a pointer to an instance of the driver's soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat(9S)` structure of class `kmem_cache` and name `name`. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. `kmflag` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it is acceptable to wait for memory if none is currently available.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

**CONTEXT**

Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

`kmem_cache_free()` can be called from user, kernel, or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Object Caching

Consider the following data structure:

```c
struct foo {
    kmutex_t foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barlist;
    int foo_refcnt;
};
```
EXAMPLE 1 Object Caching  (Continued)

Assume that a foo structure cannot be freed until there are no outstanding references to it (foo_refcnt == 0) and all of its pending bar events (whatever they are) have completed (foo_barlist == NULL). The life cycle of a dynamically allocated foo would be something like this:

    foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
    use foo;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
    kmem_free(foo);

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more expensive overhead. All of this overhead (that is, everything other than use foo above) can be eliminated by object caching.

    int
    foo_constructor(void *buf, void *arg, int size)
    {
        struct foo *foo = buf;
        mutex_init(&foo->foo_lock, ...);
        cv_init(&foo->foo_cv, ...);
        foo->foo_refcnt = 0;
        foo->foo_barlist = NULL;
    }

    void
    foo_destructor(void *buf, void *arg, int size)
    {
        struct foo *foo = buf;
        ASSERT(foo->foo_barlist == NULL);
        ASSERT(foo->foo_refcnt == 0);
        cv_destroy(&foo->foo_cv);
        mutex_destroy(&foo->foo_lock);
    }

    un = ddi_get_soft_state(foo_softc, instance);
    (void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache", 
                   ddi_get_instance(dip));
    foo_cache = kmem_cache_create(buf,
                                   sizeof (struct foo), 0,
                                   foo_constructor, foo_destructor,
                                   NULL, un, 0);

To allocate, use, and free a foo object:

    foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
    use foo;
    kmem_cache_free(foo_cache, foo);
EXAMPLE 1 Object Caching  (Continued)

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. foo_constructor and foo_destructor will be invoked only to populate and drain the cache, respectively.

RETURN VALUES

If successful, the constructor function must return a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor must return NULL.

kmem_cache_create() returns a pointer to the allocated cache. If the name parameter contains non-alphanumeric characters, kmem_cache_create() returns NULL.

If successful, kmem_cache_alloc() returns a pointer to the allocated object. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, kmem_cache_alloc() returns NULL.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

condvar(9F), kmem_alloc(9F), mutex(9F), kstat(9S)

Writing Device Drivers

### kmem_free(9F)

**NAME**  
`kmem_alloc`, `kmem_zalloc`, `kmem_free` – allocate kernel memory

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/kmem.h>

void *kmem_alloc(size_t size, int flag);
void *kmem_zalloc(size_t size, int flag);
void kmem_free(void *buf, size_t size);
```

**INTERFACE LEVEL**  
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**

- `size`  
  Number of bytes to allocate.

- `flag`  
  Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return NULL immediately if memory is not available.

- `buf`  
  Pointer to allocated memory.

**DESCRIPTION**

`kmem_alloc()` allocates `size` bytes of kernel memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can hold any C data structure. No greater alignment can be assumed. `flag` determines whether the caller can sleep for memory. `KM_SLEEP` allocations may sleep but are guaranteed to succeed. `KM_NOSLEEP` allocations are guaranteed not to sleep but may fail (return NULL) if no memory is currently available. The initial contents of memory allocated using `kmem_alloc()` are random garbage.

`kmem_zalloc()` is like `kmem_alloc()` but returns zero-filled memory.

`kmem_free()` frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.

**RETURN VALUES**

If successful, `kmem_alloc()` and `kmem_zalloc()` return a pointer to the allocated memory. If `KM_NOSLEEP` is set and memory cannot be allocated without sleeping, `kmem_alloc()` and `kmem_zalloc()` return NULL.

**CONTEXT**

`kmem_alloc()` and `kmem_zalloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid `flag`. `kmem_free()` can be called from user or interrupt context.

**SEE ALSO**

`copyout(9F)`, `freerbuf(9F)`, `getrbuf(9F)`

*Writing Device Drivers*

**WARNINGS**

Memory allocated using `kmem_alloc()` is not paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.
Excessive use of kernel memory is likely to affect overall system performance. Overcommitment of kernel memory will cause the system to hang or panic.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing a null or invalid pointer, will corrupt the kernel heap and may cause the system to corrupt data or panic.

The initial contents of memory allocated using `kmem_alloc()` are random garbage. This random garbage may include secure kernel data. Therefore, uninitialized kernel memory should be handled carefully. For example, never `copyout(9F)` a potentially uninitialized buffer.

**NOTES**

`kmem_alloc(0, flag)` always returns NULL. `kmem_free(NULL, 0)` is legal.
NAME | kmem_alloc, kmem_zalloc, kmem_free – allocate kernel memory
SYNOPSIS
#include <sys/types.h>
#include <sys/kmem.h>

void *kmem_alloc(size_t size, int flag);
void *kmem_zalloc(size_t size, int flag);
void kmem_free(void *buf, size_t size);

INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI).
PARAMETERS
size | Number of bytes to allocate.
flag | Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.
buf | Pointer to allocated memory.

DESCRIPTION
kmem_alloc() allocates size bytes of kernel memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can hold any C data structure. No greater alignment can be assumed. flag determines whether the caller can sleep for memory. KM_SLEEP allocations may sleep but are guaranteed to succeed. KM_NOSLEEP allocations are guaranteed not to sleep but may fail (return NULL) if no memory is currently available. The initial contents of memory allocated using kmem_alloc() are random garbage.

kmem_zalloc() is like kmem_alloc() but returns zero-filled memory.

kmem_free() frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.

RETURN VALUES
If successful, kmem_alloc() and kmem_zalloc() return a pointer to the allocated memory. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, kmem_alloc() and kmem_zalloc() return NULL.

CONTEXT
kmem_alloc() and kmem_zalloc() can be called from interrupt context only if the KM_NOSLEEP flag is set. They can be called from user context with any valid flag. kmem_free() can be called from user or interrupt context.

SEE ALSO
copyout(9F), freerbuf(9F), getrbuf(9F)

Writing Device Drivers

WARNINGS
Memory allocated using kmem_alloc() is not paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.
Excessive use of kernel memory is likely to affect overall system performance. Overcommitment of kernel memory will cause the system to hang or panic.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing a null or invalid pointer, will corrupt the kernel heap and may cause the system to corrupt data or panic.

The initial contents of memory allocated using `kmem_alloc()` are random garbage. This random garbage may include secure kernel data. Therefore, uninitialized kernel memory should be handled carefully. For example, never `copyout(9F)` a potentially uninitialized buffer.

**NOTES**

`kmem_alloc(0, flag)` always returns NULL. `kmem_free(NULL, 0)` is legal.
kstat_create(9F)

NAME

kstat_create – create and initialize a new kstat

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

kstat_t *kstat_create(char *module, int instance, char *name, char
  *class, uchar_t type, ulong_t ndata, uchar_t ks_flag);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

module The name of the provider’s module (such as "sd", "esp", ...). The
  "core" kernel uses the name "unix".

instance The provider’s instance number, as from ddi_get_instance(9F).
  Modules which do not have a meaningful instance number should
  use 0.

name A pointer to a string that uniquely identifies this structure. Only
  KSTAT_STRLEN – 1 characters are significant.

class The general class that this kstat belongs to. The following classes
  are currently in use: disk, tape, net, controller, vm, kvm,
  hat, streams, kstat, and misc.

type The type of kstat to allocate. Valid types are:

  KSTAT_TYPE_NAMED
  Allows more than one data record per kstat.

  KSTAT_TYPE_INTR
  Interrupt; only one data record per kstat.

  KSTAT_TYPE_IO
  I/O; only one data record per kstat

ndata The number of type-specific data records to allocate.

flag A bit-field of various flags for this kstat. flag is some combination
  of:

  KSTAT_FLAG_VIRTUAL
  Tells kstat_create() not to allocate memory for the kstat
  data section; instead, the driver will set the ks_data field to
  point to the data it wishes to export. This provides a convenient
  way to export existing data structures.

  KSTAT_FLAG_WRITABLE
  Makes the kstat data section writable by root.

  KSTAT_FLAG_PERSISTENT
  Indicates that this kstat is to be persistent over time. For
  persistent kstats, kstat_delete(9F) simply marks the
  kstat as dormant; a subsequent kstat_create() reactivates
  the kstat. This feature is provided so that statistics are not lost
  across driver close/open (such as raw disk I/O on a disk with
kstat_create(9F)

no mounted partitions.) Note: Persistent kstats cannot be virtual, since ks_data points to garbage as soon as the driver goes away.

DESCRIPTION kstat_create() is used in conjunction with kstat_install(9F) to allocate and initialize a kstat(9S) structure. The method is generally as follows:

kstat_create() allocates and performs necessary system initialization of a kstat(9S) structure. kstat_create() allocates memory for the entire kstat (header plus data), initializes all header fields, initializes the data section to all zeroes, assigns a unique kstat ID (KID), and puts the kstat onto the system’s kstat chain. The returned kstat is marked invalid because the provider (caller) has not yet had a chance to initialize the data section.

After a successful call to kstat_create() the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type KSTAT_TYPE_NAMED). Virtual kstats must have the ks_data field set at this time. The provider may also set the ks_update, ks_private, and ks_lock fields if necessary.

Once the kstat is completely initialized, kstat_install(9F) is used to make the kstat accessible to the outside world.

RETURN VALUES If successful, kstat_create() returns a pointer to the allocated kstat. NULL is returned upon failure.

CONTEXT kstat_create() can be called from user or kernel context.

EXAMPLES EXAMPLE 1 Allocating and Initializing a kstat Structure

```c
pkstat_t *ksp;
    ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}
```

SEE ALSO kstat(3KSTAT), ddi_get_instance(9F), kstat_delete(9F), kstat_install(9F), kstat_named_init(9F), kstat(9S), kstat_named(9S)

Writing Device Drivers
kstat_delete(9F)

NAME | kstat_delete – remove a kstat from the system

SYNOPSIS | 
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_delete(kstat_t *ksp);

INTERFACE LEVEL | Solaris DDI specific (Solaris DDI)
PARAMETERS | 
ksp | Pointer to a currently installed kstat(9S) structure.

DESCRIPTION | kstat_delete() removes ksp from the kstat chain and frees all associated system resources.

RETURN VALUES | None.

CONTEXT | kstat_delete() can be called from any context.
SEE ALSO | kstat_create(9F), kstat_install(9F), kstat_named_init(9F), kstat(9S)

Writing Device Drivers

NOTES | When calling kstat_delete(), the driver must not be holding that kstat's ks_lock. Otherwise, it may deadlock with a kstat reader.
NAME
kstat_install – add a fully initialized kstat to the system

SYNOPSIS
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_install(kstat_t *ksp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
ksp Pointer to a fully initialized kstat(9S) structure.

DESCRIPTION
kstat_install() is used in conjunction with kstat_create(9F) to allocate and initialize a kstat(9S) structure.

After a successful call to kstat_create() the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type KSTAT_TYPE_NAMED). Virtual kstats must have the ks_data field set at this time. The provider may also set the ks_update, ks_private, and ks_lock fields if necessary.

Once the kstat is completely initialized, kstat_install is used to make the kstat accessible to the outside world.

RETURN VALUES
None.

CONTEXT
kstat_install() can be called from user or kernel context.

EXAMPLES
EXAMPLE 1 Allocating and Initializing a kstat Structure

The method for allocating and initializing a kstat structure is generally as follows:

kstat_t *ksp;
ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}

SEE ALSO
kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S)

Writing Device Drivers
### kstat_named_init(9F)

**NAME**

kstat_named_init, kstat_named_setstr – initialize a named kstat

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_named_init(kstat_named_t *knp, char *name, uchar_t data_type);

void kstat_named_setstr(kstat_named_t *knp, const char *str);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI)

**PARAMETERS**

- `knp` Pointer to a kstat_named(9S) structure.
- `name` The name of the statistic.
- `data_type` The type of value. This indicates which field of the kstat_named(9S) structure should be used. Valid values are:
  - `KSTAT_DATA_CHAR` The "char" field.
  - `KSTAT_DATA_LONG` The "long" field.
  - `KSTAT_DATA_ULONG` The "unsigned long" field.
  - `KSTAT_DATA_LONGLONG` The "long long" field.
  - `KSTAT_DATA_ULONGLONG` The "unsigned long long" field.
  - `KSTAT_DATA_STRING` Arbitrary length "long string" field.
- `str` Pointer to a NULL-terminated string.

**DESCRIPTION**

`kstat_named_init()` associates a name and a type with a kstat_named(9S) structure.

`kstat_named_setstr()` associates `str` with the named kstat `knp`. It is an error for `knp` to be of type other than `KSTAT_DATA_STRING`. This is the only supported method of changing the value of long strings.

**RETURN VALUES**

None.

**CONTEXT**

`kstat_named_init()` and `kstat_named_setstr()` can be called from user or kernel context.

**SEE ALSO**

kstat_create(9F), kstat_install(9F), kstat(9S), kstat_named(9S)

*Writing Device Drivers*
### NAME
kstat_named_init, kstat_named_setstr – initialize a named kstat

### SYNOPSIS
```c
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_named_init(kstat_named_t *knp, char *name, uchar_t data_type);

void kstat_named_setstr(kstat_named_t *knp, const char *str);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

### PARAMETERS
- **knp**
  - Pointer to a `kstat_named(9S)` structure.
- **name**
  - The name of the statistic.
- **data_type**
  - The type of value. This indicates which field of the `kstat_named(9S)` structure should be used. Valid values are:
    - `KSTAT_DATA_CHAR`
      - The "char" field.
    - `KSTAT_DATA_LONG`
      - The "long" field.
    - `KSTAT_DATA_ULONG`
      - The "unsigned long" field.
    - `KSTAT_DATA_LONGLONG`
      - The "long long" field.
    - `KSTAT_DATA_ULONGLONG`
      - The "unsigned long long" field.
    - `KSTAT_DATA_STRING`
      - Arbitrary length "long string" field.
- **str**
  - Pointer to a NULL-terminated string.

### DESCRIPTION
- **kstat_named_init()** associates a name and a type with a `kstat_named(9S)` structure.
- **kstat_named_setstr()** associates `str` with the named kstat `knp`. It is an error for `knp` to be of type other than `KSTAT_DATA_STRING`. This is the only supported method of changing the value of long strings.

### RETURN VALUES
None.

### CONTEXT
`kstat_named_init()` and `kstat_named_setstr()` can be called from user or kernel context.

### SEE ALSO
- `kstat_create(9F)`, `kstat_install(9F)`, `kstat(9S)`, `kstat_named(9S)`
- *Writing Device Drivers*
kstat_queue(9F)

NAME
kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
kiop Pointer to a kstat_io(9S) structure.

DESCRIPTION
A large number of I/O subsystems have at least two basic “lists” (or queues) of
transactions they manage: one for transactions that have been accepted for processing
but for which processing has yet to begin, and one for transactions which are actively
being processed (but not done). For this reason, two cumulative time statistics are
kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions
between the driver wait queue and run queue.

kstat_waitq_enter()
kstat_waitq_enter() should be called when a request arrives and is placed
into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()
kstat_waitq_exit() should be used when a request is removed from its
pre-service state. (such as just prior to calling the driver’s start routine).

kstat_runq_enter()
kstat_runq_enter() is also called when a request is placed in its service state
(just prior to calling the driver’s start routine, but after kstat_waitq_exit()).

kstat_runq_exit()
kstat_runq_exit() is used when a request is removed from its service state
(just prior to calling biodone(9F)).

kstat_waitq_to_runq()
kstat_waitq_to_runq() transitions a request from the wait queue to the run
queue. This is useful wherever the driver would have normally done a
kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()
kstat_runq_back_to_waitq() transitions a request from the run queue back to
the wait queue. This may be necessary in some cases (write throttling is an
kstat_queue(9F)

```
example).

RETURN VALUES
None.

CONTEXT
kstat_create() can be called from user or kernel context.

WARNINGS
These transitions must be protected by holding the kstat’s ks_lock, and must be
completely accurate (all transitions are recorded). Forgetting a transition may, for
eexample, make an idle disk appear 100% busy.

SEE ALSO
biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F),
kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
```
The `kstat_queue()` family of functions manage these times based on the transitions between the driver wait queue and run queue.

`kstat_waitq_enter()` should be called when a request arrives and is placed into a pre-service state (such as just prior to calling `disksort(9F)`).

`kstat_waitq_exit()` should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's `start` routine).

`kstat_runq_enter()` is also called when a request is placed in its service state (just prior to calling the driver's `start` routine, but after `kstat_waitq_exit()`).

`kstat_runq_exit()` is used when a request is removed from its service state (just prior to calling `biodone(9F)`).

`kstat_waitq_to_runq()` transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a `kstat_waitq_exit()` followed by a call to `kstat_runq_enter()`.

`kstat_runq_back_to_waitq()` transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an
kstat_runq_back_to_waitq(9F)

example).

RETURN VALUES None.

CONTEXT kstat_create() can be called from user or kernel context.

WARNINGS These transitions must be protected by holding the kstat’s ks_lock, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

SEE ALSO biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
NAME

kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE

Solaris DDI specific (Solaris DDI)

PARAMETERS

kiop Pointer to a kstat_io(9S) structure.

DESCRIPTION

A large number of I/O subsystems have at least two basic "lists" (or queues) of
transactions they manage: one for transactions that have been accepted for processing
but for which processing has yet to begin, and one for transactions which are actively
being processed (but not done). For this reason, two cumulative time statistics are
kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions
between the driver wait queue and run queue.

kstat_waitq_enter()

kstat_waitq_enter() should be called when a request arrives and is placed
into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()

kstat_waitq_exit() should be used when a request is removed from its
pre-service state. (such as just prior to calling the driver's start routine).

kstat_runq_enter()

kstat_runq_enter() is also called when a request is placed in its service state
(just prior to calling the driver's start routine, but after kstat_waitq_exit()).

kstat_runq_exit()

kstat_runq_exit() is used when a request is removed from its service state
(just prior to calling biodone(9F)).

kstat_waitq_to_runq()

kstat_waitq_to_runq() transitions a request from the wait queue to the run
queue. This is useful wherever the driver would have normally done a
kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()

kstat_runq_back_to_waitq() transitions a request from the run queue back to
the wait queue. This may be necessary in some cases (write throttling is an
kstat_runq_enter(9F)

example).

RETURN VALUES
None.

CONTEXT
kstat_create() can be called from user or kernel context.

WARNINGS
These transitions must be protected by holding the kstat’s ks_lock, and must be
completely accurate (all transitions are recorded). Forgetting a transition may, for
example, make an idle disk appear 100% busy.

SEE ALSO
biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F),
kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
kstat_runq_exit(9F)

NAME  kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS  
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE
Solaris DDI specific (Solaris DDI)
LEVEL
PARAMETERS
kiop  Pointer to a kstat_io(9S) structure.

DESCRIPTION
A large number of I/O subsystems have at least two basic "lists" (or queues) of
transactions they manage: one for transactions that have been accepted for processing
but for which processing has yet to begin, and one for transactions which are actively
being processed (but not done). For this reason, two cumulative time statistics are
kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions
between the driver wait queue and run queue.

kstat_waitq_enter()
  kstat_waitq_enter() should be called when a request arrives and is placed
  into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()
  kstat_waitq_exit() should be used when a request is removed from its
  pre-service state. (such as just prior to calling the driver’s start
  routine).

kstat_runq_enter()
  kstat_runq_enter() is also called when a request is placed in its service state
  (just prior to calling the driver’s start routine, but after kstat_waitq_exit()).

kstat_runq_exit()
  kstat_runq_exit() is used when a request is removed from its service state
  (just prior to calling biodone(9F)).

kstat_waitq_to_runq()
  kstat_waitq_to_runq() transitions a request from the wait queue to the run
  queue. This is useful wherever the driver would have normally done a
  kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()
  kstat_runq_back_to_waitq() transitions a request from the run queue back to
  the wait queue. This may be necessary in some cases (write throttling is an
RETURN VALUES
None.

CONTEXT
kstat_create() can be called from user or kernel context.

WARNINGS
These transitions must be protected by holding the kstat’s ks_lock, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

SEE ALSO
biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
**NAME**

kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit, kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI)

**PARAMETERS**

kiop Pointer to a kstat_io(9S) structure.

**DESCRIPTION**

A large number of I/O subsystems have at least two basic "lists" (or queues) of transactions they manage: one for transactions that have been accepted for processing but for which processing has yet to begin, and one for transactions which are actively being processed (but not done). For this reason, two cumulative time statistics are kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions between the driver wait queue and run queue.

kstat_waitq_enter()

kstat_waitq_enter() should be called when a request arrives and is placed into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()

kstat_waitq_exit() should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's start routine).

kstat_runq_enter()

kstat_runq_enter() is also called when a request is placed in its service state (just prior to calling the driver's start routine, but after kstat_waitq_exit()).

kstat_runq_exit()

kstat_runq_exit() is used when a request is removed from its service state (just prior to calling biodone(9F)).

kstat_waitq_to_runq()

kstat_waitq_to_runq() transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()

kstat_runq_back_to_waitq() transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an
\textbf{RETURN VALUES} None.

\textbf{CONTEXT} \texttt{kstat\_create()} can be called from user or kernel context.

\textbf{WARNINGS} These transitions must be protected by holding the \texttt{kstat}'s \texttt{ks\_lock}, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100\% busy.

\textbf{SEE ALSO} \texttt{biodone(9F), disksort(9F), kstat\_create(9F), kstat\_delete(9F), kstat\_named\_init(9F), kstat(9S), kstat\_io(9S)}

\textit{Writing Device Drivers}
kstat_waitq_exit(9F)

NAME
kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
kiop Pointer to a kstat_io(9S) structure.

DESCRIPTION
A large number of I/O subsystems have at least two basic "lists" (or queues) of
transactions they manage: one for transactions that have been accepted for processing
but for which processing has yet to begin, and one for transactions which are actively
being processed (but not done). For this reason, two cumulative time statistics are
kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions
between the driver wait queue and run queue.

kstat_waitq_enter()
   kstat_waitq_enter() should be called when a request arrives and is placed
   into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()
   kstat_waitq_exit() should be used when a request is removed from its
   pre-service state. (such as just prior to calling the driver's start routine).

kstat_runq_enter()
   kstat_runq_enter() is also called when a request is placed in its service state
   (just prior to calling the driver's start routine, but after kstat_waitq_exit()).

kstat_runq_exit()
   kstat_runq_exit() is used when a request is removed from its service state
   (just prior to calling biodone(9F)).

kstat_waitq_to_runq()
   kstat_waitq_to_runq() transitions a request from the wait queue to the run
   queue. This is useful wherever the driver would have normally done a
   kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()
   kstat_runq_back_to_waitq() transitions a request from the run queue back to
   the wait queue. This may be necessary in some cases (write throttling is an
RETURN VALUES
None.

CONTEXT
kstat_create() can be called from user or kernel context.

WARNINGS
These transitions must be protected by holding the kstat’s ks_lock, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

SEE ALSO
biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
NAME  
kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
        kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS  
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE

LEVEL  
Solaris DDI specific (Solaris DDI)

PARAMETERS  
kiop  
Pointer to a kstat_io structure.

DESCRIPTION  
A large number of I/O subsystems have at least two basic "lists" (or queues) of
transactions they manage: one for transactions that have been accepted for processing
but for which processing has yet to begin, and one for transactions which are actively
being processed (but not done). For this reason, two cumulative time statistics are
kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions
between the driver wait queue and run queue.

kstat_waitq_enter()  
    kstat_waitq_enter() should be called when a request arrives and is placed
into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit()  
    kstat_waitq_exit() should be used when a request is removed from its
pre-service state. (such as just prior to calling the driver's start routine).

kstat_runq_enter()  
    kstat_runq_enter() is also called when a request is placed in its service state
(just prior to calling the driver's start routine, but after kstat_waitq_exit()).

kstat_runq_exit()  
    kstat_runq_exit() is used when a request is removed from its service state
(just prior to calling biodone(9F)).

kstat_waitq_to_runq()  
    kstat_waitq_to_runq() transitions a request from the wait queue to the run
queue. This is useful wherever the driver would have normally done a
    kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq()  
    kstat_runq_back_to_waitq() transitions a request from the run queue back to
the wait queue. This may be necessary in some cases (write throttling is an
<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT</td>
<td><code>kstat_create()</code> can be called from user or kernel context.</td>
</tr>
<tr>
<td>WARNINGS</td>
<td>These transitions must be protected by holding the <code>kstat</code>'s <code>ks_lock</code>, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S), kstat_io(9S)</code></td>
</tr>
</tbody>
</table>

---

Writing Device Drivers
**NAME**
linkb – concatenate two message blocks

**SYNOPSIS**
```c
#include <sys/stream.h>

void linkb(mblk_t *mp1, mblk_t *mp2);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**DESCRIPTION**
linkb() creates a new message by adding mp2 to the tail of mp1. The continuation pointer, b_cont, of mp1 is set to point to mp2.

The following figure describes how the `linkb(m1, m2);` function concatenates two message blocks, mp1 and mp2:

![Diagram showing the concatenation of two message blocks](image)

**PARAMETERS**
- `mp1` The message to which `mp2` is to be added. mblk_t is an instance of the msgb(9S) structure.
- `mp2` The message to be added.

**CONTEXT**
linkb() can be called from user or interrupt context.

**EXAMPLES**
See dupb(9F) for an example that uses linkb().

**SEE ALSO**
dupb(9F), unlinkb(9F), msgb(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*
makecom(9F)

NAME
makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 – make a packet
for SCSI commands

SYNOPSIS
#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int
flag, int cmd, int addr, int cnt);

void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp,
int flag, int cmd, int cnt, int fixbit);

void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int
flag, int cmd, int addr, int cnt);

void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int
flag, int cmd, int addr, int cnt);

These interfaces are obsolete. scsi_setup_cdb(9F) should be used instead.

INTERFACE
PARAMETERS

pkt Pointer to an allocated scsi_pkt(9S) structure.

devp Pointer to the target’s scsi_device(9S) structure.

flag Flags for the pkt_flags member.

cmd First byte of a group 0 or 1 or 5 SCSI CDB.

addr Pointer to the location of the data.

cnt Data transfer length in units defined by the SCSI device type. For
sequential devices cnt is the number of bytes. For block devices,
cnt is the number of blocks.

fixbit Fixed bit in sequential access device commands.

DESCRIPTION
makecom functions initialize a packet with the specified command descriptor block,
devp and transport flags. The pkt_address, pkt_flags, and the command
descriptor block pointed to by pkt_cdbp are initialized using the remaining
arguments. Target drivers may use makecom_g0() for Group 0 commands (except for
sequential access devices), or makecom_g0_s() for Group 0 commands for sequential
access devices, or makecom_g1() for Group 1 commands, or makecom_g5() for
Group 5 commands. fixbit is used by sequential access devices for accessing fixed
block sizes and sets the the tag portion of the SCSI CDB.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using makecom Functions
if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1,
               (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE,
               (int) blkno, nblk);
}
EXAMPLE 1 Using makecom Functions  (Continued)

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), scsi_setup_cdb(9F), scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES

The makecom_g0(), makecom_g0_s(), makecom_g1(), and makecom_g5() functions are obsolete and will be discontinued in a future release. These functions have been replaced by the scsi_setup_cdb() function. See scsi_setup_cdb(9F).
makecom_g0(9F)

**NAME**
makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 - make a packet for SCSI commands

**SYNOPSIS**
```
#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int addr, int cnt);
void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt, int fixbit);
void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt);
void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int addr, int cnt);
```

These interfaces are obsolete. `scsi_setup_cdb(9F)` should be used instead.

**INTERFACE LEVEL PARAMETERS**
- **pkt** Pointer to an allocated `scsi_pkt(9S)` structure.
- **devp** Pointer to the target's `scsi_device(9S)` structure.
- **flag** Flags for the `pkt_flags` member.
- **cmd** First byte of a group 0 or 1 or 5 SCSI CDB.
- **addr** Pointer to the location of the data.
- **cnt** Data transfer length in units defined by the SCSI device type. For sequential devices `cnt` is the number of bytes. For block devices, `cnt` is the number of blocks.
- **fixbit** Fixed bit in sequential access device commands.

**DESCRIPTION**
makecom functions initialize a packet with the specified command descriptor block, `devp` and transport flags. The `pkt_address`, `pkt_flags`, and the command descriptor block pointed to by `pkt_cdbp` are initialized using the remaining arguments. Target drivers may use `makecom_g0()` for Group 0 commands (except for sequential access devices), or `makecom_g0_s()` for Group 0 commands for sequential access devices, or `makecom_g1()` for Group 1 commands, or `makecom_g5()` for Group 5 commands. `fixbit` is used by sequential access devices for accessing fixed block sizes and sets the the tag portion of the SCSI CDB.

**CONTEXT**
These functions can be called from user or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Using makecom Functions
```
if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1, 
              (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE, 
              (int) blkno, nblk);
}
```
EXAMPLE 1 Using makecom Functions (Continued)

ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), scsi_setup_cdb(9F), scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES
The makecom_g0(), makecom_g0_s(), makecom_g1(), and makecom_g5() functions are obsolete and will be discontinued in a future release. These functions have been replaced by the scsi_setup_cdb() function. See scsi_setup_cdb(9F).
makecom_g0_s(9F)

NAME
makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 – make a packet for SCSI commands

SYNOPSIS
#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int addr, int cnt);

void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt, int fixbit);

void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt);

void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt);

INTERFACE
These interfaces are obsolete. scsi_setup_cdb(9F) should be used instead.

LEVEL

PARAMETERS

 pkt Pointer to an allocated scsi_pkt(9S) structure.

 devp Pointer to the target’s scsi_device(9S) structure.

 flag Flags for the pkt_flags member.

 cmd First byte of a group 0 or 1 or 5 SCSI CDB.

 addr Pointer to the location of the data.

 cnt Data transfer length in units defined by the SCSI device type. For sequential devices cnt is the number of bytes. For block devices, cnt is the number of blocks.

 fixbit Fixed bit in sequential access device commands.

DESCRIPTION
makecom functions initialize a packet with the specified command descriptor block, devp and transport flags. The pkt_address, pkt_flags, and the command descriptor block pointed to by pkt_cdbp are initialized using the remaining arguments. Target drivers may use makecom_g0() for Group 0 commands (except for sequential access devices), or makecom_g0_s() for Group 0 commands for sequential access devices, or makecom_g1() for Group 1 commands, or makecom_g5() for Group 5 commands. fixbit is used by sequential access devices for accessing fixed block sizes and sets the tag portion of the SCSI CDB.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using makecom Functions

if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1,
               (int) blkno, nbblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE,
                (int) blkno, nbblk);
}
EXAMPLE 1 Using makecom Functions (Continued)

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), scsi_setup_cdb(9F), scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES

The makecom_g0(), makecom_g0_s(), makecom_g1(), and makecom_g5() functions are obsolete and will be discontinued in a future release. These functions have been replaced by the scsi_setup_cdb() function. See scsi_setup_cdb(9F).
makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 - make a packet for SCSI commands

#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int addr, int cnt);
void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt, int fixbit);
void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int cnt);
void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd, int addr, int cnt);

These interfaces are obsolete. scsi_setup_cdb(9F) should be used instead.

 pkt Pointer to an allocated scsi_pkt(9S) structure.

devp Pointer to the target's scsi_device(9S) structure.

flag Flags for the pkt_flags member.

cmd First byte of a group 0 or 1 or 5 SCSI CDB.

addr Pointer to the location of the data.

cnt Data transfer length in units defined by the SCSI device type. For sequential devices cnt is the number of bytes. For block devices, cnt is the number of blocks.

fixbit Fixed bit in sequential access device commands.

DESCRIPTION

makecom functions initialize a packet with the specified command descriptor block, devp and transport flags. The pkt_address, pkt_flags, and the command descriptor block pointed to by pkt_cdbp are initialized using the remaining arguments. Target drivers may use makecom_g0() for Group 0 commands (except for sequential access devices), or makecom_g0_s() for Group 0 commands for sequential access devices, or makecom_g1() for Group 1 commands, or makecom_g5() for Group 5 commands. fixbit is used by sequential access devices for accessing fixed block sizes and sets the tag portion of the SCSI CDB.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1 Using makecom Functions

if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1, (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE, (int) blkno, nblk);
}
EXAMPLE 1 Using makecom Functions  (Continued)

ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), scsi_setup_cdb(9F), scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES
The makecom_g0(), makecom_g0_s(), makecom_g1(), and makecom_g5() functions are obsolete and will be discontinued in a future release. These functions have been replaced by the scsi_setup_cdb() function. See scsi_setup_cdb(9F).
`makecom_g0()`, `makecom_g0_s()`, `makecom_g1()`, `makecom_g5()` - make a packet for SCSI commands

```c
#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int fl, int cmd, int addr, int cnt);
void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp, int fl, int cmd, int cnt, int fixbit);
void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int fl, int cmd, int cnt);
void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int fl, int cmd, int cnt);
```

These interfaces are obsolete. `scsi_setup_cdb(9F)` should be used instead.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Level</th>
<th>Parameters</th>
<th>Description</th>
<th>Context</th>
<th>Examples</th>
</tr>
</thead>
</table>
| `makecom` | Kernel Functions for Drivers | `pkt` Pointer to an allocated `scsi_pkt(9S)` structure. | `makecom` functions initialize a packet with the specified command descriptor block, `devp` and transport flags. The `pkt_address`, `pkt_flags`, and the command descriptor block pointed to by `pkt_cdbp` are initialized using the remaining arguments. Target drivers may use `makecom_g0()` for Group 0 commands (except for sequential access devices), or `makecom_g0_s()` for Group 0 commands for sequential access devices, or `makecom_g1()` for Group 1 commands, or `makecom_g5()` for Group 5 commands. `fixbit` is used by sequential access devices for accessing fixed block sizes and sets the the tag portion of the SCSI CDB. | These functions can be called from user or interrupt context. | **EXAMPLE 1** Using `makecom` Functions
```c
if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1, (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE, (int) blkno, nblk);
}
```
EXAMPLE 1 Using makecom Functions (Continued)

ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), scsi_setup_cdb(9F), scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES

The makecom_g0(), makecom_g0_s(), makecom_g1(), and makecom_g5() functions are obsolete and will be discontinued in a future release. These functions have been replaced by the scsi_setup_cdb() function. See scsi_setup_cdb(9F).
NAME
makedevice – make device number from major and minor numbers

SYNOPSIS
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

dev_t makedevice(major_t majnum, minor_t minnum);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS

majnum       Major device number.

minnum       Minor device number.

DESCRIPTION
makedevice() creates a device number from a major and minor device number. 
makedevice() should be used to create device numbers so the driver will port easily 
to releases that treat device numbers differently.

RETURN VALUES
The device number, containing both the major number and the minor number, is 
returned. No validation of the major or minor numbers is performed.

CONTEXT
makedevice() can be called from user or interrupt context.

SEE ALSO
getmajor(9F), getminor(9F)
#include <sys/ddi.h>

int max(int int1, int int2);

Architecture independent level 1 (DDI/DKI).

max() compares two signed integers and returns the larger of the two.

The larger of the two numbers.

max() can be called from user or interrupt context.

Writing Device Drivers
<table>
<thead>
<tr>
<th>NAME</th>
<th>min – return the lesser of two integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int min(int int1, int int2);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td></td>
</tr>
<tr>
<td>int1</td>
<td>The first integer.</td>
</tr>
<tr>
<td>int2</td>
<td>The second integer.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>min() compares two signed integers and returns the lesser of the two.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The lesser of the two integers.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>min() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>max(9F)</td>
</tr>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
</tbody>
</table>
physio, minphys – perform physical I/O

#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>

int physio(int (*strat)(struct buf *), struct buf *bp, dev_t dev, int rw, void (*mincnt)(struct buf *), struct uio *uio);

void minphys(struct buf *bp);

Solaris DDI specific (Solaris DDI).

strat Pointer to device strategy routine.
bp Pointer to a buf(9S) structure describing the transfer. If bp is set to NULL then physio() allocates one which is automatically released upon completion.
dev The device number.
rw Read/write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.
mincnt Routine which bounds the maximum transfer unit size.
uio Pointer to the uio structure which describes the user I/O request.

DESCRIPTION physio() performs unbuffered I/O operations between the device dev and the address space described in the uio structure.

Prior to the start of the transfer physio() verifies the requested operation is valid by checking the protection of the address space specified in the uio structure. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, strat(), is then called one or more times to perform the physical I/O operations. physio() uses biowait(9F) to block until strat() has completed each transfer. Upon completion, or detection of an error, physio() unlocks the pages and returns the error status.

physio() uses mincnt() to bound the maximum transfer unit size to the system, or device, maximum length. minphys() is the system mincnt() routine for use with physio() operations. Drivers which do not provide their own local mincnt() routines should call physio() with minphys().

minphys() limits the value of bp->b_bcount to a sensible default for the capabilities of the system. Drivers that provide their own mincnt() routine should also call minphys() to make sure they do not exceed the system limit.

RETURN VALUES physio() returns:
0 Upon success.
Upon failure, `physio()` can be called from user context only. When a call to `physio()` returns a non-zero value, it indicates failure. The drivers are responsible for calling `biodone(9F)` when the transfer is complete, otherwise `physio()` will block forever.

**SEE ALSO**
- `strategy(9E)`
- `biodone(9F)`
- `biowait(9F)`
- `buf(9S)`
- `uio(9S)`

**Writing Device Drivers**

**WARNINGS**

Since `physio()` calls `biowait()` to block until each buf transfer is complete, it is the drivers' responsibility to call `biodone(9F)` when the transfer is complete, or `physio()` will block forever.
**mkiocb(9F)**

**NAME**
mkiocb – allocates a STREAMS ioctl block for M_IOCTL messages in the kernel.

**SYNOPSIS**

```c
#include <sys/stream.h>

mblk_t *mkiocb(uint_t command);
```

**INTERFACE**

Solaris DDI specific (Solaris DDI).

**LEVEL**

PARAMETERS

- `command` The ioctl command for the ioc_cmd field.

**DESCRIPTION**

STREAMS modules or drivers might need to issue an ioctl to a lower module or driver. The `mkiocb()` function tries to allocate (using `allocb(9F)`) a STREAMS M_IOCTL message block (`iocblk(9S)`). Buffer allocation fails only when the system is out of memory. If no buffer is available, the `qbufcall(9F)` function can help a module recover from an allocation failure.

The `mkiocb` function returns a `mblk_t` structure which is large enough to hold any of the ioctl messages (`iocblk(9S)`, `copyreq(9S)` or `copyresp(9S)`), and has the following special properties:

- `b_wptr` Set to `b_rptr + sizeof(struct iocblk)`.
- `b_cont` Set to `NULL`.
- `b_datap->db_type` Set to `M_IOCTL`.

The fields in the iocblk structure are initialized as follows:

- `ioc_cmd` Set to the command value passed in.
- `ioc_id` Set to a unique identifier.
- `ioc_cr` Set to point to a credential structure encoding the maximum system privilege and which does not need to be freed in any fashion.
- `ioc_count` Set to 0.
- `ioc_rval` Set to 0.
- `ioc_error` Set to 0.
- `ioc_flags` Set to `IOC_NATIVE` to reflect that this is native to the running kernel.

**RETURN VALUES**

Upon success, the `mkiocb()` function returns a pointer to the allocated `mblk_t` of type `M_IOCTL`.

On failure, it returns a null pointer.

**CONTEXT**

The `mkiocb()` function can be called from user or interrupt context.
EXAMPLE 1 M_IOCTL Allocation

The first example shows an M_IOCTL allocation with the ioctl command TEST_CMD. If the iocblk(9S) cannot be allocated, NULL is returned, indicating an allocation failure (line 5). In line 11, the putnext(9F) function is used to send the message downstream.

```c
1 test_function(queue_t *q, test_info_t *testinfo)  
2 {  
3    mblk_t *mp;  
4    if ((mp = mkiocb(TEST_CMD)) == NULL)  
5      return (0);  
6  
7    /* save off ioctl ID value */  
8    testinfo->xx_iocid = ((struct iocblk *)mp->b_rptr)->ioc_id;  
9    putnext(q, mp);    /* send message downstream */  
10   return (1);  
11 }
```

EXAMPLE 2 The ioctl ID Value

During the read service routine, the ioctl ID value for M_IOCACK or M_IOCNACK should equal the ioctl that was previously sent by this module before processing.

```c
1 test_lrsrv(queue_t *q)  
2 {  
3    ...  
4    switch (DB_TYPE(mp)) {  
5      case M_IOCACK:  
6        case M_IOCNACK:  
7          /* Does this match the ioctl that this module sent */  
8            ioc = (struct iocblk*)mp->b_rptr;  
9            if (ioc->ioc_id == testinfo->xx_iocid) {  
10              /* matches, so process the message */  
11                ...  
12                freemsg(mp);  
13            }  
14            break;  
15            ...  
16            ...  
17        }  
18    }
```

EXAMPLE 3 An iocblk Allocation Which Fails

The next example shows an iocblk allocation which fails. Since the open routine is in user context, the caller may block using qbufcall(9F) until memory is available.

```c
1 test_open(queue_t *q, dev_t devp, int oflag, int sflag, cred_t *credp)  
2 {  
3    while ((mp = mkiocb(TEST_IOCTL)) == NULL) {  
4       int id;  
5```
EXAMPLE 3  An iocblk Allocation Which Fails  (Continued)

6    id = qbufcall(q, sizeof (union ioctypes), BPRI_HI,
7        dummy_callback, 0);
8    /* Handle interrupts */
9    if (!qwait_sig(q)) {
10       qunbufcall(q, id);
11       return (EINTR);
12    }
13    putnext(q, mp);
14    }

SEE ALSO  allocb(9F), putnext(9F), qbufcall(9F), qwait_sig(9F), copyreq(9S),
           copyresp(9S), iocblk(9S)

Writing Device Drivers

STREAMS Programming Guide

WARNINGS  It is the module’s responsibility to remember the ID value of the M_IOCTL that was
allocated. This will ensure proper cleanup and ID matching when the M_IOCTLACK or
M_IOCTLNACK is received.
mod_install, mod_remove, mod_info – add, remove or query a loadable module

```
#include <sys/modctrl.h>

int mod_install(struct modlinkage *modlinkage);
int mod_remove(struct modlinkage *modlinkage);
int mod_info(struct modlinkage *modlinkage, struct modinfo *modinfo);
```

Solaris DDI specific (Solaris DDI).

**modlinkage**
Pointer to the loadable module’s modlinkage structure which describes what type(s) of module elements are included in this loadable module.

**modinfo**
Pointer to the modinfo structure passed to _info(9E).

**DESCRIPTION**

mod_install() must be called from a module’s _init(9E) routine.

mod_remove() must be called from a module’s _fini(9E) routine.

mod_info() must be called from a module’s _info(9E) routine.

**RETURN VALUES**

mod_install() and mod_remove() return 0 upon success and non-zero on failure.

mod_info() returns a non-zero value on success and 0 upon failure.

**EXAMPLES**

See _init(9E) for an example that uses these functions.

**SEE ALSO**

_fini(9E), _info(9E), _init(9E), modldr(9S), modlinkage(9S), modlstrmod(9S)

*Writing Device Drivers*
mod_install(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>mod_install, mod_remove, mod_info – add, remove or query a loadable module</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/modctl.h&gt;</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
<tr>
<td>LEVEL PARAMETERS</td>
<td>Pointer to the loadable module’s modlinkage structure which describes what type(s) of module elements are included in this loadable module.</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>Pointer to the modinfo structure passed to _info(9E).</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>mod_install() must be called from a module’s _init(9E) routine.</td>
</tr>
<tr>
<td></td>
<td>mod_remove() must be called from a module’s _fini(9E) routine.</td>
</tr>
<tr>
<td></td>
<td>mod_info() must be called from a module’s _info(9E) routine.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>mod_install() and mod_remove() return 0 upon success and non-zero on failure.</td>
</tr>
<tr>
<td></td>
<td>mod_info() returns a non-zero value on success and 0 upon failure.</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>See _init(9E) for an example that uses these functions.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>_fini(9E), _info(9E), _init(9E), modldr(9S), modlinkage(9S), modlstrmod(9S)</td>
</tr>
</tbody>
</table>

Writing Device Drivers
mod_install, mod_remove, mod_info — add, remove or query a loadable module

```c
#include <sys/modctl.h>

int mod_install(struct modlinkage *modlinkage);
int mod_remove(struct modlinkage *modlinkage);
int mod_info(struct modlinkage *modlinkage, struct modinfo *modinfo);
```

Solaris DDI specific (Solaris DDI).

**INTERFACE LEVEL PARAMETERS**

- `modlinkage`: Pointer to the loadable module’s `modlinkage` structure which describes what type(s) of module elements are included in this loadable module.
- `modinfo`: Pointer to the `modinfo` structure passed to `_info(9E)`.

**DESCRIPTION**

- `mod_install()` must be called from a module’s `_init(9E)` routine.
- `mod_remove()` must be called from a module’s `_fini(9E)` routine.
- `mod_info()` must be called from a module’s `_info(9E)` routine.

**RETURN VALUES**

- `mod_install()` and `mod_remove()` return 0 upon success and non-zero on failure.
- `mod_info()` returns a non-zero value on success and 0 upon failure.

**EXAMPLES**

See `_init(9E)` for an example that uses these functions.

**SEE ALSO**

- `_fini(9E), _info(9E), _init(9E), modldrv(9S), modlinkage(9S), modlstrmod(9S)`

*Writing Device Drivers*
msgdsize(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>msgdsize – return the number of bytes in a message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>mp Message to be evaluated.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>msgdsize() counts the number of bytes in a data message. Only bytes included in the data blocks of type M_DATA are included in the count.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The number of data bytes in a message, expressed as an integer.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>msgdsize() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>See bufcall(9F) for an example that uses msgdsize().</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>bufcall(9F)</td>
</tr>
</tbody>
</table>

Writing Device Drivers

STREAMS Programming Guide
**NAME**
msgpullup – concatenate bytes in a message

**SYNOPSIS**
```
#include <sys/stream.h>

mblk_t *msgpullup(mblk_t *mp, ssize_t len);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `mp` Pointer to the message whose blocks are to be concatenated.
- `len` Number of bytes to concatenate.

**DESCRIPTION**
`msgpullup()` concatenates and aligns the first `len` data bytes of the message pointed to by `mp`, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered. If `len` equals −1, all data are concatenated. If `len` bytes of the same message type cannot be found, `msgpullup()` fails and returns NULL.

**RETURN VALUES**
`msgpullup` returns the following values:
- Non-null Successful completion. A pointer to the new message is returned.
- NULL An error occurred.

**CONTEXT**
`msgpullup()` can be called from user or interrupt context.

**SEE ALSO**
srv(9E), allocb(9F), pullupmsg(9F), msgb(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**
`msgpullup()` is a DKI-compliant replacement for the older pullupmsg(9F) routine. Users are strongly encouraged to use `msgpullup()` instead of pullupmsg(9F).
mt-streams(9F)

NAME  mt-streams – STREAMS multithreading
SYNOPSIS  
```
#include <sys/conf.h>
```

INTERFACE LEVEL
DESCRIPTION  Solaris DDI specific (Solaris DDI).

STREAMS drivers configures the degree of concurrency using the cb_flag field in the cb_ops structure (see cb_ops(9S)). The corresponding field for STREAMS modules is the f_flag in the fmodsw structure.

For the purpose of restricting and controlling the concurrency in drivers/modules, we define the concepts of inner and outer perimeters. A driver/module can be configured either to have no perimeters, to have only an inner or an outer perimeter, or to have both an inner and an outer perimeter. Each perimeter acts as a readers-writers lock, that is, there can be multiple concurrent readers or a single writer. Thus, each perimeter can be entered in two modes: shared (reader) or exclusive (writer). The mode depends on the perimeter configuration and can be different for the different STREAMS entry points (open(9E), close(9E), put(9E), or srv(9E)).

The concurrency for the different entry points is (unless specified otherwise) to enter with exclusive access at the inner perimeter (if present) and shared access at the outer perimeter (if present).

The perimeter configuration consists of flags that define the presence and scope of the inner perimeter, the presence of the outer perimeter (which can only have one scope), and flags that modify the default concurrency for the different entry points.

All MT safe modules/drivers specify the D_MP flag.

### Inner Perimeter Flags

The inner perimeter presence and scope are controlled by the mutually exclusive flags:

- **D_MTPERQ**: The module/driver has an inner perimeter around each queue.
- **D_MTQPAIR**: The module/driver has an inner perimeter around each read/write pair of queues.
- **D_MTPERMOD**: The module/driver has an inner perimeter that encloses all the module’s/driver’s queues.
- **None of the above**: The module/driver has no inner perimeter.

### Outer Perimeter Flags

The outer perimeter presence is configured using:

- **D_MTOUTPERIM**: In addition to any inner perimeter, the module/driver has an outer perimeter that encloses all the module’s/driver’s queues. This can be combined with all the inner perimeter options except D_MTPERMOD.

Note that acquiring exclusive access at the outer perimeter (that is, using qwriter(9F) with the PERIM_OUTER flag) can incur significant performance penalties, which grow linearly with the number of open instances of the module or driver in the system.
The default concurrency can be modified using:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_MTPUTSHARED</td>
<td>This flag modifies the default behavior when put(9E) procedure are invoked so that the inner perimeter is entered shared instead of exclusively.</td>
</tr>
<tr>
<td>D_MTOCEXCL</td>
<td>This flag modifies the default behavior when open(9E) and close(9E) procedures are invoked so the outer perimeter is entered exclusively instead of shared.</td>
</tr>
</tbody>
</table>

Note that drivers and modules using this flag can cause significant system performance degradation during stream open or close when many instances of the driver or module are in use simultaneously. For this reason, use of this flag is discouraged. Instead, since open(9E) and close(9E) both execute with user context, developers are encouraged to use traditional synchronization routines such as cv_wait_sig(9F) to coordinate with other open instances of the module or driver.

The module/driver can use qwait(9F) or qwait_sig() in the open(9E) and close(9E) procedures if it needs to wait "outside" the perimeters.

The module/driver can use qwriter(9F) to upgrade the access at the inner or outer perimeter from shared to exclusive.

The use and semantics of qprocson() and qprocsoff(9F) is independent of the inner and outer perimeters.

**SEE ALSO**

- close(9E), open(9E), put(9E), srv(9E), qprocsoff(9F), qprocson(9F), qwait(9F), qwriter(9F), cb_ops(9S)

**STREAMS Programming Guide**

**Writing Device Drivers**
## INTERFACE

Solaris DDI specific (Solaris DDI).

### LEVEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mp</code></td>
<td>Pointer to a kernel mutex lock (<code>kmutex_t</code>).</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Descriptive string. This is obsolete and should be <code>NULL</code>. (Non-<code>NULL</code> strings are legal, but they are a waste of kernel memory.)</td>
</tr>
<tr>
<td><code>type</code></td>
<td>Type of mutex lock.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Type-specific argument for initialization routine.</td>
</tr>
</tbody>
</table>

### DESCRIPTION

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

`mutex_init()` initializes a mutex. It is an error to initialize a mutex more than once. The `type` argument should be set to `MUTEX_DRIVER`.

`arg` provides type-specific information for a given variant type of mutex. When `mutex_init()` is called for driver mutexes, if the mutex is used by the interrupt handler, the `arg` should be the value of `iblock_cookie` returned from `ddi_get_iblock_cookie()` or `ddi_get_soft_iblock_cookie()`. Note that `arg` should be cast to `void*`, not the address of the cookie. The arguments passed to `ddi_get_iblock_cookie()` and `ddi_get_soft_iblock_cookie()`, on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be `NULL`. 
mutex_enter() is used to acquire a mutex. If the mutex is already held, then the
caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex
is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not
being defined unless the preprocessor symbol DEBUG is defined. Its return value is
non-zero if the current thread (or, if that cannot be determined, at least some thread)
holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block
when the mutex is already held. mutex_tryenter() returns non-zero when it
acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked
on the mutex.

mutex_destroy() releases any resources that might have been allocated by
mutex_init(). mutex_destroy() must be called before freeing the memory
containing the mutex, and should be called with the mutex unheld (not owned by any
thread). The caller must somehow be sure that no other thread will attempt to use the
mutex.

mutex_destroy() returns non-zero on success and zero of failure.

mutex_owned() returns non-zero if the calling thread currently holds the mutex
pointed to by mp, or when that cannot be determined, if any thread holds the mutex.
mutex_owned() returns zero otherwise.

These functions can be called from user, kernel, or high-level interrupt context, except
for mutex_init() and mutex_destroy(), which can be called from user or kernel
context only.

EXAMPLE 1 Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in
its interrupt routine:

ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
(void *)iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
(caddr_t)un);

EXAMPLE 2 Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following
ASSERT:

xxstart(struct xxunit *un)
{
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));
    ...

SEE ALSO  lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
          ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
        lock statistics, see lockstat(1M).
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

```c
#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);
```

**NAME**
mutex_destroy(9F)

**SYNOPSIS**

```c
void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);
```

**INTERFACE LEVEL PARAMETERS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp</td>
<td>Pointer to a kernel mutex lock (kmutex_t).</td>
</tr>
<tr>
<td>name</td>
<td>Descriptive string. This is obsolete and should be NULL.</td>
</tr>
<tr>
<td>type</td>
<td>Type of mutex lock.</td>
</tr>
<tr>
<td>arg</td>
<td>Type-specific argument for initialization routine.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

```c
mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.
```

arg provides type-specific information for a given variant type of mutex. When `mutex_init()` is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from `ddi_get_iblock_cookie(9F)` or `ddi_get_soft_iblock_cookie(9F)`. Note that arg should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to `ddi_get_iblock_cookie(9F)` and `ddi_get_soft_iblock_cookie(9F)`, on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_destroy() is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not being defined unless the preprocessor symbol DEBUG is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block when the mutex is already held. mutex_tryenter() returns non-zero when it acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked on the mutex.

mutex_destroy() releases any resources that might have been allocated by mutex_init(). mutex_destroy() must be called before freeing the memory containing the mutex, and should be called with the mutex unheld (not owned by any thread). The caller must somehow be sure that no other thread will attempt to use the mutex.

RETURN VALUES
mutex_tryenter() returns non-zero on success and zero on failure.
mutex_owned() returns non-zero if the calling thread currently holds the mutex pointed to by mp, or when that cannot be determined, if any thread holds the mutex.
mutex_owned() returns zero otherwise.

CONTEXT
These functions can be called from user, kernel, or high-level interrupt context, except for mutex_init() and mutex_destroy(), which can be called from user or kernel context only.

EXAMPLES

**EXAMPLE 1** Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
(void *)iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
(caddr_t)un);

**EXAMPLE 2** Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following ASSERT:

xxstart(struct xxunit *un)
{
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

        ASSERT(mutex_owned(&un->un_lock));
        ...

SEE ALSO lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
       ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

```c
#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);
```

Solaris DDI specific (Solaris DDI).

### INTERFACE

- **mp**: Pointer to a kernel mutex lock (`kmutex_t`).
- **name**: Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
- **type**: Type of mutex lock.
- **arg**: Type-specific argument for initialization routine.

### LEVEL

### PARAMETERS

### DESCRIPTION

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

`mutex_init()` initializes a mutex. It is an error to initialize a mutex more than once. The `type` argument should be set to `MUTEX_DRIVER`.

`arg` provides type-specific information for a given variant type of mutex. When `mutex_init()` is called for driver mutexes, if the mutex is used by the interrupt handler, the `arg` should be the `ddi_iblock_cookie` returned from `ddi_get_iblock_cookie(9F)` or `ddi_get_soft_iblock_cookie(9F)`. Note that `arg` should be the value of the iblock cookie casted to `(void*)`, not the address of the cookie. The arguments passed to `ddi_get_iblock_cookie(9F)` and `ddi_get_soft_iblock_cookie(9F)`, on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_enter() is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not being defined unless the preprocessor symbol DEBUG is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block when the mutex is already held. mutex_tryenter() returns non-zero when it acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked on the mutex.

mutex_destroy() releases any resources that might have been allocated by mutex_init(). mutex_destroy() must be called before freeing the memory containing the mutex, and should be called with the mutex unheld (not owned by any thread). The caller must somehow be sure that no other thread will attempt to use the mutex.

RETURN VALUES
mutex_tryenter() returns non-zero on success and zero of failure.
mutex_owned() returns non-zero if the calling thread currently holds the mutex pointed to by mp, or when that cannot be determined, if any thread holds the mutex. mutex_owned() returns zero otherwise.

CONTEXT
These functions can be called from user, kernel, or high-level interrupt context, except for mutex_init() and mutex_destroy(), which can be called from user or kernel context only.

EXAMPLES
EXAMPLE 1 Initializing a Mutex
A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

def_get_iblock_cookie(dip, 0, &iblock);
m_mutex_init(&un->un_lock, NULL, MUTE Dương_DRIVER, (void *)iblock);
def_add_intr(dip, 0, NULL, &dev_cookie, xxintr, (caddr_t)un);

EXAMPLE 2 Calling a Routine with a Lock
A routine that expects to be called with a certain lock held might have the following ASSERT:

xxstart(struct xxunit *un)
{
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));
    ...

SEE ALSO  lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
          ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);

Solaris DDI specific (Solaris DDI).

**INTERFACE**

**LEVEL**

**PARAMETERS**

- **mp** Pointer to a kernel mutex lock (kmutex_t).
- **name** Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
- **type** Type of mutex lock.
- **arg** Type-specific argument for initialization routine.

**DESCRIPTION**

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.

arg provides type-specific information for a given variant type of mutex. When mutex_init() is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). Note that arg should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to ddi_get_iblock_cookie(9F) and ddi_get_soft_iblock_cookie(9F), on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_exit(9F)

mutex_enter() is used to acquire a mutex. If the mutex is already held, then the
caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex
is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not
being defined unless the preprocessor symbol DEBUG is defined. Its return value is
non-zero if the current thread (or, if that cannot be determined, at least some thread)
holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block
when the mutex is already held. mutex_tryenter() returns non-zero when it
acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked
on the mutex.

mutex_destroy() releases any resources that might have been allocated by
mutex_init().mutex_destroy() must be called before freeing the memory
containing the mutex, and should be called with the mutex unheld (not owned by any
thread). The caller must somehow be sure that no other thread will attempt to use the
mutex.

RETURN VALUES

mutex_tryenter() returns non-zero on success and zero of failure.

mutex_owned() returns non-zero if the calling thread currently holds the mutex
pointed to by mp, or when that cannot be determined, if any thread holds the mutex.
mutex_owned() returns zero otherwise.

CONTEXT

These functions can be called from user, kernel, or high-level interrupt context, except
for mutex_init() and mutex_destroy(), which can be called from user or kernel
context only.

EXAMPLES

EXAMPLE 1 Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in
its interrupt routine:

ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
(void *)iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
(caddr_t)un);

EXAMPLE 2 Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following
ASSERT:

xxstart(struct xxunit *un)
	{
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));

    ...

SEE ALSO  lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
           ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
       lock statistics, see lockstat(1M).
mutex_init(9F)

NAME
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

SYNOPSIS
#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
mp Pointer to a kernel mutex lock (kmutex_t).
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
type Type of mutex lock.
arg Type-specific argument for initialization routine.

DESCRIPTION
A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.

arg provides type-specific information for a given variant type of mutex. When mutex_init() is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). Note that arg should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to ddi_get_iblock_cookie(9F) and ddi_get_soft_iblock_cookie(9F), on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_enter() is used to acquire a mutex. If the mutex is already held, then the
caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex
is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not
being defined unless the preprocessor symbol DEBUG is defined. Its return value is
non-zero if the current thread (or, if that cannot be determined, at least some thread)
holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block
when the mutex is already held. mutex_tryenter() returns non-zero when it
acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked
on the mutex.

mutex_destroy() releases any resources that might have been allocated by
mutex_init(). mutex_destroy() must be called before freeing the memory
containing the mutex, and should be called with the mutex unheld (not owned by any
thread). The caller must somehow be sure that no other thread will attempt to use the
mutex.

mutex_tryenter() returns non-zero on success and zero of failure.

mutex_owned() returns non-zero if the calling thread currently holds the mutex
pointed to by mp, or when that cannot be determined, if any thread holds the mutex.
mutex_owned() returns zero otherwise.

RETURN VALUES

mutex_destroy() must be called before freeing the memory

CONTEXT

These functions can be called from user, kernel, or high-level interrupt context, except
for mutex_init() and mutex_destroy(), which can be called from user or kernel
context only.

EXAMPLES

EXAMPLE 1 Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in
its interrupt routine:

```
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
(void *)iblock);
```

EXAMPLE 2 Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following
ASSERT:

```
xstart(struct xxunit *un) 
```
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));
    ...

SEE ALSO lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);

Solaris DDI specific (Solaris DDI).

**INTERFACE LEVEL**

**PARAMETERS**

<table>
<thead>
<tr>
<th>mp</th>
<th>Pointer to a kernel mutex lock (kmutex_t).</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)</td>
</tr>
<tr>
<td>type</td>
<td>Type of mutex lock.</td>
</tr>
<tr>
<td>arg</td>
<td>Type-specific argument for initialization routine.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.

arg provides type-specific information for a given variant type of mutex. When mutex_init() is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). Note that arg should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to ddi_get_iblock_cookie(9F) and ddi_get_soft_iblock_cookie(9F), on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_enter() is used to acquire a mutex. If the mutex is already held, then the
caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex
is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not
being defined unless the preprocessor symbol DEBUG is defined. Its return value is
non-zero if the current thread (or, if that cannot be determined, at least some thread)
holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block
when the mutex is already held. mutex_tryenter() returns non-zero when it
acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked
on the mutex.

mutex_destroy() releases any resources that might have been allocated by
mutex_init(). mutex_destroy() must be called before freeing the memory
containing the mutex, and should be called with the mutex unheld (not owned by any
thread). The caller must somehow be sure that no other thread will attempt to use the
mutex.

RETURN VALUES
mutex_tryenter() returns non-zero on success and zero of failure.
mutex_owned() returns non-zero if the calling thread currently holds the mutex
pointed to by mp, or when that cannot be determined, if any thread holds the mutex.
mutex_owned() returns zero otherwise.

CONTEXT
These functions can be called from user, kernel, or high-level interrupt context, except
for mutex_init() and mutex_destroy(), which can be called from user or kernel
context only.

EXAMPLES
EXAMPLE 1 Initializing a Mutex
A driver might do this to initialize a mutex that is part of its unit structure and used in
its interrupt routine:

ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
(void *)&iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
(caddr_t)un);

EXAMPLE 2 Calling a Routine with a Lock
A routine that expects to be called with a certain lock held might have the following
ASSERT:

xxstart(struct xxunit *un)
EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));

    ...

SEE ALSO  lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
           ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
       lock statistics, see lockstat(1M).
mutextryenter(9F)

NAME
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutextryenter – mutual exclusion lock routines

SYNOPSIS
#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutextryenter(kmutex_t *mp);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

mp Pointer to a kernel mutex lock (kmutex_t).

name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)

type Type of mutex lock.

arg Type-specific argument for initialization routine.

DESCRIPTION
A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.

arg provides type-specific information for a given variant type of mutex. When mutex_init() is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). Note that arg should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to ddi_get_iblock_cookie(9F) and ddi_get_soft_iblock_cookie(9F), on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.
mutex_enter() is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT() and may be enforced by not being defined unless the preprocessor symbol DEBUG is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by mp.

mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block when the mutex is already held. mutex_tryenter() returns non-zero when it acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked on the mutex.

mutex_destroy() releases any resources that might have been allocated by mutex_init(). mutex_destroy() must be called before freeing the memory containing the mutex, and should be called with the mutex unheld (not owned by any thread). The caller must somehow be sure that no other thread will attempt to use the mutex.

mutex_destroy() returns non-zero on success and zero of failure.

mutex_owned() returns non-zero if the calling thread currently holds the mutex pointed to by mp, or when that cannot be determined, if any thread holds the mutex. mutex_owned() returns zero otherwise.

These functions can be called from user, kernel, or high-level interrupt context, except for mutex_init() and mutex_destroy(), which can be called from user or kernel context only.

EXAMPLE 1 Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

```c
ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER, (void *)iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr, (caddr_t)un);
```

EXAMPLE 2 Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following ASSERT:

```c
xxstart(struct xxunit *un) {  
```
mutex_tryenter(9F)

EXAMPLE 2 Calling a Routine with a Lock  (Continued)

    ASSERT(mutex_owned(&un->un_lock));
...

SEE ALSO  lockstat(1M), condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
nochpoll(9F)

**NAME**
nochpoll – error return function for non-pollable devices

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int nochpoll(dev_t dev, short events, int anyyet, short *reventsp,
             struct pollhead **pollhdrp);
```

**INTERFACE LEVEL PARAMETERS**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>Device number.</td>
</tr>
<tr>
<td>events</td>
<td>Event flags.</td>
</tr>
<tr>
<td>anyyet</td>
<td>Check current events only.</td>
</tr>
<tr>
<td>reventsp</td>
<td>Event flag pointer.</td>
</tr>
<tr>
<td>pollhdrp</td>
<td>Poll head pointer.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**
nochpoll() is a routine that simply returns the value ENXIO. It is intended to be used in the cb_ops(9S) structure of a device driver for devices that do not support the poll(2) system call.

**RETURN VALUES**
nochpoll() returns ENXIO.

**CONTEXT**
nochpoll() can be called from user or interrupt context.

**SEE ALSO**
poll(2), chpoll(9E), cb_ops(9S)

Writing Device Drivers
### nodev(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>nodev – error return function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/conf.h&gt;</code>&lt;br&gt;<code>#include &lt;sys/ddi.h&gt;</code>&lt;br&gt;<code>int nodev();</code></td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td><code>nodev()</code> returns ENXIO. It is intended to be used in the cb_ops(9S) data structure of a device driver for device entry points which are not supported by the driver. That is, it is an error to attempt to call such an entry point.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td><code>nodev()</code> returns ENXIO.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td><code>nodev()</code> can be only called from user context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>nulldev(9F), cb_ops(9S)</code>&lt;br&gt;<code>Writing Device Drivers</code></td>
</tr>
</tbody>
</table>
NAME
noenable – prevent a queue from being scheduled

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

void noenable(queue_t *q);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL

PARAMETERS
q Pointer to the queue.

DESCRIPTION
noenable() prevents the queue q from being scheduled for service by insq(9F), putq(9F) or putbq(9F) when enqueuing an ordinary priority message. The queue can be re-enabled with the enableok(9F) function.

CONTEXT
noenable() can be called from user or interrupt context.

SEE ALSO
enableok(9F), insq(9F), putbq(9F), putq(9F), qenable(9F)

Writing Device Drivers

STREAMS Programming Guide
nulldev(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>nulldev – zero return function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>nulldev()</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>include &lt;sys/conf.h&gt;</td>
</tr>
<tr>
<td>LEVEL</td>
<td>include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>int nulldev();</td>
</tr>
<tr>
<td></td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td></td>
<td>nulldev() returns 0. It is intended to be used in the cb_ops(9S) data structure of a device driver for device entry points that do nothing.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>nulldev() returns 0.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>nulldev() can be called from any context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>nodev(9F), cb_ops(9S)</td>
</tr>
</tbody>
</table>

Writing Device Drivers
numtos(9F)

NAME
stoi, numtos – convert between an integer and a decimal string

SYNOPSIS
#include <sys/ddi.h>

int stoi(char **str);
void numtos(unsigned long num, char *s);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
str   Pointer to a character string to be converted.
num   Decimal number to be converted to a character string.
s     Character buffer to hold converted decimal number.

stoi()  stoi() returns the integer value of a string of decimal numeric characters beginning at **str. No overflow checking is done. *str is updated to point at the last character examined.

numtos() numtos() converts a long into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.

RETURN VALUES
stoi() returns the integer value of the string str.

CONTEXT
stoi() can be called from user or interrupt context.

SEE ALSO
Writing Device Drivers

NOTES
stoi() handles only positive integers; it does not handle leading minus signs.
NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array
– add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
      uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
      uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t
      *val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
      uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t
      *val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
      uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t
      *val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
      uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl          The nvlist_t to be processed.
name          Name of the name-value pair (nvpair).
These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES
0  success
EINVAL  invalid argument
ENOMEM  insufficient memory

CONTEXT
These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl The nvlist_t to be processed.
name Name of the name-value pair (nvpair).
nvlist_add_byte(9F)

nelem     Number of elements in value (that is, array size).
val      Value or starting address of the array value.

DESCRIPTION     These functions adds a new name-value pair to nvlist_t. The memory allocation
                  policy follows that specified in nvlist_alloc(), nvlist_unpack(), or
                  nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data
                  types follow the nvflag argument specified in nvlist_alloc().

                  If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names
                  are removed before the new nvpair is added.

                  If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching
                  names and data types are removed before the new nvpair is added.

                  If neither was specified for nvflag, the new nvpair is unconditionally added at the
                  end of the list. The library preserves the order of the name-value pairs across packing,
                  unpacking, and duplication.

RETURN VALUES

0       success
EINVAL     invalid argument
ENOMEM     insufficient memory

CONTEXT     These functions can be called from interrupt context only if the nvlist_t was
                  allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of
                  KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_byte_array(9F)

NAME nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl The nvlist_t to be processed.
name Name of the name-value pair (nvpair).
These functions add a new name-value pair to `nvlist_t`. The memory allocation policy follows that specified in `nvlist_alloc()`, `nvlist_unpack()`, or `nvlist_dup()`. See `nvlist_alloc(9F)`. The uniqueness of `nvpair` name and data types follow the `nvflag` argument specified in `nvlist_alloc()`. If `NV_UNIQUE_NAME` was specified for `nvflag`, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for `nvflag`, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for `nvflag`, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

**RETURN VALUES**

- **0**: success
- **EINVAL**: invalid argument
- **ENOMEM**: insufficient memory

**CONTEXT**

These functions can be called from interrupt context only if the `nvlist_t` was allocated with the `KM_NOSLEEP` flag set. See `nvlist_alloc(9F)` for a description of `KM_NOSLEEP`. These functions can be called from user context in all cases.
nvlist_add_int16(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl
The nvlist_t to be processed.

name
Name of the name-value pair (nvpair).
These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0    success
EINVAL    invalid argument
ENOMEM    insufficient memory

CONTEXT

These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
**NAME**

nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array

**SYNOPSIS**

```
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val, uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val, uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val, uint_t nelem);
```
nvlist_add_int16_array(9F)

nelem      Number of elements in value (that is, array size).
val       Value or starting address of the array value.

DESCRIPTION
These functions add a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES
0          success
EINVAL     invalid argument
ENOMEM      insufficient memory

CONTEXT
These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_int32(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                          uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                           uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *
                           val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *
                           val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *
                           val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *
                           val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *
                           val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                           uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

PARAMETERS
nvl
The nvlist_t to be processed.

name
Name of the name-value pair (nvpair).
nelem  Number of elements in value (that is, array size).

val  Value or starting address of the array value.

DESCRIPTION  These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES  

0  success

EINVAL  invalid argument

ENOMEM  insufficient memory

CONTEXT  These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_int32_array(9F)

NAME nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
 nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
 nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array, nvlist_add_uint16_array,
 nvlist_add_int32_array, nvlist_add_uint32_array, nvlist_add_int64_array, nvlist_add_uint64_array,
 nvlist_add_string_array – add new name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val, uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val, uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val, uint_t nelem);

INTERFACE LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI)

nvl The nvlist_t to be processed.

name Name of the name-value pair (nvpair).
nvlist_add_int32_array(9F)

DESCRIPTION

These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0              success
EINVAL          invalid argument
ENOMEM          insufficient memory

CONTEXT

These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_int64(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16, nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64, nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array, nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array, nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val, uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val, uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val, uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl The nvlist_t to be processed.
name Name of the name-value pair (nvpair).
DESCRIPTION

These functions add a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc(). If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0 success
EINVAL invalid argument
ENOMEM insufficient memory

CONTEXT

These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
## NAME

nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

## SYNOPSIS

```
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                         uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                         uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
                         uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
                         uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
                         uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
                         uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
                         uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                         uint_t nelem);
```
nvlist_add_int64_array(9F)

**nelem**
Number of elements in value (that is, array size).

**val**
Value or starting address of the array value.

**DESCRIPTION**
These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the *nvflag* argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for *nvflag*, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for *nvflag*, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for *nvflag*, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

**RETURN VALUES**
0 success
EINVAL invalid argument
ENOMEM insufficient memory

**CONTEXT**
These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_string(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
uint_t nelem);

PARAMETERS

nvlist_t *nvl
The nvlist_t to be processed.

char *name
Name of the name-value pair (nvpair).
These functions add a new name-value pair to `nvlist_t`. The memory allocation policy follows that specified in `nvlist_alloc()`, `nvlist_unpack()`, or `nvlist_dup()`. See `nvlist_alloc(9F)`. The uniqueness of `nvpair` name and data types follow the `nvflag` argument specified in `nvlist_alloc()`. If `NV_UNIQUE_NAME` was specified for `nvflag`, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for `nvflag`, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for `nvflag`, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

### RETURN VALUES
- **0**  success
- **EINVAL** invalid argument
- **ENOMEM** insufficient memory

### CONTEXT
These functions can be called from interrupt context only if the `nvlist_t` was allocated with the `KM_NOSLEEP` flag set. See `nvlist_alloc(9F)` for a description of `KM_NOSLEEP`. These functions can be called from user context in all cases.

---

**DESCRIPTION**

- **nelem**  Number of elements in value (that is, array size).
- **val** Value or starting address of the array value.
nvlist_add_string_array(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                          uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                           uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *
                           val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *
                           val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *
                           val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *
                           val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *
                           val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                           uint_t nelem);

INTERFACE
Solari DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl The nvlist_t to be processed.
name Name of the name-value pair (nvpair).
**DESCRIPTION**

These functions add a new name-value pair to `nvlist_t`. The memory allocation policy follows that specified in `nvlist_alloc()`, `nvlist_unpack()`, or `nvlist_dup()`. See `nvlist_alloc(9F)`. The uniqueness of `nvpair` name and data types follow the `nvflag` argument specified in `nvlist_alloc()`.

If `NV_UNIQUE_NAME` was specified for `nvflag`, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for `nvflag`, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for `nvflag`, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

**RETURN VALUES**

- 0: success
- EINVAL: invalid argument
- ENOMEM: insufficient memory

**CONTEXT**

These functions can be called from interrupt context only if the `nvlist_t` was allocated with the `KM_NOSLEEP` flag set. See `nvlist_alloc(9F)` for a description of `KM_NOSLEEP`. These functions can be called from user context in all cases.
nvlist_add_uint16(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
uint_t nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvl</td>
<td>The nvlist_t to be processed.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the name-value pair (nvpair).</td>
</tr>
</tbody>
</table>
nvlist_add_uint16(9F)

nenlem Number of elements in value (that is, array size).

val Value or starting address of the array value.

DESCRIPTION These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0 success

EINVAL invalid argument

ENOMEM insufficient memory

CONTEXT These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_uint16_array(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array, nvlist_add_int64_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                        uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                       uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
                       uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
                       uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
                       uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
                       uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
                       uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                       uint_t nelem);

INTERFACE
SOLARIS DDI SPECIFIC (SOLARIS DDI)

LEVEL
PARAMETERS
nvl
The nvlist_t to be processed.
name
Name of the name-value pair (nvpair).
These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0 success
EINVAL invalid argument
ENOMEM insufficient memory

CONTEXT

These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_add_uint32(9F)

NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                         uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                          uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
                          uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
                          uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
                          uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
                          uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
                          uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                          uint_t nelem);

INTERFACE
LEVEL
PARAMETERS
Solaris DDI specific (Solaris DDI)

nvl
The nvlist_t to be processed.

name
Name of the name-value pair (nvpair).
nvlist_add_uint32(9F)

nelem Number of elements in value (that is, array size).

val Value or starting address of the array value.

DESCRIPTION These functions adds a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES

0 success
EINVAL invalid argument
ENOMEM insufficient memory

CONTEXT These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
## NAME
nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16,
nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64,
nvlist_add_string, nvlist_add_byte_array, nvlist_add_int16_array,
nvlist_add_uint16_array, nvlist_add_int32_array, nvlist_add_uint32_array,
nvlist_add_int64_array, nvlist_add_uint64_array, nvlist_add_string_array – add new
name-value pair to nvlist_t

## SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                       uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                        uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
                      uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
                     uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
                   uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
                    uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
                   uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                    uint_t nelem);

## INTERFACE LEVEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvl</td>
<td>The nvlist_t to be processed.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the name-value pair (nvpair).</td>
</tr>
</tbody>
</table>
These functions adds a new name-value pair to `nvlist_t`. The memory allocation policy follows that specified in `nvlist_alloc()`, `nvlist_unpack()`, or `nvlist_dup()`. See `nvlist_alloc(9F)`. The uniqueness of `nvpair` name and data types follow the `nvfl`ag argument specified in `nvlist_alloc()`. If `NV_UNIQUE_NAME` was specified for `nvfl`ag, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for `nvfl`ag, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for `nvfl`ag, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

**RETURN VALUES**

- **0** success
- **EINVAL** invalid argument
- **ENOMEM** insufficient memory

**CONTEXT**

These functions can be called from interrupt context only if the `nvlist_t` was allocated with the `KM_NOSLEEP` flag set. See `nvlist_alloc(9F)` for a description of `KM_NOSLEEP`. These functions can be called from user context in all cases.
**SYNOPSIS**

```
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val, uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val, uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val, uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val, uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val, uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val, uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val, uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val, uint_t nelem);
```
These functions adds a new name-value pair to `nvlist_t`. The memory allocation policy follows that specified in `nvlist_alloc()`, `nvlist_unpack()`, or `nvlist_dup()`. See `nvlist_alloc(9F)`. The uniqueness of `nvpair` name and data types follow the `nvflag` argument specified in `nvlist_alloc()`.

If `NV_UNIQUE_NAME` was specified for `nvflag`, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for `nvflag`, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for `nvflag`, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

**RETURN VALUES**

- `0`  success
- `EINVAL`  invalid argument
- `ENOMEM`  insufficient memory

**CONTEXT**

These functions can be called from interrupt context only if the `nvlist_t` was allocated with the `KM_NOSLEEP` flag set. See `nvlist_alloc(9F)` for a description of `KM_NOSLEEP`. These functions can be called from user context in all cases.
SYNOPSIS
#include <sys/nvpair.h>

int nvlist_add_boolean(nvlist_t *nvl, char *name);
int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val);
int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val);
int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val);
int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val);
int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val);
int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val);
int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val);
int nvlist_add_string(nvlist_t *nvl, char *name, char *val);
int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,
                          uint_t nelem);
int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,
                           uint_t nelem);
int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t *val,
                           uint_t nelem);
int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,
                           uint_t nelem);
int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t *val,
                           uint_t nelem);
int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,
                           uint_t nelem);
int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t *val,
                           uint_t nelem);
int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,
                           uint_t nelem);

PARAMETERS

nvl
The nvlist_t to be processed.

name
Name of the name-value pair (nvpair).
DESCRIPTION
These functions add a new name-value pair to nvlist_t. The memory allocation policy follows that specified in nvlist_alloc(), nvlist_unpack(), or nvlist_dup(). See nvlist_alloc(9F). The uniqueness of nvpair name and data types follow the nvflag argument specified in nvlist_alloc().

If NV_UNIQUE_NAME was specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added.

If NV_UNIQUE_NAME_TYPE was specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added.

If neither was specified for nvflag, the new nvpair is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

RETURN VALUES
0       success
EINVAL   invalid argument
ENOMEM    insufficient memory

CONTEXT
These functions can be called from interrupt context only if the nvlist_t was allocated with the KM_NOSLEEP flag set. See nvlist_alloc(9F) for a description of KM_NOSLEEP. These functions can be called from user context in all cases.
nvlist_alloc(9F)

NAME   nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a name-value pair list

SYNOPSIS  
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag, int kmflag);
void nvlist_free(nvlist_t *nvl);
int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);
int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);
int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);
int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);

INTERFACE Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvlp       Address of a pointer to list of name-value pairs (nvlist_t).
nvflag     Specify bit fields defining nvlist_t properties:
            NV_UNIQUE_NAME
            The nvpair names are unique.
            NV_UNIQUE_NAME_TYPE
            Name-data type combination is unique

kmflag     Kernel memory allocation policy, either KM_SLEEP or KM_NOSLEEP.
nvl        The nvlist_t to be processed.
size       Pointer to buffer to contain the encoded size.
bufp       Address of buffer to pack nvlist into. Must be 8-byte aligned. If NULL, library will allocate memory.
buf        Buffer containing packed nvlist_t.
buflen     Size of buffer bufp or buf points to.
encoding   Encoding method for packing.

DESCRIPTION  
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp to point to the handle. The argument nvflag specifies nvlist_t properties to remain persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large enough to pack nvl. The encoding parameter specifies the method of encoding when packing nvl. Supported encoding methods are:
nvlist_alloc(9F)

| **NV_ENCODE_NATIVE** | Straight `bcopy()` as described in `bcopy(9F)`.
| **NV_ENCODE_XDR** | Use XDR encoding, suitable for sending to another host.

The `nvlist_pack()` function packs `nvlist` into contiguous memory starting at `*bufp`. The `encoding` parameter specifies the method of encoding (see above).

- If `*bufp` is not `NULL`, `*bufp` is expected to be a caller-allocated buffer of size `*buflen`. The `kmflag` argument is ignored.
- If `*bufp` is `NULL`, the library will allocate memory and update `*bufp` to point to the memory and update `*buflen` to contain the size of the allocated memory. The value of `kmflag` indicates the memory allocation policy.

The `nvlist_unpack()` function takes a buffer with a packed `nvlist_t` and unpacks it into a searchable `nvlist_t`. The library allocates memory for `nvlist_t`. The caller is responsible for freeing the memory by calling `nvlist_free()`.

The `nvlist_dup()` function makes a copy of `nvlist` and updates `nvlp` to point to the copy.

**RETURN VALUES**

For `nvlist_alloc()` and `nvlist_dup()`:

- `0` success
- `EINVAL` invalid argument
- `ENOMEM` insufficient memory

For `nvlist_pack()` and `nvlist_unpack()`:

- `0` success
- `EINVAL` invalid argument
- `ENOMEM` insufficient memory
- `EFAULT` encode/decode error
- `ENOTSUP` encode/decode method not supported

For `nvlist_size()`:

- `0` success
- `EINVAL` invalid argument

**CONTEXT**

The `nvlist_alloc()`, `nvlist_pack()`, `nvlist_unpack()`, and `nvlist_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.
nvlist_dup(9F)

NAME
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a
name-value pair list

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nflags, int kmflag);
void nvlist_free(nvlist_t *nvl);

int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);

int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);

int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);

int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);

INTERFACE
Solaris DDI specific (Solaris DDI)

PARAMETERS

nvlp Address of a pointer to list of name-value pairs (nvlist_t).
nflags Specify bit fields defining nvlist_t properties:

NV_UNIQUE_NAME
The nvpair names are unique.

NV_UNIQUE_NAME_TYPE
Name-data type combination is unique

kmflag Kernel memory allocation policy, either KM_SLEEP or
KM_NOSLEEP.
nvl The nvlist_t to be processed.
size Pointer to buffer to contain the encoded size.
bufp Address of buffer to pack nvlist into. Must be 8-byte aligned. If
NULL, library will allocate memory.
buf Buffer containing packed nvlist_t.
buflen Size of buffer buf or buf points to.
encoding Encoding method for packing.

DESCRIPTION
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp
to point to the handle. The argument nflags specifies nvlist_t properties to remain
persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large
efficient enough to pack nvl. The encoding parameter specifies the method of encoding when
packing nvl. Supported encoding methods are:
<table>
<thead>
<tr>
<th>NV_ENCODE_NATIVE</th>
<th>Straight <code>bcopy()</code> as described in <code>bcopy(9F)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV_ENCODE_XDR</td>
<td>Use XDR encoding, suitable for sending to another host.</td>
</tr>
</tbody>
</table>

The `nvlist_pack()` function packs `nvl` into contiguous memory starting at `*bufp`. The `encoding` parameter specifies the method of encoding (see above).

- If `*bufp` is not NULL, `*bufp` is expected to be a caller-allocated buffer of size `*buflen`. The `kmflag` argument is ignored.
- If `*bufp` is NULL, the library will allocate memory and update `*bufp` to point to the memory and update `*buflen` to contain the size of the allocated memory. The value of `kmflag` indicates the memory allocation policy.

The `nvlist_unpack()` function takes a buffer with a packed `nvlist_t` and unpacks it into a searchable `nvlist_t`. The library allocates memory for `nvlist_t`. The caller is responsible for freeing the memory by calling `nvlist_free()`.

The `nvlist_dup()` function makes a copy of `nvl` and updates `nvlp` to point to the copy.

**RETURN VALUES**

For `nvlist_alloc()`, `nvlist_dup()`:

- **0** success
- **EINVAL** invalid argument
- **ENOMEM** insufficient memory

For `nvlist_pack()`, `nvlist_unpack()`:

- **0** success
- **EINVAL** invalid argument
- **ENOMEM** insufficient memory
- **EFAULT** encode/decode error
- **ENOTSUP** encode/decode method not supported

For `nvlist_size()`:

- **0** success
- **EINVAL** invalid argument

**CONTEXT**

The `nvlist_alloc()`, `nvlist_pack()`, `nvlist_unpack()`, and `nvlist_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.
nvlist_free(9F)

NAME
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a
name-value pair list

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag, int kmflag);
void nvlist_free(nvlist_t *nvl);
int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);
int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);
int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);
int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);

INTERFACE Solaris DDI specific (Solaris DDI)

PARAMETERS
nvlp Address of a pointer to list of name-value pairs (nvlist_t).
nvflag Specify bit fields defining nvlist_t properties:
   NV_UNIQUE_NAME
       The nvpair names are unique.
   NV_UNIQUE_NAME_TYPE
       Name-data type combination is unique
kmflag Kernel memory allocation policy, either KM_SLEEP or
   KM_NOSLEEP.
nvl The nvlist_t to be processed.
size Pointer to buffer to contain the encoded size.
bufp Address of buffer to pack nvlist into. Must be 8-byte aligned. If
   NULL, library will allocate memory.
buf Buffer containing packed nvlist_t.
buflen Size of buffer bufp or buf points to.
encoding Encoding method for packing.

DESCRIPTION
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp
to point to the handle. The argument nvflag specifies nvlist_t properties to remain
persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large
enough to pack nvl. The encoding parameter specifies the method of encoding when
packing nvl. Supported encoding methods are:
NV_ENCODE_NATIVE  Straight bcopy() as described in bcopy(9F).

NV_ENCODE_XDR    Use XDR encoding, suitable for sending to another host.

The nvlist_pack() function packs nvl into contiguous memory starting at *bufp. The encoding parameter specifies the method of encoding (see above).

- If *bufp is not NULL, *bufp is expected to be a caller-allocated buffer of size *buflen. The kmflag argument is ignored.
- If *bufp is NULL, the library will allocate memory and update *bufp to point to the memory and update *buflen to contain the size of the allocated memory. The value of kmflag indicates the memory allocation policy.

The nvlist_unpack() function takes a buffer with a packed nvlist_t and unpacks it into a searchable nvlist_t. The library allocates memory for nvlist_t. The caller is responsible for freeing the memory by calling nvlist_free().

The nvlist_dup() function makes a copy of nvl and updates nvlp to point to the copy.

RETURN VALUES

For nvlist_alloc(), nvlist_dup():

- 0  success
- EINVAL invalid argument
- ENOMEM insufficient memory

For nvlist_pack(), nvlist_unpack():

- 0  success
- EINVAL invalid argument
- ENOMEM insufficient memory
- EFAULT encode/decode error
- ENOTSUP encode/decode method not supported

For nvlist_size():

- 0  success
- EINVAL invalid argument

CONTEXT

The nvlist_alloc(), nvlist_pack(), nvlist_unpack(), and nvlist_dup() functions can be called from interrupt context only if the KM_NOSLEEP flag is set. They can be called from user context with any valid flag.

Kernel Functions for Drivers  1145
nvlist_lookup_boolean(9F)

NAME nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16, nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64, nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array, nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array, nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array, nvlist_lookup_string_array – match name and type indicated by the interface name and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvl
The list of name-value pairs (nvlist_t) to be processed.

name
Name of the name-value pair (nvpair) to search.
These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_ts` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>invalid argument</td>
</tr>
<tr>
<td>ENOENT</td>
<td>no matching name-value pair found</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>encode/decode method not supported</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt contexts.
nvlist_lookup_byte(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL
PARAMETERS

nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
DESCRIPTION

These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, nelem and val are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist_t allocated with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in nvlist_alloc(). (See nvlist_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist_free() is called on nvl.

RETURN VALUES

0 success
EINVAL invalid argument
ENOENT no matching name-value pair found
ENOTSUP encode/decode method not supported

CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS

#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI)

nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of the array value.

DESCRIPTION These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, nelem and val are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist_ts allocated with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in nvlist_alloc(). (See nvlist_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist_free() is called on nvl.

RETURN VALUES
0 success
EINVAL invalid argument
ENOENT no matching name-value pair found
ENOTSUP encode/decode method not supported

CONTEXT These functions can be called from user or interrupt contexts.
nvlist_lookup_int16(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
DESCRIPTION

These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nlem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

RETURN VALUES

- 0: success
- EINVAL: invalid argument
- ENOENT: no matching name-value pair found
- ENOTSUP: encode/decode method not supported

CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_lookup_int16_array(9F)

NAME nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
    nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32,
    nvlist_lookup_int64, nvlist_lookup_uint64, nvlist_lookup_string,
    nvlist_lookup_byte_array, nvlist_lookup_int16_array, nvlist_lookup_uint16_array,
    nvlist_lookup_int32_array, nvlist_lookup_uint32_array, nvlist_lookup_int64_array,
    nvlist_lookup_uint64_array, nvlist_lookup_string_array – match name and type indicated by the interface name
    and retrieve data value

SYNOPSIS #include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char **val, uint_t *nelem);

INTERFACE

LEVEL Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl The list of name-value pairs (nvlist_t) to be processed.

name Name of the name-value pair (nvpair) to search.
nvlist_lookup_int16_array(9F)

nelem Address to store the number of elements in value.
val Address to store the value or starting address of the array value.

DESCRIPTION These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, nelem and val are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist_t allocations with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in nvlist_alloc(). (See nvlist_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist_free() is called on nvl.

RETURN VALUES

0 success
EINVAL invalid argument
ENOENT no matching name-value pair found
ENOTSUP encode/decode method not supported

CONTEXT These functions can be called from user or interrupt contexts.
nvlist_lookup_int32(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
These functions find the \texttt{nvpair} that matches the name and type as indicated by the interface name. If one is found, \texttt{nelem} and \texttt{val} are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for \texttt{nvlist_t}s allocated with \texttt{NV_UNIQUE_NAME} or \texttt{NV_UNIQUE_NAME_TYPE} specified in \texttt{nvlist_alloc()}. (See \texttt{nvlist_alloc(9F)).} If this is not the case, the interface will return \texttt{ENOTSUP} because the list potentially contains multiple \texttt{nvpairs} with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until \texttt{nvlist_free()} is called on \texttt{nvl}.

**RETURN VALUES**

- \texttt{0} 
  success
- \texttt{EINVAL} 
  invalid argument
- \texttt{ENOENT} 
  no matching name-value pair found
- \texttt{ENOTSUP} 
  encode/decode method not supported

**CONTEXT**

These functions can be called from user or interrupt contexts.
nvlist_lookup_int32_array(9F)

NAME      nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,  
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,  
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,  
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,  
nvlist_lookup_uint32_array, nvlist_lookup_int32_array, nvlist_lookup_uint32_array,  
nvlist_lookup_int64_array, nvlist_lookup_uint64_array, nvlist_lookup_string_array  

INTERFACE

LEVEL     Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl             The list of name-value pairs (nvlist_t) to be processed.

name            Name of the name-value pair (nvpair) to search.

SYNOPSIS

#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);
nvlist_lookup_int32_array(9F)

<table>
<thead>
<tr>
<th>nelem</th>
<th>Address to store the number of elements in value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Address to store the value or starting address of the array value.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>invalid argument</td>
</tr>
<tr>
<td>ENOENT</td>
<td>no matching name-value pair found</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>encode/decode method not supported</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt contexts.
nvlist_lookup_int64(9F)

NAME nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
   nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
   nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
   nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
   nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
   nvlist_lookup_string_array – match name and type indicated by the interface name
   and retrieve data value

SYNOPSIS #include <sys/nvpair.h>

   int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
   int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
   int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
   int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
   int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
   int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
   int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
   int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
   int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
   int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
   int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
   int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
   int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
   int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
   int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
   int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
   int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI)

   nvl The list of name-value pairs (nvlist_t) to be processed.
   name Name of the name-value pair (nvpair) to search.
These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_ts` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

**RETURN VALUES**

- 0: success
- EINVAL: invalid argument
- ENOENT: no matching name-value pair found
- ENOTSUP: encode/decode method not supported

**CONTEXT**

These functions can be called from user or interrupt contexts.
The list of name-value pairs (nvlist_t) to be processed.

Name of the name-value pair (nvpair) to search.
nvlist_lookup_int64_array(9F)

**nelem** Address to store the number of elements in value.

**val** Address to store the value or starting address of the array value.

**DESCRIPTION** These functions find the *nvpair* that matches the name and type as indicated by the interface name. If one is found, **nelem** and **val** are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for *nvlist_t* allocated with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in **nvlist_alloc()**. (See **nvlist_alloc(9F)**.) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple *nvpairs* with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until **nvlist_free()** is called on *nvl*.

**RETURN VALUES**

- **0** success
- **EINVAL** invalid argument
- **ENOENT** no matching name-value pair found
- **ENOTSUP** encode/decode method not supported

**CONTEXT** These functions can be called from user or interrupt contexts.
nvlist_lookup_string(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int32_array, nvlist_lookup_uint32_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char **val, uint_t *nelem);

INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI)

nvl  The list of name-value pairs (nvlist_t) to be processed.
name  Name of the name-value pair (nvpair) to search.
nvlist_lookup_string(9F)

**nelem**
Address to store the number of elements in value.

**val**
Address to store the value or starting address of the array value.

**DESCRIPTION**
These functions find the *nvpair* that matches the name and type as indicated by the interface name. If one is found, *nelem* and *val* are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for *nvlist_ts* allocated with **NV_UNIQUE_NAME** or **NV_UNIQUE_NAME_TYPE** specified in *nvlist_alloc()*. (See *nvlist_alloc(9F)*.) If this is not the case, the interface will return **ENOTSUP** because the list potentially contains multiple *nvpairs* with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until *nvlist_free()* is called on *nvl*.

**RETURN VALUES**

- **0**
  success

- **EINVAL**
  invalid argument

- **ENOENT**
  no matching name-value pair found

- **ENOTSUP**
  encode/decode method not supported

**CONTEXT**
These functions can be called from user or interrupt contexts.
nvlist_lookup_string_array(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16, nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64, nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array, nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array, nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array, nvlist_lookup_string_array

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
DESCRIPTION

These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_ts` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>invalid argument</td>
</tr>
<tr>
<td>ENOENT</td>
<td>no matching name-value pair found</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>encode/decode method not supported</td>
</tr>
</tbody>
</table>

CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_lookup_uint16(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
in
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
in
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
in
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
in
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
in
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl
The list of name-value pairs (nvlist_t) to be processed.

name
Name of the name-value pair (nvpair) to search.
DESCRIPTION

These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvlist`.

RETURN VALUES

- 0: success
- EINVAL: invalid argument
- ENOENT: no matching name-value pair found
- ENOTSUP: encode/decode method not supported

CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_lookup_uint16_array(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl
The list of name-value pairs (nvlist_t) to be processed.

name
Name of the name-value pair (nvpair) to search.
nvlist_lookup_uint16_array(9F)

### Description

These functions find the *nvpair* that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_ts` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

### Return Values

- 0: success
-EINVAL: invalid argument
-ENOENT: no matching name-value pair found
-ENOTSUP: encode/decode method not supported

### Context

These functions can be called from user or interrupt contexts.
nvlist_lookup_uint32(9F)

NAME  nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS  #include <sys/nvpair.h>
int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist *nvl, char *name, uchar_t **val,
uint_t *nelem);
int nvlist_lookup_int16_array(nvlist *nvl, char *name, int16_t **val,
uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist *nvl, char *name, uint16_t **val,
uint_t *nelem);
int nvlist_lookup_int32_array(nvlist *nvl, char *name, int32_t **val,
uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist *nvl, char *name, uint32_t **val,
uint_t *nelem);
int nvlist_lookup_int64_array(nvlist *nvl, char *name, int64_t **val,
uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist *nvl, char *name, uint64_t **val,
uint_t *nelem);
int nvlist_lookup_string_array(nvlist *nvl, char *name, char **val,
uint_t *nelem);

INTERFACE LEVEL PARAMETERS  Solaris DDI specific (Solaris DDI)

nvl  The list of name-value pairs (nvlist_t) to be processed.
name  Name of the name-value pair (nvpair) to search.
DESCRIPTION
These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, nelem and val are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist_ts allocated with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in nvlist_alloc(). (See nvlist_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist_free() is called on nvl.

RETURN VALUES
0 success
EINVAL invalid argument
ENOENT no matching name-value pair found
ENOTSUP encode/decode method not supported

CONTEXT These functions can be called from user or interrupt contexts.
nvlist_lookup_uint32_array(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ****val, uint_t *nelem);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl
The list of name-value pairs (nvlist_t) to be processed.

name
Name of the name-value pair (nvpair) to search.
### nvlist_lookup_uint32_array(9F)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nelem</td>
<td>Address to store the number of elements in value.</td>
</tr>
<tr>
<td>val</td>
<td>Address to store the value or starting address of the array value.</td>
</tr>
</tbody>
</table>

#### DESCRIPTION

These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpair`s with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvl`.

#### RETURN VALUES

- 0: success
- EINVAL: invalid argument
- ENOENT: no matching name-value pair found
- ENOTSUP: encode/decode method not supported

#### CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_lookup_uint64(9F)

NAME
nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val,
                           uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val,
                          uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val,
                        uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val,
                        uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val,
                       uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val,
                        uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val,
                        uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char **val,
                        uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvl The list of name-value pairs (nvlist_t) to be processed.
name Name of the name-value pair (nvpair) to search.
nvlist_lookup_uint64(9F)

nelem Address to store the number of elements in value.
val Address to store the value or starting address of the array value.

DESCRIPTION These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, nelem and val are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist_ts allocated with NV_UNIQUE_NAME or NV_UNIQUE_NAME_TYPE specified in nvlist_alloc(). (See nvlist_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist_free() is called on nvlist.

RETURN VALUES
0 success
EINVAL invalid argument
ENOENT no matching name-value pair found
ENOTSUP encode/decode method not supported

CONTEXT These functions can be called from user or interrupt contexts.
nvlist_lookup_uint64_array(9F)

NAME

nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16,
nvlist_lookup_uint16, nvlist_lookup_int32, nvlist_lookup_uint32, nvlist_lookup_int64,
nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_byte_array,
nvlist_lookup_int16_array, nvlist_lookup_uint16_array, nvlist_lookup_int32_array,
nvlist_lookup_uint32_array, nvlist_lookup_int64_array, nvlist_lookup_uint64_array,
nvlist_lookup_string_array – match name and type indicated by the interface name
and retrieve data value

SYNOPSIS

#include <sys/nvpair.h>

int nvlist_lookup_boolean(nvlist_t *nvl, char *name);
int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val);
int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val);
int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val);
int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val);
int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val);
int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val);
int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val);
int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val);
int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t **val, uint_t *nelem);
int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t **val, uint_t *nelem);
int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t **val, uint_t *nelem);
int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t **val, uint_t *nelem);
int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t **val, uint_t *nelem);
int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t **val, uint_t *nelem);
int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t **val, uint_t *nelem);
int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char ***val, uint_t *nelem);

INTERFACE

Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvl The list of name-value pairs (nvlist_t) to be processed.

name Name of the name-value pair (nvpair) to search.
DESCRIPTION

These functions find the `nvpair` that matches the name and type as indicated by the interface name. If one is found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for `nvlist_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlist_alloc()`. (See `nvlist_alloc(9F)`.) If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlist_free()` is called on `nvlist`.

RETURN VALUES

- 0: success
- EINVAL: invalid argument
- ENOENT: no matching name-value pair found
- ENOTSUP: encode/decode method not supported

CONTEXT

These functions can be called from user or interrupt contexts.
nvlist_next_nvpair(9F)

NAME

nvlist_next_nvpair, nvpair_name, nvpair_type – return data regarding name-value pairs

SYNOPSIS

#include <sys/nvpair.h>

nvpair_t *nvlist_next_nvpair(nvlist_t *nvl, nvpair_t *nvpair);
char *nvpair_name(nvpair_t *nvpair);
data_type_t nvpair_type(nvpair_t *nvpair);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

PARAMETERS

nvl The list of name-value pairs (nvlist_t) to be processed.
nvpair Handle to a name-value pair.

DESCRIPTION

The nvlist_next_nvpair() function returns a handle to the next name-value pair (nvpair) in the list following nvpair. If nvpair is NULL, the first pair is returned. If nvpair is the last pair in the nvlist_t, NULL is returned.

The nvpair_name() function returns a string containing the name of nvpair.

The nvpair_type() function retrieves the value of the nvpair in the form of enumerated type data_type_t. This is used to determine the appropriate nvpair_*() function to call for retrieving the value.

RETURN VALUES

For nvpair_name(), a string containing the name.

For nvpair_type(), an enumerated data type data_type_t. Possible values for data_type_t are as follows:

DATA_TYPE_BOOLEAN
DATA_TYPE_BYTE
DATA_TYPE_INT16
DATA_TYPE_UINT16
DATA_TYPE_INT32
DATA_TYPE_UINT32
DATA_TYPE_INT64
DATA_TYPE_UINT64
DATA_TYPE_STRING
DATA_TYPE_BYTE_ARRAY
DATA_TYPE_INT16_ARRAY
DATA_TYPE_UINT16_ARRAY
DATA_TYPE_INT32_ARRAY
DATA_TYPE_UINT32_ARRAY
DATA_TYPE_INT64_ARRAY
DATA_TYPE_UINT64_ARRAY
DATA_TYPE_STRING_ARRAY

For nvlist_next_pair():

NULL Reached end of list.
otherwise: Handle to next nvpair in the list.
nvlist_next_nvpair(9F)

CONTEXT | The functions described here can be called from user or interrupt context.
nvlist_pack(9F)

NAME
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a name-value pair list

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag, int kmflag);
void nvlist_free(nvlist_t *nvl);

int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);

int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);

int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);

int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS

nvlp Address of a pointer to list of name-value pairs (nvlist_t).

nvflag Specify bit fields defining nvlist_t properties:

NV_UNIQUE_NAME
The nvpair names are unique.

NV_UNIQUE_NAME_TYPE
Name-data type combination is unique

kmflag Kernel memory allocation policy, either KM_SLEEP or KM_NOSLEEP.

nvl The nvlist_t to be processed.

size Pointer to buffer to contain the encoded size.

bufp Address of buffer to pack nvlist into. Must be 8-byte aligned. If NULL, library will allocate memory.

buf Buffer containing packed nvlist_t.

buflen Size of buffer bufp or buf points to.

encoding Encoding method for packing.

DESCRIPTION
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp to point to the handle. The argument nvflag specifies nvlist_t properties to remain persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large enough to pack nvl. The encoding parameter specifies the method of encoding when packing nvl. Supported encoding methods are:
The `nvlist_pack()` function packs `nvl` into contiguous memory starting at `*bufp`. The `encoding` parameter specifies the method of encoding (see above).

- If `*bufp` is not NULL, `*bufp` is expected to be a caller-allocated buffer of size `*buflen`. The `kmflag` argument is ignored.
- If `*bufp` is NULL, the library will allocate memory and update `*bufp` to point to the memory and update `*buflen` to contain the size of the allocated memory. The value of `kmflag` indicates the memory allocation policy.

The `nvlist_unpack()` function takes a buffer with a packed `nvlist_t` and unpacks it into a searchable `nvlist_t`. The library allocates memory for `nvlist_t`. The caller is responsible for freeing the memory by calling `nvlist_free()`.

The `nvlist_dup()` function makes a copy of `nvl` and updates `nvlp` to point to the copy.

**RETURN VALUES**

For `nvlist_alloc()`, `nvlist_dup()`:
- 0 success
- EINVAL invalid argument
- ENOMEM insufficient memory

For `nvlist_pack()`, `nvlist_unpack()`:
- 0 success
- EINVAL invalid argument
- ENOMEM insufficient memory
- EFAULT encode/decode error
- ENOTSUP encode/decode method not supported

For `nvlist_size()`:
- 0 success
- EINVAL invalid argument

**CONTEXT**

The `nvlist_alloc()`, `nvlist_pack()`, `nvlist_unpack()`, and `nvlist_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.
nvlist_remove(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>nvlist_remove, nvlist_remove_all – remove name-value pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/nvpair.h&gt;</td>
</tr>
<tr>
<td></td>
<td>void nvlist_remove(nvlist_t *nvl, char *name, data_type_t type);</td>
</tr>
<tr>
<td></td>
<td>void nvlist_remove_all(nvlist_t *nvl, char *name);</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Solaris DDI specific (Solaris DDI)</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>nvl The list of name-value pairs (nvlist_t) to be processed.</td>
</tr>
<tr>
<td></td>
<td>name Name of the name-value pair (nvpair) to be removed.</td>
</tr>
<tr>
<td></td>
<td>type Data type of the nvpair to be removed.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>The nvlist_remove() function removes the first occurrence of nvpair that matches the name and the type.</td>
</tr>
<tr>
<td></td>
<td>The nvlist_remove_all() function removes all occurrences of nvpair that match the name, regardless of type.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>None</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>The nvlist_remove() and nvlist_remove_all() functions can be called from user or interrupt context.</td>
</tr>
</tbody>
</table>
The `nvlist_remove()` function removes the first occurrence of `nvpair` that matches the name and the type.

The `nvlist_remove_all()` function removes all occurrences of `nvpair` that match the name, regardless of type.

None

The `nvlist_remove()` and `nvlist_remove_all()` functions can be called from user or interrupt context.
**NAME**
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a name-value pair list.

**SYNOPSIS**

```c
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag, int kmflag);
void nvlist_free(nvlist_t *nvl);
int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);
int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);
int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);
int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI)

**PARAMETERS**

table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvlp</td>
<td>Address of a pointer to list of name-value pairs (nvlist_t).</td>
</tr>
<tr>
<td>nvflag</td>
<td>Specify bit fields defining nvlist_t properties:</td>
</tr>
<tr>
<td></td>
<td>NV_UNIQUE_NAME</td>
</tr>
<tr>
<td></td>
<td>The nvpair names are unique.</td>
</tr>
<tr>
<td></td>
<td>NV_UNIQUE_NAME_TYPE</td>
</tr>
<tr>
<td></td>
<td>Name-data type combination is unique</td>
</tr>
<tr>
<td>kmflag</td>
<td>Kernel memory allocation policy, either KM_SLEEP or KM_NOSLEEP.</td>
</tr>
<tr>
<td>nvl</td>
<td>The nvlist_t to be processed.</td>
</tr>
<tr>
<td>size</td>
<td>Pointer to buffer to contain the encoded size.</td>
</tr>
<tr>
<td>bufp</td>
<td>Address of buffer to pack nvlist into. Must be 8-byte aligned. If NULL, library will allocate memory.</td>
</tr>
<tr>
<td>buf</td>
<td>Buffer containing packed nvlist_t.</td>
</tr>
<tr>
<td>buflen</td>
<td>Size of buffer bufp or buf points to.</td>
</tr>
<tr>
<td>encoding</td>
<td>Encoding method for packing.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp to point to the handle. The argument nvflag specifies nvlist_t properties to remain persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large enough to pack nvl. The encoding parameter specifies the method of encoding when packing nvl. Supported encoding methods are:
The `nvlist_pack()` function packs `nvl` into contiguous memory starting at `*bufp`. The `encoding` parameter specifies the method of encoding (see above).
- If `*bufp` is not NULL, `*bufp` is expected to be a caller_allocated buffer of size `*buflen`. The `kmflag` argument is ignored.
- If `*bufp` is NULL, the library will allocate memory and update `*bufp` to point to the memory and update `*buflen` to contain the size of the allocated memory. The value of `kmflag` indicates the memory allocation policy.

The `nvlist_unpack()` function takes a buffer with a packed `nvlist_t` and unpacks it into a searchable `nvlist_t`. The library allocates memory for `nvlist_t`. The caller is responsible for freeing the memory by calling `nvlist_free()`.

The `nvlist_dup()` function makes a copy of `nvl` and updates `nvlp` to point to the copy.

RETURN VALUES
For `nvlist_alloc()`, `nvlist_dup()`:
- 0: success
- EINVAL: invalid argument
- ENOMEM: insufficient memory
For `nvlist_pack()`, `nvlist_unpack()`:
- 0: success
- EINVAL: invalid argument
- ENOMEM: insufficient memory
- EFAULT: encode/decode error
- ENOTSUP: encode/decode method not supported
For `nvlist_size()`:
- 0: success
- EINVAL: invalid argument

CONTEXT
The `nvlist_alloc()`, `nvlist_pack()`, `nvlist_unpack()`, and `nvlist_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.
nvlist_alloc(9F)

NAME
nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a name-value pair list

SYNOPSIS
#include <sys/nvpair.h>

int nvlist_alloc(nvlist_t **nvlp, uint_t nvlflag, int kmflag);
void nvlist_free(nvlist_t *nvl);
int nvlist_size(nvlist_t *nvl, size_t *size, int encoding);
int nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding, int kmflag);
int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int kmflag);
int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int kmflag);

INTERFACE
Solaris DDI specific (Solaris DDI)

PARAMETERS
nvlp Address of a pointer to list of name-value pairs (nvlist_t).
nvflag Specify bit fields defining nvlist_t properties:
   NV_UNIQUE_NAME
      The nvpair names are unique.
   NV_UNIQUE_NAME_TYPE
      Name-data type combination is unique
kmflag Kernel memory allocation policy, either KM_SLEEP or KM_NOSLEEP.
nvl The nvlist_t to be processed.
size Pointer to buffer to contain the encoded size.
bufp Address of buffer to pack nvlist into. Must be 8-byte aligned. If NULL, library will allocate memory.
buf Buffer containing packed nvlist_t.
 buflen Size of buffer bufp or buf points to.
 encoding Encoding method for packing.

DESCRIPTION
The nvlist_alloc() function allocates a new name-value pair list and updates nvlp to point to the handle. The argument nvflag specifies nvlist_t properties to remain persistent across packing, unpacking, and duplication.

The nvlist_free() function frees a name-value pair list.

The nvlist_size() function returns the minimum size of a contiguous buffer large enough to pack nvl. The encoding parameter specifies the method of encoding when packing nvl. Supported encoding methods are:
nvlist_unpack(9F)

<table>
<thead>
<tr>
<th>NV_ENCODE_NATIVE</th>
<th>Straight bcopy() as described in bcopy(9F).</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV_ENCODE_XDR</td>
<td>Use XDR encoding, suitable for sending to another host.</td>
</tr>
</tbody>
</table>

The nvlist_pack() function packs nvl into contiguous memory starting at *bufp. The encoding parameter specifies the method of encoding (see above).

- If *bufp is not NULL, *bufp is expected to be a caller-allocated buffer of size *buflen. The kmflag argument is ignored.
- If *bufp is NULL, the library will allocate memory and update *bufp to point to the memory and update *buflen to contain the size of the allocated memory. The value of kmflag indicates the memory allocation policy.

The nvlist_unpack() function takes a buffer with a packed nvlist_t and unpacks it into a searchable nvlist_t. The library allocates memory for nvlist_t. The caller is responsible for freeing the memory by calling nvlist_free().

The nvlist_dup() function makes a copy of nvl and updates nvlp to point to the copy.

**RETURN VALUES**

For nvlist_alloc(), nvlist_dup():

- 0        success
- EINVAL   invalid argument
- ENOMEM   insufficient memory

For nvlist_pack(), nvlist_unpack():

- 0        success
- EINVAL   invalid argument
- ENOMEM   insufficient memory
- EFAULT   encode/decode error
- ENOTSUP   encode/decode method not supported

For nvlist_size():

- 0        success
- EINVAL   invalid argument

**CONTEXT**

The nvlist_alloc(), nvlist_pack(), nvlist_unpack(), and nvlist_dup() functions can be called from interrupt context only if the KM_NOSLEEP flag is set. They can be called from user context with any valid flag.
nvpair_name(9F)

NAME
nvlist_next_nvpair, nvpair_name, nvpair_type – return data regarding name-value pairs

SYNOPSIS
#include <sys/nvpair.h>

nvpair_t *nvlist_next_nvpair(nvlist_t *nvl, nvpair_t *nvpair);
char *nvpair_name(nvpair_t *nvpair);
data_type_t nvpair_type(nvpair_t *nvpair);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
nvl The list of name-value pairs (nvlist_t) to be processed.
nvpair Handle to a name-value pair.

DESCRIPTION
The nvlist_next_nvpair() function returns a handle to the next name-value pair
(nvpair) in the list following nvpair. If nvpair is NULL, the first pair is returned. If
nvpair is the last pair in the nvlist_t, NULL is returned.

The nvpair_name() function returns a string containing the name of nvpair.

The nvpair_type() function retrieves the value of the nvpair in the form of
enumerated type data_type_t. This is used to determine the appropriate
nvpair_*() function to call for retrieving the value.

RETURN VALUES
For nvpair_name(), a string containing the name.

For nvpair_type(), an enumerated data type data_type_t. Possible values for
data_type_t are as follows:

DATA_TYPE_BOOLEAN
DATA_TYPE_BYTE
DATA_TYPE_INT16
DATA_TYPE_UINT16
DATA_TYPE_INT32
DATA_TYPE_UINT32
DATA_TYPE_INT64
DATA_TYPE_UINT64
DATA_TYPE_STRING
DATA_TYPE_BYTE_ARRAY
DATA_TYPE_INT16_ARRAY
DATA_TYPE_UINT16_ARRAY
DATA_TYPE_INT32_ARRAY
DATA_TYPE_UINT32_ARRAY
DATA_TYPE_INT64_ARRAY
DATA_TYPE_UINT64_ARRAY
DATA_TYPE_STRING_ARRAY

For nvlist_next_pair():

NULL Reached end of list.
otherwise: Handle to next nvpair in the list.
| CONTEXT | The functions described here can be called from user or interrupt context. |
NAME
nvlist_next_nvpair, nvpair_name, nvpair_type – return data regarding name-value pairs

SYNOPSIS
#include <sys/nvpair.h>

nvpair_t *nvlist_next_nvpair(nvlist_t *nvl, nvpair_t *nvpair);
char *nvpair_name(nvpair_t *nvpair);
data_type_t nvpair_type(nvpair_t *nvpair);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
nvl The list of name-value pairs (nvlist_t) to be processed.
nvpair Handle to a name-value pair.

DESCRIPTION
The nvlist_next_nvpair() function returns a handle to the next name-value pair (nvpair) in the list following nvpair. If nvpair is NULL, the first pair is returned. If nvpair is the last pair in the nvlist_t, NULL is returned.

The nvpair_name() function returns a string containing the name of nvpair.

The nvpair_type() function retrieves the value of the nvpair in the form of enumerated type data_type_t. This is used to determine the appropriate nvpair_*() function to call for retrieving the value.

RETURN VALUES
For nvpair_name(), a string containing the name.

For nvpair_type(), an enumerated data type data_type_t. Possible values for data_type_t are as follows:

DATA_TYPE_BOOLEAN
DATA_TYPE_BYTE
DATA_TYPE_INT16
DATA_TYPE_UINT16
DATA_TYPE_INT32
DATA_TYPE_UINT32
DATA_TYPE_INT64
DATA_TYPE_UINT64
DATA_TYPE_STRING
DATA_TYPE_BYTE_ARRAY
DATA_TYPE_INT16_ARRAY
DATA_TYPE_UINT16_ARRAY
DATA_TYPE_INT32_ARRAY
DATA_TYPE_UINT32_ARRAY
DATA_TYPE_INT64_ARRAY
DATA_TYPE_UINT64_ARRAY
DATA_TYPE_STRING_ARRAY

For nvlist_next_pair():

NULL Reached end of list.

otherwise: Handle to next nvpair in the list.
The functions described here can be called from user or interrupt context.
NAME
	nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
	nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
	nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
	nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
	nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
	name-value pair

SYNOPSIS

#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE

LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI)


"""nvpair"" Name-value pair (nvpair) to be processed.

nelem Address to store the number of elements in value.

val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

**RETURN VALUES**

- **0**  
  Success

- **EINVAL**  
  Either one of the arguments is NULL or type of `nvpair` does not match the interface name.

**CONTEXT**

These functions can be called from user or interrupt context.

desc
NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair.

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE LEVEL
PARAMETERS
Solaris DDI specific (Solaris DDI)
nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context.
NAME

nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair.

SYNOPSIS

#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI)

nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

**RETURN VALUES**

- **0**: Success
- **EINVAL**: Either one of the arguments is NULL or type of `nvpair` does not match the interface name.

**CONTEXT**

These functions can be called from user or interrupt context.
nvpair_value_int16_array(9F)

NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`, the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

These functions can be called from user or interrupt context.
### NAME

nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair

### SYNOPSIS

```c
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

### PARAMETERS

- **nvpair**: Name-value pair (nvpair) to be processed.
- **nelem**: Address to store the number of elements in value.
- **val**: Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

**RETURN VALUES**

- **0**: Success
- **EINVAL**: Either one of the arguments is NULL or type of `nvpair` does not match the interface name.

**CONTEXT**

These functions can be called from user or interrupt context.
nvpair_value_int32_array(9F)

NAME nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair

SYNOPSIS #include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val,
uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val,
uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val,
uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val,
uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val,
uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val,
uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t
*nelem);

INTERFACE LEVEL PARAMETERS Solaris DDI specific (Solaris DDI)
nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

These functions can be called from user or interrupt context.
NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);

int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

### RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

### CONTEXT

These functions can be called from user or interrupt context.
NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context.
nvpair_value_string(9F)

NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS
nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.

1210 man pages section 9: DDI and DKI Kernel Functions • Last Revised 27 September 2000
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

These functions can be called from user or interrupt context.
NAME
nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair.

SYNOPSIS
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE
Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvpair
Name-value pair (nvpair) to be processed.

nelem
Address to store the number of elements in value.

val
Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean();` the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free(9)` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9)`

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>These functions can be called from user or interrupt context.</td>
<td></td>
</tr>
</tbody>
</table>
**NAME**

nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_string, nvpair_value_string_array – retrieve value from a name-value pair.

**SYNOPSIS**

```c
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char **val, uint_t *nelem);
```

**PARAMETERS**

- **nvpair**: Name-value pair (nvpair) to be processed.
- **nelem**: Address to store the number of elements in value.
- **val**: Address to store the value or starting address of array value.

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI)
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

### RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

### CONTEXT

These functions can be called from user or interrupt context.
**NAME**

nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array

- retrieve value from a name-value pair

**SYNOPSIS**

```c
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);
```

**ADDRESS**

- nvpair: Name-value pair (nvpair) to be processed.
- nelem: Address to store the number of elements in value.
- val: Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

### RETURN VALUES

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

### CONTEXT

These functions can be called from user or interrupt context.
NAME  nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array

SYNOPSIS

#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE

LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI)
nvpair  Name-value pair (nvpair) to be processed.
nelem   Address to store the number of elements in value.
val     Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean();` the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0</code></td>
<td>Success</td>
</tr>
<tr>
<td><code>EINVAL</code></td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

These functions can be called from user or interrupt context.
nvpair_value_uint32_array(9F)

NAME    nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair

SYNOPSIS    

#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE LEVEL
PARAMETERS

Solaris DDI specific (Solaris DDI)

nvpair    Name-value pair (nvpair) to be processed.

nelem    Address to store the number of elements in value.

val    Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean();` the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context.
#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

NAME  nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair

SYNOPSIS

INTERFACE

LEVEL

PARAMETERS

Solaris DDI specific (Solaris DDI)

nvpair    Name-value pair (nvpair) to be processed.
nelem     Address to store the number of elements in value.
val       Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

These functions can be called from user or interrupt context.
NAME

nvpair_value_byte, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32,
nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string,
nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array,
nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array,
nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a
name-value pair

SYNOPSIS

#include <sys/nvpair.h>

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val, uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val, uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val, uint_t *nelem);
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val, uint_t *nelem);
int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val, uint_t *nelem);
int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val, uint_t *nelem);
int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t *nelem);

INTERFACE

Solaris DDI specific (Solaris DDI)

LEVEL

PARAMETERS

nvpair Name-value pair (nvpair) to be processed.
nelem Address to store the number of elements in value.
val Address to store the value or starting address of array value.
These functions retrieve the value of `nvpair`. The data type of `nvpair` must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which `nvpair` is obtained. See `nvlist_free(9F)`.

### RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either one of the arguments is NULL or type of <code>nvpair</code> does not match the interface name.</td>
</tr>
</tbody>
</table>

### CONTEXT

These functions can be called from user or interrupt context.
OTHERQ(9F)

NAME | OTHERQ, otherq – get pointer to queue’s partner queue

SYNOPSIS

```c
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *OTHERQ(queue_t *q);
```

INTERFACE LEVEL

ARCHITECTURE INDEPENDENT LEVEL 1 (DDI/DKI).

PARAMETERS

`q` Pointer to the queue.

DESCRIPTION

The `OTHERQ()` function returns a pointer to the other of the two queue structures that make up a STREAMS module or driver. If `q` points to the read queue the write queue will be returned, and vice versa.

RETURN VALUES

`OTHERQ()` returns a pointer to a queue’s partner.

CONTEXT

`OTHERQ()` can be called from user or interrupt context.

EXAMPLES

**EXAMPLE 1 Setting Queues**

This routine sets the minimum packet size, the maximum packet size, the high water mark, and the low water mark for the read and write queues of a given module or driver. It is passed either one of the queues. This could be used if a module or driver wished to update its queue parameters dynamically.

```c
void set_q_params(q, min, max, hi, lo)
queue_t *q;
short min;
short max;
ushort_t hi;
ushort_t lo;
{
    q->q_minpsz = min;
    q->q_maxpsz = max;
    q->q_hiwat = hi;
    q->q_lowat = lo;
    OTHERQ(q)->q_minpsz = min;
    OTHERQ(q)->q_maxpsz = max;
    OTHERQ(q)->q_hiwat = hi;
    OTHERQ(q)->q_lowat = lo;
}
```

SEE ALSO

*Writing Device Drivers*

*STREAMS Programming Guide*
otherq(9F)

NAME
OTHERQ, otherq – get pointer to queue’s partner queue

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *OTHERQ(queue_t *q);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
q Pointer to the queue.

DESCRIPTION
The OTHERQ() function returns a pointer to the other of the two queue structures that make up a STREAMS module or driver. If q points to the read queue the write queue will be returned, and vice versa.

RETURN VALUES
OTHERQ() returns a pointer to a queue’s partner.

CONTEXT
OTHERQ() can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1 Setting Queues

This routine sets the minimum packet size, the maximum packet size, the high water mark, and the low water mark for the read and write queues of a given module or driver. It is passed either one of the queues. This could be used if a module or driver wished to update its queue parameters dynamically.

1  void
2  set_q_params(q, min, max, hi, lo)
3  queue_t *q;
4  short min;
5  short max;
6  ushort_t hi;
7  ushort_t lo;
8  {
9      q->q_minpsz = min;
10     q->q_maxpsz = max;
11     q->q_hiwat = hi;
12     q->q_lowat = lo;
13     OTHERQ(q)->q_minpsz = min;
14     OTHERQ(q)->q_maxpsz = max;
15     OTHERQ(q)->q_hiwat = hi;
16     OTHERQ(q)->q_lowat = lo;
17  }

SEE ALSO
Writing Device Drivers

STREAMS Programming Guide
NAME  | outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

SYNOPSIS  | #include <sys/ddi.h>
          #include <sys/sunddi.h>

          void outb(int port, unsigned char value);
          void outw(int port, unsigned short value);
          void outl(int port, unsigned long value);
          void repoutsb(int port, unsigned char *addr, int count);
          void repoutsw(int port, unsigned short *addr, int count);
          void repoutsd(int port, unsigned long *addr, int count);

INTERFACE LEVEL  | The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsd(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.

PARAMETERS  | port  | A valid I/O port address.
          | value | The data to be written to the I/O port.
          | addr  | The address of a buffer from which the values will be fetched.
          | count | The number of values to be written.

DESCRIPTION  | These routines write data of various sizes to the I/O port with the address specified by port.

          | The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.
          | The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

CONTEXT  | These functions may be called from user or interrupt context.

ATTRIBUTES  | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO  | eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)
Writing Device Drivers

outb(9F)
NAME
outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);

INTERFACE LEVEL
The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsd(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.

PARAMETERS
port A valid I/O port address.
value The data to be written to the I/O port.
addr The address of a buffer from which the values will be fetched.
count The number of values to be written.

DESCRIPTION
These routines write data of various sizes to the I/O port with the address specified by port.

The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.

The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

CONTEXT
These functions may be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)
Writing Device Drivers
outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

#include <sys/ddi.h>
#include <sys/sunddi.h>

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);

The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsd(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.

PARAMETERS
port A valid I/O port address.
value The data to be written to the I/O port.
addr The address of a buffer from which the values will be fetched.
count The number of values to be written.

DESCRIPTION
These routines write data of various sizes to the I/O port with the address specified by port.

The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.

The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

CONTEXT
These functions may be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)

Writing Device Drivers

Kernel Functions for Drivers  1231
pci_config_get16(9F)

NAME  pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64, 
       pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64, 
       pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb, 
       pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of 
       various sizes to the PCI Local Bus Configuration space

SYNOPSIS  
           #include <sys/ddi.h>
           #include <sys/sunddi.h>

           uint8_t  pci_config_get8(ddi_acc_handle_t handle, off_t offset);
           uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
           uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
           uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);

           void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
           void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
           void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
           void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE  
LEVEL      Solaris DDI specific (Solaris DDI).
PARAMETERS  

handle      The data access handle returned from pci_config_setup(9F).
offset      Byte offset from the beginning of the PCI Configuration space.
value       Output data.

DESCRIPTION  
These routines read or write a single datum of various sizes from or to the PCI Local 
Bus Configuration space. The pci_config_get8(), pci_config_get16(), 
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32 
bits, and 64 bits of data, respectively. The pci_config_put8(), 
pci_config_put16(), pci_config_put32(), and pci_config_put64() 
functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset 
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data 
format, these functions translate the data from or to native host format to or from little 
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES  
pci_config_get8(), pci_config_get16(), pci_config_get32(), and 
pci_config_get64() return the value read from the PCI Local Bus Configuration 
space.
These routines can be called from user, kernel, or interrupt context.

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO `attributes(5), pci_config_setup(9F), pci_config_teardown(9F)`

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space.

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
 endian format.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

**SEE ALSO**
attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

**NOTES**
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME

pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.

1240 man pages section 9: DDI and DKI Kernel Functions • Last Revised 1 Jan 1997
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME  pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
       pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
       pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw,
       pci_config_putb, pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
       various sizes to the PCI Local Bus Configuration space

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE  Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS

handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space.

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
 pci_config_getw(9F)

NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space.

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO `attributes(5), pci_config_setup(9F), pci_config_teardown(9F)`

NOTES

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pci_config_getb</code></td>
<td><code>pci_config_get8</code></td>
</tr>
<tr>
<td><code>pci_config_getw</code></td>
<td><code>pci_config_get16</code></td>
</tr>
<tr>
<td><code>pci_config_getl</code></td>
<td><code>pci_config_get32</code></td>
</tr>
<tr>
<td><code>pci_config_getll</code></td>
<td><code>pci_config_get64</code></td>
</tr>
<tr>
<td><code>pci_config_putb</code></td>
<td><code>pci_config_put8</code></td>
</tr>
<tr>
<td><code>pci_config_putw</code></td>
<td><code>pci_config_put16</code></td>
</tr>
<tr>
<td><code>pci_config_putl</code></td>
<td><code>pci_config_put32</code></td>
</tr>
<tr>
<td><code>pci_config_putll</code></td>
<td><code>pci_config_put64</code></td>
</tr>
</tbody>
</table>

pci_config_getw(9F)
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).
PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

**NOTES**

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
pci_config_put64(9F)

NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME  pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
       pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
       pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
       pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
       various sizes to the PCI Local Bus Configuration space

SYNOPSIS  #include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t  pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void     pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void     pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void     pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void     pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
handle  The data access handle returned from pci_config_setup(9F).
offset  Byte offset from the beginning of the PCI Configuration space.
value   Output data.

DESCRIPTION  These routines read or write a single datum of various sizes from or to the PCI Local
       Bus Configuration space. The pci_config_get8(),pci_config_get16(),
       pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
       bits, and 64 bits of data, respectively. The pci_config_put8(),
       pci_config_put16(),pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
       argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES  pci_config_get8(),pci_config_get16(),pci_config_get32(), and
       pci_config_get64() return the value read from the PCI Local Bus Configuration
       space.
These routines can be called from user, kernel, or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

**NOTES**

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
pci_con

NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
 PCI_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t  pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void   pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void   pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void   pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void   pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
handle  The data access handle returned from pci_config_setup(9F).
offset  Byte offset from the beginning of the PCI Configuration space.
value   Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
pci_config_put8(9F)

NAME  pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,  
      pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,  
      pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,  
      pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of  
      various sizes to the PCI Local Bus Configuration space

SYNOPSIS  #include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t  pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);

void  pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void  pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void  pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void  pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE  Solaris DDI specific (Solaris DDI).
LEVEL  PARAMETERS
PARAMETERS
handle  The data access handle returned from pci_config_setup(9F).
offset  Byte offset from the beginning of the PCI Configuration space.
value  Output data.

DESCRIPTION  These routines read or write a single datum of various sizes from or to the PCI Local  
Bus Configuration space. The pci_config_get8(), pci_config_get16(),  
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32  
bits, and 64 bits of data, respectively. The pci_config_put8(),  
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset  
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data  
format, these functions translate the data from or to native host format to or from little  
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES  pci_config_get8(), pci_config_get16(), pci_config_get32(), and  
pci_config_get64() return the value read from the PCI Local Bus Configuration  
space.
These routines can be called from user, kernel, or interrupt context.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

**SEE ALSO**

attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

**NOTES**

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
pci_config_putll(9F)

NAME  pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,  
      pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,  
      pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,  
      pci_config_putl, pci_config_putll, pci_config_putw — read or write single datum of  
      various sizes to the PCI Local Bus Configuration space

SYNOPSIS  
#include <sys/ddi.h>  
#include <sys/sunddi.h>  
uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);  
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);  
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);  
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);  
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);  
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);  
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);  
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE  
LEVEL  Solaris DDI specific (Solaris DDI).
PARAMETERS  
handle  The data access handle returned from pci_config_setup(9F).
offset  Byte offset from the beginning of the PCI Configuration space.
value  Output data.

DESCRIPTION  
These routines read or write a single datum of various sizes from or to the PCI Local  
Bus Configuration space. The pci_config_get8(), pci_config_get16(),  
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32  
bits, and 64 bits of data, respectively. The pci_config_put8(),  
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset  
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data  
format, these functions translate the data from or to native host format to or from little  
endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES  
pci_config_get8(), pci_config_get16(), pci_config_get32(), and  
pci_config_get64() return the value read from the PCI Local Bus Configuration  
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64,
pci_config_put8, pci_config_put16, pci_config_put32, pci_config_put64,
pci_config_getb, pci_config_getl, pci_config_getll, pci_config_getw, pci_config_putb,
pci_config_putl, pci_config_putll, pci_config_putw – read or write single datum of
various sizes to the PCI Local Bus Configuration space

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);

void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local
Bus Configuration space. The pci_config_get8(), pci_config_get16(),
pci_config_get32(), and pci_config_get64() functions read 8 bits, 16 bits, 32
bits, and 64 bits of data, respectively. The pci_config_put8(),
pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset
argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data
format, these functions translate the data from or to native host format to or from little
 endian format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() return the value read from the PCI Local Bus Configuration
space.
These routines can be called from user, kernel, or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME    pci_config_setup, pci_config_teardown – setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space

SYNOPSIS  

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle);
void pci_config_teardown(ddi_acc_handle_t *handle);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

- `dip` Pointer to the device’s dev_info structure.
- `handle` Pointer to a data access handle.

DESCRIPTION

- `pci_config_setup()` sets up the necessary resources for enabling subsequent data accesses to the PCI Local Bus Configuration space. `pci_config_teardown()` reclaims and removes those resources represented by the data access handle returned from `pci_config_setup()`.

RETURN VALUES

- `pci_config_setup()` returns:
  - `DDI_SUCCESS` Successfully setup the resources.
  - `DDI_FAILURE` Unable to allocate resources for setup.

CONTEXT

- `pci_config_setup()` must be called from user or kernel context.
- `pci_config_teardown()` can be called from any context.

NOTES

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

ATTRIBUTES

See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO

`attributes(5)`

IEEE 1275 PCI Bus Binding
pci_confg_setup, pci_confg_teardown – setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle);
void pci_config_teardown(ddi_acc_handle_t *handle);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**

- `dip` Pointer to the device’s dev_info structure.
- `handle` Pointer to a data access handle.

**DESCRIPTION**
`pci_config_setup()` sets up the necessary resources for enabling subsequent data accesses to the PCI Local Bus Configuration space. `pci_config_teardown()` reclaims and removes those resources represented by the data access handle returned from `pci_config_setup()`.

**RETURN VALUES**
`pci_config_setup()` returns:
- `DDI_SUCCESS` Successfully setup the resources.
- `DDI_FAILURE` Unable to allocate resources for setup.

**CONTEXT**
`pci_config_setup()` must be called from user or kernel context.
`pci_config_teardown()` can be called from any context.

**NOTES**
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

**SEE ALSO**

- attributes(5)

*IEEE 1275 PCI Bus Binding*
pci_report_pmcap(9F)

NAME  pci_report_pmcap – Report Power Management capability of a PCI device

SYNOPSIS  #include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_report_pmcap(dev_info_t *dip, int cap, void *arg);

INTERFACE LEVEL
PARAMETERS  Solaris DDI specific (Solaris DDI)

dip Pointer to the device’s dev_info structure
cap Power management capability
arg Argument for the capability

DESCRIPTION  Some PCI devices provide power management capabilities in addition to those provided by the PCI Power Management Specification. The pci_report_pmcap(9F) function reports those Power Management capabilities of the PCI device to the framework. Framework supports dynamic changing of the capability by allowing pci_report_pmcap(9F) to be called multiple times. Following are the supported capabilities as indicated by the cap:

PCI_PM_IDLESPEED — The PCI_PM_IDLESPEED value indicates the lowest PCI clock speed that a device can tolerate when idle, and is applicable only to 33 MHz PCI bus. arg represents the lowest possible idle speed in KHz (1 KHz is 1000 Hz). The integer value representing the speed should be cast to (void *) before passing as arg to pci_report_pmcap(9F).

The special values of arg are:

PCI_PM_IDLESPEED_ANY The device can tolerate any idle clock speed.
PCI_PM_IDLESPEED_NONE The device cannot tolerate slowing down of PCI clock even when idle.

If the driver doesn’t make this call, PCI_PM_IDLESPEED_NONE is assumed. In this case, one offending device can keep the entire bus from being power managed.

RETURN VALUES  The pci_report_pmcap(9F) function returns:

DDI_SUCCESS Successful reporting of the capability
DDI_FAILURE Failure to report capability because of invalid argument(s)

CONTEXT  The pci_report_pmcap(9F) function can be called from user, kernel and interrupt context.

EXAMPLES  1. A device driver knows that the device it controls works with any clock between DC and 33 MHz as specified in Section 4.2.3.1: Clock Specification of the PCI Bus Specification Revision 2.1. The device driver makes the following call from its attach(9E):

```c
if (pci_report_pmcap(dip, PCI_PM_IDLESPEED, PCI_PM_IDLESPEED_ANY) !=
    DDI_SUCCESS)
    cmn_err(CE_WARN, "%s%d: pci_report_pmcap failed\n",
```
2. A device driver controls a 10/100 Mb Ethernet device which runs the device state machine on the chip from the PCI clock. For the device state machine to receive packets at 100 Mb, the PCI clock cannot drop below 4 MHz. The driver makes the following call whenever it negotiates a 100 Mb Ethernet connection:

```c
if (pci_report_pmcap(dip, PCI_PM_IDLESPEED, (void *)4000) != DDI_SUCCESS)
    cmn_err(CE_WARN, "%s%d: pci_report_pmcap failed\n",
            ddi_driver_name(dip), ddi_get_instance(dip));
```

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

**SEE ALSO**

*Writing Device Drivers*

*PCI Bus Power Management Interface Specification Version 1.1*

*PCI Bus Specification Revision 2.1*
pci_restore_config_regs(9F)

| NAME | pci_save_config_regs, pci_restore_config_regs – save and restore the PCI configuration registers |
| SYNOPSIS | #include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_save_config_regs(dev_info_t *dip);
int pci_restore_config_regs(dev_info_t *dip);
| INTERFACE LEVEL | Solaris DDI-specific (Solaris DDI). |
| ARGUMENTS | dip |
| DESCRIPTION | pci_save_config_regs() saves the current configuration registers on persistent system memory. pci_restore_config_regs() restores configuration registers previously saved by pci_save_config_regs().

pci_save_config_regs() should be called by the driver's power() entry point before powering a device off (to PCI state D3). Likewise, pci_restore_config_regs() should be called after powering a device on (from PCI state D3), but before accessing the device. See power(9E).
| RETURN VALUES | pci_save_config_regs() and pci_restore_config_regs() return:
| DDI_SUCCESS | Operation completed successfully. |
| DDI_FAILURE | Operation failed to complete successfully. |
| CONTEXT | Both these functions can be called from user or kernel context. |
| EXAMPLES | EXAMPLE 1 Invoking the save and restore functions

static int
xx_power(dev_info_t *dip, int component, int level) {
    struct xx *xx;
    int rval = DDI_SUCCESS;
    xx = ddi_get_soft_state(xx_softstate, ddi_get_instance(dip));
    if (xx == NULL) {
        return (DDI_FAILURE);
    }
    mutex_enter(&xx->x_mutex);
    switch (level) {
    case PM_LEVEL_D0:
        XX_POWER_ON(xx);
        if (pci_restore_config_regs(dip) == DDI_FAILURE) {
            /*
             * appropriate error path handling here
             */
        }
    }
}
EXAMPLE 1 Invoking the save and restore functions

```c
rval = DDI_FAILURE;
}
break;

case PM_LEVEL_D1:
    if (pci_save_config_regs(dip) == DDI_FAILURE) {
        /*
         * appropriate error path handling here
         */
        ...
        rval = DDI_FAILURE;
    } else {
        XX_POWER_OFF(xx);
    }
    break;

default:
    rval = DDI_FAILURE;
    break;

mutex_exit(&xx->x_mutex);
return (rval);
```

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), power(9E)

Writing Device Drivers

PCI Bus Power Management Interface Specification Version 1.1

PCI Bus Specification Revision 2.1
pci_save_config_regs(9F)

NAME  pci_save_config_regs, pci_restore_config_regs – save and restore the PCI configuration registers

SYNOPSIS  #include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_save_config_regs(dev_info_t *dip);
int pci_restore_config_regs(dev_info_t *dip);

INTERFACE  Solaris DDI-specific (Solaris DDI).
LEVEL
ARGUMENTS  *dip  Pointer to the device’s dev_info structure.

DESCRIPTION  pci_save_config_regs() saves the current configuration registers on persistent system memory. pci_restore_config_regs() restores configuration registers previously saved by pci_save_config_regs().

pci_save_config_regs() should be called by the driver’s power() entry point before powering a device off (to PCI state D3). Likewise, pci_restore_config_regs() should be called after powering a device on (from PCI state D3), but before accessing the device. See power(9E).

RETURN VALUES  pci_save_config_regs() and pci_restore_config_regs() return:

DDI_SUCCESS  Operation completed successfully.
DDI_FAILURE  Operation failed to complete successfully.

CONTEXT  Both these functions can be called from user or kernel context.

EXAMPLES  EXAMPLE 1 Invoking the save and restore functions

static int
xx_power(dev_info_t *dip, int component, int level) {
    struct xx *xx;
    int rval = DDI_SUCCESS;
    xx = ddi_get_soft_state(xx_softstate, ddi_get_instance(dip));
    if (xx == NULL) {
        return (DDI_FAILURE);
    }
    mutex_enter(&xx->x_mutex);
    switch (level) {
    case PM_LEVEL_D0:
        XX_POWER_ON(xx);
        if (pci_restore_config_regs(dip) == DDI_FAILURE) {
            /*
             * appropriate error path handling here
             */
            ...
...
EXAMPLE 1 Invoking the save and restore functions (Continued)

    rval = DDI_FAILURE;

break;

    case PM_LEVEL_D3:
        if (pci_save_config_regs(dip) == DDI_FAILURE) {
            /*
             * appropriate error path handling here
             */
            ...  
            rval = DDI_FAILURE;
        }  
        else {  
            XX_POWER_OFF(xx);
        }  
        break;
    
    default:
        rval = DDI_FAILURE;
        break;
    }

mutex_exit(&xx->x_mutex);
    return (rval);

ATTRIBUTES  See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5), power(9E)

Writing Device Drivers

PCI Bus Power Management Interface Specification Version 1.1

PCI Bus Specification Revision 2.1
### physio(9F)

**NAME**
physio, minphys – perform physical I/O

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>

int physio(int (*strat)(struct buf *), struct buf *bp, dev_t dev, int rw, void (*mincnt)(struct buf *), struct uio *uio);

void minphys(struct buf *bp);
```

**INTERFACE LEVEL**

- **physio()**
  - **strat**
    Pointer to device strategy routine.
  - **bp**
    Pointer to a buf(9S) structure describing the transfer. If bp is set to NULL then physio() allocates one which is automatically released upon completion.
  - **dev**
    The device number.
  - **rw**
    Read/write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.
  - **mincnt**
    Routine which bounds the maximum transfer unit size.
  - **uio**
    Pointer to the uio structure which describes the user I/O request.

- **minphys()**
  - **bp**
    Pointer to a buf structure.

**DESCRIPTION**

physio() performs unbuffered I/O operations between the device dev and the address space described in the uio structure.

Prior to the start of the transfer physio() verifies the requested operation is valid by checking the protection of the address space specified in the uio structure. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, strat(), is then called one or more times to perform the physical I/O operations. physio() uses biowait(9F) to block until strat() has completed each transfer. Upon completion, or detection of an error, physio() unlocks the pages and returns the error status.

physio() uses mincnt() to bound the maximum transfer unit size to the system, or device, maximum length. minphys() is the system mincnt() routine for use with physio() operations. Drivers which do not provide their own local mincnt() routines should call physio() with minphys().

minphys() limits the value of bp->b_bcount to a sensible default for the capabilities of the system. Drivers that provide their own mincnt() routine should also call minphys() to make sure they do not exceed the system limit.

**RETURN VALUES**

physio() returns:

- 0
  Upon success.
Upon failure, the `physio()` function can be called from the user context only. The `biodone()` function is called to signal the completion of each `buf` transfer. If `physio()` is not called, it will block forever. The `biowait()` function is called internally by `physio()` to block until each `buf` transfer is complete. It is the driver's responsibility to call `biodone()` when the transfer is complete, or `physio()` will block forever.

**See Also**
- `strategy(9E)`
- `biodone(9F)`
- `biowait(9F)`
- `buf(9S)`
- `uio(9S)`

**Writing Device Drivers**

**Warnings**
Since `physio()` calls `biowait()` to block until each `buf` transfer is complete, it is the driver's responsibility to call `biodone(9F)` when the transfer is complete, or `physio()` will block forever.
pm_busy_component(9F)

NAME pm_busy_component, pm_idle_component – Control device component availability for Power Management

SYNOPSIS

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_busy_component(dev_info_t *dip, int component);
int pm_idle_component(dev_info_t *dip, int component);
```

INTERFACE LEVEL Solaris DDI specific (Solaris DDI)

pm_busy_component

- `dip`: Pointer to the device’s dev_info structure.
- `component`: The number of the component to be power-managed.

pm_idle_component

- `dip`: Pointer to the device’s dev_info structure.
- `component`: The number of the component to be power-managed.

DESCRIPTION

The `pm_busy_component()` function sets `component` of `dip` to be busy. Calls to `pm_busy_component()` are stacked, requiring a corresponding number of calls to `pm_idle_component()` to make the component idle again. When a device is busy it will not be power-managed by the system.

The `pm_idle_component()` function marks `component` idle, recording the time that `component` went idle. This function must be called once for each call to `pm_busy_component()`. A component which is idle is available to be power-managed by the system. The `pm_idle_component()` function has no effect if the component is already idle, except to update the system’s notion of when the device went idle.

RETURN VALUES

The `pm_busy_component()` and `pm_idle_component()` functions return:

- `DDI_SUCCESS`: Successfully set the indicated component busy or idle.
- `DDI_FAILURE`: Invalid component number `component` or the device has no components.

CONTEXT

These functions can be called from user or kernel context. These functions may also be called from interrupt context, providing they are not the first Power Management function called by the driver.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

power.conf(4), pm(7D), pm_create_components(9F), pm_destroy_components(9F), pm_raise_power(9F), pm(9P), pm-components(9P)
Writing Device Drivers
pm_create_components(9F)

NAME

pm_create_components, pm_destroy_components – Create or destroy power-manageable components

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_create_components(dev_info_t *dip, int components);

void pm_destroy_components(dev_info_t *dip);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

dip Pointer to the device’s dev_info structure

components Number of components to create

DESCRIPTION

The pm_create_components() and pm_destroy_components() functions are now obsolete and will be removed in a future release. It is recommended that the driver use pm-components(9) instead.

The pm_create_components() function creates power-manageable components for a device. It should be called from the driver’s attach(9E) entry point if the device has power-manageable components.

The correspondence of components to parts of the physical device controlled by the driver are the responsibility of the driver.

The pm_destroy_components() function removes all components from the device. It should be called from the driver’s detach(9E) entry point.

RETURN VALUES

The pm_create_components() function returns:

DDI_SUCCESS Components are successfully created.

DDI_FAILURE The device already has components.

CONTEXT

These functions may be called from user or kernel context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

power.conf(4), pm(7D), attach(9E), detach(9E), pm_busy_component(9F), pm_idle_component(9F), pm(9P), pm-components(9P)

Writing Device Drivers
pm_create_components, pm_destroy_components – Create or destroy power-manageable components

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_create_components(dev_info_t *dip, int components);
void pm_destroy_components(dev_info_t *dip);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

- `dip` Pointer to the device’s dev_info structure
- `components` Number of components to create

DESCRIPTION

The `pm_create_components()` and `pm_destroy_components()` functions are now obsolete and will be removed in a future release. It is recommended that the driver use `pm-components(9)` instead.

The `pm_create_components()` function creates power-manageable components for a device. It should be called from the driver’s attach(9E) entry point if the device has power-manageable components.

The correspondence of components to parts of the physical device controlled by the driver are the responsibility of the driver.

The `pm_destroy_components()` function removes all components from the device. It should be called from the driver’s detach(9E) entry point.

RETURN VALUES

The `pm_create_components()` function returns:

- `DDI_SUCCESS` Components are successfully created.
- `DDI_FAILURE` The device already has components.

CONTEXT

These functions may be called from user or kernel context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

`power.conf(4), pm(7D), attach(9E), detach(9E), pm_busy_component(9F), pm_idle_component(9F), pm(9P), pm-components(9P)`

`Writing Device Drivers`
pm_get_normal_power

NAME
pm_get_normal_power, pm_set_normal_power – get or set a device component’s normal power level

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_get_normal_power(dev_info_t *dip, int component);
void pm_set_normal_power(dev_info_t *dip, int component, int level);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
pm_get_normal_power() parameters:
dip Pointer to the device’s dev_info structure
component Number of component from which to get normal power level

pm_set_normal_power() parameters:
dip Pointer to the device’s dev_info structure
component Number of component for which to set normal power level
level Component’s new normal power level

DESCRIPTION
The pm_get_normal_power() and pm_set_normal_power() functions are now obsolete and will be removed in a future release. It is recommended that device drivers use new automatic device Power Management interfaces.

The pm_get_normal_power() function returns the normal power level of component of the device dip.

The pm_set_normal_power() function sets the normal power level of component of the device dip to level.

When a device has been power managed and is being returned to a state to be used by the system, it will be brought to its normal power level. Except for a power level of 0, which is defined by the system to mean “powered off,” the interpretation of the meaning of the power level is entirely up to the driver.

RETURN VALUES
The pm_get_normal_power() function returns:
level The normal power level of the specified component (a positive integer).

DDI_FAILURE Invalid component number component or the device has no components.

CONTEXT
These functions can be called from user or kernel context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:
pm_get_normal_power(9F)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

power.conf(4), pm(7D), pm(9P), power(9E), pm_busy_component(9F), pm_create_components(9F), pm_destroy_components(9F), pm_idle_component(9F)

Writing Device Drivers
pm_idle_component(9F)

NAME pm_busy_component, pm_idle_component – Control device component availability for Power Management

SYNOPSIS #include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_busy_component(dev_info_t *dip, int component);
int pm_idle_component(dev_info_t *dip, int component);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI)

pm_busy_component

dip Pointer to the device’s dev_info structure.
component The number of the component to be power-managed.

pm_idle_component
dip Pointer to the device’s dev_info structure.
component The number of the component to be power-managed.

DESCRIPTION The pm_busy_component() function sets component of dip to be busy. Calls to pm_busy_component() are stacked, requiring a corresponding number of calls to pm_idle_component() to make the component idle again. When a device is busy it will not be power-managed by the system.

The pm_idle_component() function marks component idle, recording the time that component went idle. This function must be called once for each call to pm_busy_component(). A component which is idle is available to be power-managed by the system. The pm_idle_component() function has no effect if the component is already idle, except to update the system’s notion of when the device went idle.

RETURN VALUES The pm_busy_component() and pm_idle_component() functions return:

DDI_SUCCESS Successfully set the indicated component busy or idle.
DDI_FAILURE Invalid component number component or the device has no components.

CONTEXT These functions can be called from user or kernel context. These functions may also be called from interrupt context, providing they are not the first Power Management function called by the driver.

ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO power.conf(4), pm(7D), pm_create_components(9F), pm_destroy_components(9F), pm_raise_power(9F), pm(9P), pm-components(9P)
Writing Device Drivers

pm_idle_component(9F)
pm_lower_power(9F)

NAME
pm_raise_power, pm_lower_power – Raise or lower power of components

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_raise_power(dev_info_t *dip, int component, int level);
int pm_lower_power(dev_info_t *dip, int component, int level);

INTERFACE LEVEL

pm_raise_power

dip Pointer to the device’s dev_info structure
component The number of the component for which a power level change is desired
level The power level to which the indicated component will be raised

pm_lower_power
dip Pointer to the device’s dev_info structure
component The number of the component for which a power level change is desired
level The power level to which the indicated component will be lowered

DESCRIPTION
The pm_raise_power(9F) function requests the Power Management framework to raise the power level of component of dip to at least level.

The state of the device should be examined before each physical access. The pm_raise_power(9F) function should be called to set a component to the required power level if the operation to be performed requires the component to be at a power level higher than its current power level.

When pm_raise_power(9F) returns with success, the component is guaranteed to be at least at the requested power level. All devices that depend on this will be at their full power level. Since the actual device power level may be higher than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from pm_raise_power(9F).

The pm_raise_power(9F) function may cause re-entry of the driver power(9E) to raise the power level. Deadlock may result if the driver locks are held across the call to pm_raise_power(9F).

The pm_lower_power(9F) function requests the Power Management framework to lower the power level of component of dip to at most level.

Normally, transitions to lower power levels are initiated by the Power Management framework based on component idleness. However, when detaching, the driver should also initiate reduced power levels by setting the power level of all device components to their lowest levels. The pm_lower_power(9F) function is intended for this use only, and will return DDI_FAILURE if the driver is not detaching at the time of the call.
If automatic Power Management is disabled (see `dtpower(1M)` and `power.conf(4)`), `pm_lower_power(9F)` returns `DDI_SUCCESS` without changing the power level of the component. Otherwise, when `pm_lower_power(9F)` returns with success, the component is guaranteed to be at most at the requested power level. Since the actual device power level may be lower than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from `pm_lower_power(9F)`.

The `pm_lower_power(9F)` may cause re-entry of the driver `power(9E)` to lower the power level. Deadlock may result if the driver locks are held across the call to `pm_raise_power(9F)`.

**RETURN VALUES**

The `pm_raise_power(9F)` function returns:

- `DDI_SUCCESS`: Component is now at the requested power level or higher.
- `DDI_FAILURE`: Component or level is out of range, or the framework was unable to raise the power level of the component to the requested level.

The `pm_lower_power(9F)` function returns:

- `DDI_SUCCESS`: Component is now at the requested power level or lower, or automatic Power Management is disabled.
- `DDI_FAILURE`: Component or level is out of range, or the framework was unable to lower the power level of the component to the requested level, or the device is not detaching.

**EXAMPLES**

A hypothetical disk driver might include this code to handle `pm_raise_power(9F)`:

```c
static int
xxdisk_strategy(struct buf *bp)
{

  ...

  /*
   * At this point we have determined that we need to raise the
   * power level of the device. Since we have to drop the
   * mutex, we need to take care of case where framework is
   * lowering power at the same time we are raising power.
   * We resolve this by marking the device busy and failing
   * lower power in power() entry point when device is busy.
   */
  ASSERT(mutex_owned(xsp->lock));
  if (xsp->pm_busycount < 1) {
      /*
       * Component is not already marked busy
       */
      if (pm_busy_component(xsp->dip, XXDISK_COMPONENT) != DDI_SUCCESS) {
          bioerror(bp,EIO);
          biodone(bp);
          return 0;
      }
  }
```

Kernel Functions for Drivers 1283
pm_lower_power(9F)

}  
xsp->pm_busycnt++;  
}  
mutex_exit(xsp->lock);  
if (pm_raise_power(xsp->dip,  
XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
    bioerror(bp, EIO);  
biodone(bp);  
return (0);  
}
mutex_enter(xsp->lock);  
....
}

xxdisk_power(dev_info *dip, int comp, int level)  
{
...  
/*  
* We fail the power() entry point if the device is busy and  
* request is to lower the power level.  
*/  
ASSERT(mutex_owned( xsp->lock)));  
if (xsp->pm_busycnt >= 1) {
    (level < xsp->cur_level) {
        mutex_exit( xsp->lock);  
        mutex_exit( xsp->lock);  
        return (DDI_FAILURE);  
    }
...  
}

CONTEXT These functions can be called from user or kernel context.

ATTRIBUTES See attributes(5) for a description of the following attribute:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO power.conf(4), pm(7D), power(9E), pm_busy_component(9F), pm_idle_component(9F), pm(9P), pm-components(9P)

Writing Device Drivers
pm_power_has_changed(9F)

NAME  pm_power_has_changed – Notify Power Management framework of autonomous power level change

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_power_has_changed(dev_info_t *dip, int component, int level);

INTERFACE LEVEL
PARAMETERS
  dip                  Pointer to the device dev_info structure
  component            Number of the component that has changed power level
  level                Power level to which the indicated component has changed

DESCRIPTION
The pm_power_has_changed(9) function notifies the Power Management framework that the power level of component of dip has changed to level.

Normally power level changes are initiated by the Power Management framework due to device idleness, or through a request to the framework from the driver via pm_raise_power(9F) or pm_lower_power(9F), but some devices may change power levels on their own. For the framework to track the power level of the device under these circumstances, the framework must be notified of autonomous power level changes by a call to pm_power_has_changed().

Because of the asynchronous nature of these events, the Power Management framework might have called power(9E) between the device’s autonomous power level change and the driver calling pm_power_has_changed(), or the framework may be in the process of changing the power level when pm_power_has_changed() is called. To handle these situations correctly, the driver should verify that the device is indeed at the level or set the device to the level if it doesn’t support inquiring of power levels, before calling pm_power_has_changed(). In addition, the driver should prevent a power(9E) entry point from running in parallel with pm_power_has_changed().

RETURN VALUES
The pm_power_has_changed() function returns:

  DDI_SUCCESS     The power level of component was successfully updated to level.
  DDI_FAILURE     Invalid component component or power level level

CONTEXT
This function can be called from user or kernel context. This function can also be called from interrupt context, providing that it is not the first Power Management function called by the driver.

EXAMPLES
A hypothetical driver might include this code to handle pm_power_has_changed(9):

static int
xxusb_intr(struct buf *bp)
{
...

 Kernel Functions for Drivers   1285
/*
   * At this point the device has informed us that it has
   * changed power level on its own. Inform this to framework.
   * We need to take care of the case when framework has
   * already called power() entry point and changed power level
   * before we were able to inform framework of this change.
   * Handle this by comparing the informed power level with
   * the actual power level and only doing the call if they
   * are same. In addition, make sure that power() doesn’t get
   * run in parallel with this code by holding the mutex.
   */

void pm_power_has_changed(dev_info *dip, XXUSB_COMPONENT, int level)
{
    ... } 

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability level</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO

power.conf(4), pm(7D), power(9E), pm_busy_components(9F),
pm_idle_components(9F), pm_raise_power(9F), pm_lower_power(9F), pm(9P),
pm-components(9P)

Writing Device Drivers
pm_raise_power(9F)

NAME
pm_raise_power, pm_lower_power – Raise or lower power of components

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_raise_power(dev_info_t *dip, int component, int level);
int pm_lower_power(dev_info_t *dip, int component, int level);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

pm_raise_power

<table>
<thead>
<tr>
<th>dip</th>
<th>Pointer to the device’s dev_info structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>The number of the component for which a power level change is desired</td>
</tr>
<tr>
<td>level</td>
<td>The power level to which the indicated component will be raised</td>
</tr>
</tbody>
</table>

pm_lower_power

<table>
<thead>
<tr>
<th>dip</th>
<th>Pointer to the device’s dev_info structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>The number of the component for which a power level change is desired</td>
</tr>
<tr>
<td>level</td>
<td>The power level to which the indicated component will be lowered</td>
</tr>
</tbody>
</table>

DESCRIPTION
The pm_raise_power(9F) function requests the Power Management framework to raise the power level of component of dip to at least level.

The state of the device should be examined before each physical access. The pm_raise_power(9F) function should be called to set a component to the required power level if the operation to be performed requires the component to be at a power level higher than its current power level.

When pm_raise_power(9F) returns with success, the component is guaranteed to be at least at the requested power level. All devices that depend on this will be at their full power level. Since the actual device power level may be higher than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from pm_raise_power(9F).

The pm_raise_power(9F) function may cause re-entry of the driver power(9E) to raise the power level. Deadlock may result if the driver locks are held across the call to pm_raise_power(9F).

The pm_lower_power(9F) function requests the Power Management framework to lower the power level of component of dip to at most level.

Normally, transitions to lower power levels are initiated by the Power Management framework based on component idleness. However, when detaching, the driver should also initiate reduced power levels by setting the power level of all device components to their lowest levels. The pm_lower_power(9F) function is intended for this use only, and will return DDI_FAILURE if the driver is not detaching at the time of the call.
pm_raise_power(9F)

If automatic Power Management is disabled (see dtpower(1M) and power.conf(4)), pm_lower_power(9F) returns DDI_SUCCESS without changing the power level of the component. Otherwise, when pm_lower_power(9F) returns with success, the component is guaranteed to be at most at the requested power level. Since the actual device power level may be lower than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from pm_lower_power(9F).

The pm_lower_power(9F) may cause re-entry of the driver power(9E) to lower the power level. Deadlock may result if the driver locks are held across the call to pm_raise_power(9F).

RETURN VALUES

The pm_raise_power(9F) function returns:

DDI_SUCCESS  Component is now at the requested power level or higher.

DDI_FAILURE  Component or level is out of range, or the framework was unable to raise the power level of the component to the requested level.

The pm_lower_power(9F) function returns:

DDI_SUCCESS  Component is now at the requested power level or lower, or automatic Power Management is disabled.

DDI_FAILURE  Component or level is out of range, or the framework was unable to lower the power level of the component to the requested level, or the device is not detaching.

EXAMPLES

A hypothetical disk driver might include this code to handle pm_raise_power(9F):

```c
static int xxdisk_strategy(struct buf *bp)
{
...

  /* At this point we have determined that we need to raise the
  * power level of the device. Since we have to drop the
  * mutex, we need to take care of case where framework is
  * lowering power at the same time we are raising power.
  * We resolve this by marking the device busy and failing
  * lower power in power() entry point when device is busy.
  */
  if (xsp->pm_busy < 1) {
    /*
    * Component is not already marked busy
    */
    ...
```
pm_raise_power(9F)

```c
...
xxxdisk_power(dev_info *dip, int comp, int level)
{
...

  /*
   * We fail the power() entry point if the device is busy and
   * request is to lower the power level.
   */
  ASSERT(mutex_owned( xsp->lock));
  if (xsp->pm_busycnt >= 1) {
    (level < xsp->cur_level) {
      mutex_exit( xsp->lock);
      return (DDI_FAILURE);
    }
  }
...
}
```

**CONTEXT**

These functions can be called from user or kernel context.

**ATTRIBUTES**

See attributes(5) for a description of the following attribute:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

**SEE ALSO**

power.conf(4), pm(7D), power(9E), pm_busy_component(9F),
pm_idle_component(9F), pm(9P), pm-components(9P)

*Writing Device Drivers*
### pm_set_normal_power(9F)

#### NAME
pm_get_normal_power, pm_set_normal_power – get or set a device component’s normal power level

#### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_get_normal_power(dev_info_t *dip, int component);
void pm_set_normal_power(dev_info_t *dip, int component, int level);
```

#### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

#### PARAMETERS
**pm_get_normal_power()** parameters:
- `dip` Pointer to the device’s dev_info structure
- `component` Number of component from which to get normal power level

**pm_set_normal_power()** parameters:
- `dip` Pointer to the device’s dev_info structure
- `component` Number of component for which to set normal power level
- `level` Component’s new normal power level

#### DESCRIPTION
The **pm_get_normal_power()** and **pm_set_normal_power()** functions are now obsolete and will be removed in a future release. It is recommended that device drivers use new automatic device Power Management interfaces.

The **pm_get_normal_power()** function returns the normal power level of `component` of the device `dip`.

The **pm_set_normal_power()** function sets the normal power level of `component` of the device `dip` to `level`.

When a device has been power managed and is being returned to a state to be used by the system, it will be brought to its normal power level. Except for a power level of 0, which is defined by the system to mean “powered off,” the interpretation of the meaning of the power level is entirely up to the driver.

#### RETURN VALUES
The **pm_get_normal_power()** function returns:
- `level` The normal power level of the specified component (a positive integer).
- `DDI_FAILURE` Invalid component number `component` or the device has no components.

#### CONTEXT
These functions can be called from user or kernel context.

#### ATTRIBUTES
See attributes(5) for descriptions of the following attributes:
pm_set_normal_power(9F)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

power.conf(4), pm(7D), pm(9P), power(9E), pm_busy_component(9F), pm_create_components(9F), pm_destroy_components(9F), pm_idle_component(9F)

Writing Device Drivers
pm_trans_check(9F)

NAME
pm_trans_check – Device power cycle advisory check

SYNOPSIS
#include <sys/sunddi.h>

int pm_trans_check(struct pm_trans_data *datap, time_t *intervalp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

PARAMETERS
datap Pointer to a pm_trans_data structure
intervalp Pointer to time difference when next power cycle will be advised

DESCRIPTION
The pm_trans_check(9F) function checks if a power-cycle is currently advised based on data in the pm_trans_data structure. This function is provided to prevent damage to devices from excess power cycles; drivers for devices that are sensitive to the number of power cycles should call pm_trans_check(9F) from their power(9E) function before powering-off a device. If pm_trans_check(9F) indicates that the device should not be power cycled, the driver should not attempt to power cycle the device and should fail the call to power(9E) entry point.

If pm_trans_check(9F) returns that it is not advised to power cycle the device, it attempts to calculate when the next power cycle is advised, based on the supplied parameters. In such case, intervalp returns the time difference (in seconds) from the current time to when the next power cycle is advised. If the time for the next power cycle cannot be determined, intervalp indicates 0.

To avoid excessive calls to the power(9E) entry point during a period when power cycling is not advised, the driver should mark the corresponding device component busy for the intervalp time period (if interval is not 0). Conveniently, the driver can utilize the fact that calls to pm_busy_component(9F) are stacked. If power cycling is not advised, the driver can call pm_busy_component(9F) and issue a timeout(9F) for the intervalp time. The timeout() handler can issue the corresponding pm_idle_component(9F) call.

When the format field of pm_trans_data is set to DC_SCSI_FORMAT, the caller must provide valid data in svc_date[], lifemax, and ncycles. Currently, flag must be set to 0.

struct pm_scsi_cycles {
    int lifemax; /* lifetime max power cycles */
    int ncycles; /* number of cycles so far */
    char svc_date[DC_SCSI_MFR_LEN]; /* service date YYYYWW */
    int flag; /* reserved for future */
};

struct pm_trans_data {
    int format; /* data format */
    union {
        struct pm_scsi_cycles scsi_cycles;
    } un;
};

RETURN VALUES
1 Power cycle is advised
pm_trans_check(9F)

0       Power cycle is not advised
-1      Error due to invalid argument.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO
power.conf(4), attributes(5), power(9E)

Writing Device Drivers

Using Power Management
pollwakeup(9F)

NAME pollwakeup – inform a process that an event has occurred

SYNOPSIS #include <sys/poll.h>

void pollwakeup(struct pollhead *php, short event);

INTERFACE LEVEL
PARAMETERS
Architecture independent level 1 (DDI/DKI).

php Pointer to a pollhead structure.
event Event to notify the process about.

DESCRIPTION pollwakeup() wakes a process waiting on the occurrence of an event. It should be called from a driver for each occurrence of an event. The pollhead structure will usually be associated with the driver’s private data structure associated with the particular minor device where the event has occurred. See chpoll(9E) and poll(2) for more detail.

CONTEXT pollwakeup() can be called from user or interrupt context.

SEE ALSO poll(2), chpoll(9E)

Writing Device Drivers

NOTES Driver defined locks should not be held across calls to this function.
proc_signal, proc_ref, proc_unref – send a signal to a process

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/signal.h>

void *proc_ref(void);
void proc_unref(void *pref);
int proc_signal(void *pref, int sig);

Solaris DDI specific (Solaris DDI).

pref A handle for the process to be signalled.
sig Signal number to be sent to the process.

This set of routines allows a driver to send a signal to a process. The routine
proc_ref() is used to retrieve an unambiguous reference to the process for
signalling purposes. The return value can be used as a unique handle on the process,
even if the process dies. Because system resources are committed to a process
reference, proc_unref() should be used to remove it as soon as it is no longer
needed. proc_signal() is used to send signal sig to the referenced process. The
following set of signals may be sent to a process from a driver:

SIGHUP The device has been disconnected.
SIGINT The interrupt character has been received.
SIGQUIT The quit character has been received.
SIGPOLL A pollable event has occurred.
SIGKILL Kill the process (cannot be caught or ignored).
SIGWINCH Window size change.
SIGURG Urgent data are available.

See signal(3HEAD) for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, proc_signal() returns an
error code; the caller should remove the reference on the process by calling
proc_unref().

The driver writer must ensure that for each call made to proc_ref(), there is exactly
one corresponding call to proc_unref().

proc_ref() returns the following:

pref An opaque handle used to refer to the current process.

proc_signal() returns the following:

0 The process existed before the signal was sent.
The process no longer exists; no signal was sent.

**CONTEXT**

proc_unref() and proc_signal() can be called from user or interrupt context.
proc_ref() should only be called from user context.

**SEE ALSO**
signal(3HEAD), putnextctl11(9F)

*Writing Device Drivers*
### proc_signal(9F)

**NAME**
proc_signal, proc_ref, proc_unref – send a signal to a process

**SYNOPSIS**
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/signal.h>

void *proc_ref(void);
void proc_unref(void *pref);
int proc_signal(void *pref, int sig);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- **pref** A handle for the process to be signalled.
- **sig** Signal number to be sent to the process.

**DESCRIPTION**
This set of routines allows a driver to send a signal to a process. The routine `proc_ref()` is used to retrieve an unambiguous reference to the process for signalling purposes. The return value can be used as a unique handle on the process, even if the process dies. Because system resources are committed to a process reference, `proc_unref()` should be used to remove it as soon as it is no longer needed.`proc_signal()` is used to send signal `sig` to the referenced process. The following set of signals may be sent to a process from a driver:

- **SIGHUP** The device has been disconnected.
- **SIGINT** The interrupt character has been received.
- **SIGQUIT** The quit character has been received.
- **SIGPOLL** A pollable event has occurred.
- **SIGKILL** Kill the process (cannot be caught or ignored).
- **SIGWINCH** Window size change.
- **SIGURG** Urgent data are available.

See `signal(3HEAD)` for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, `proc_signal()` returns an error code; the caller should remove the reference on the process by calling `proc_unref()`.

The driver writer must ensure that for each call made to `proc_ref()`, there is exactly one corresponding call to `proc_unref()`.

**RETURN VALUES**
- `proc_ref()` returns the following:
  - **pref** An opaque handle used to refer to the current process.

- `proc_signal()` returns the following:
  - **0** The process existed before the signal was sent.
proc_signal(9F)

-1 The process no longer exists; no signal was sent.

CONTEXT proc_unref() and proc_signal() can be called from user or interrupt context.
proc_ref() should only be called from user context.

SEE ALSO signal(3HEAD), putnextctl11(9F)

Writing Device Drivers
proc_signal, proc_ref, proc_unref – send a signal to a process

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/signal.h>

void *proc_ref(void);
void proc_unref(void *pref);
int proc_signal(void *pref, int sig);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
pref A handle for the process to be signalled.
sig Signal number to be sent to the process.

DESCRIPTION
This set of routines allows a driver to send a signal to a process. The routine proc_ref() is used to retrieve an unambiguous reference to the process for signalling purposes. The return value can be used as a unique handle on the process, even if the process dies. Because system resources are committed to a process reference, proc_unref() should be used to remove it as soon as it is no longer needed. proc_signal() is used to send signal sig to the referenced process. The following set of signals may be sent to a process from a driver:

SIGHUP The device has been disconnected.
SIGINT The interrupt character has been received.
SIGQUIT The quit character has been received.
SIGPOLL A pollable event has occurred.
SIGKILL Kill the process (cannot be caught or ignored).
SIGWINCH Window size change.
SIGURG Urgent data are available.

See signal(3HEAD) for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, proc_signal() returns an error code; the caller should remove the reference on the process by calling proc_unref().

The driver writer must ensure that for each call made to proc_ref(), there is exactly one corresponding call to proc_unref().

RETURN VALUES
proc_ref() returns the following:
pref An opaque handle used to refer to the current process.

proc_signal() returns the following:
0 The process existed before the signal was sent.
.proc_unref(9F)

-1  The process no longer exists; no signal was sent.

**CONTEXT**

.proc_unref() and proc_signal() can be called from user or interrupt context.
.proc_unref() should only be called from user context.

**SEE ALSO**

signal(3HEAD), putnextctl11(9F)

*Writing Device Drivers*
### NAME
ptob – convert size in pages to size in bytes

### SYNOPSIS
```
#include <sys/ddi.h>

unsigned long ptob(unsigned long numpages);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- **numpages**
  Size in number of pages to convert to size in bytes.

### DESCRIPTION
This function returns the number of bytes that are contained in the specified number of pages. For example, if the page size is 2048, then `ptob(2)` returns 4096. `ptob(0)` returns 0.

### RETURN VALUES
The return value is always the number of bytes in the specified number of pages. There are no invalid input values, and no checking will be performed for overflow in the case of a page count whose corresponding byte count cannot be represented by an `unsigned long`. Rather, the higher order bits will be ignored.

### CONTEXT
`ptob()` can be called from user or interrupt context.

### SEE ALSO
- `btop(9F)`, `btopl(9F)`, `ddi_ptob(9F)`
- *Writing Device Drivers*
**NAME**
pullupmsg – concatenate bytes in a message

**SYNOPSIS**
```c
#include <sys/stream.h>

int pullupmsg(mblk_t *mp, ssize_t len);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `mp` Pointer to the message whose blocks are to be concatenated. mblk_t is an instance of the msgb(9S) structure.
- `len` Number of bytes to concatenate.

**DESCRIPTION**
pullupmsg() tries to combine multiple data blocks into a single block. pullupmsg() concatenates and aligns the first `len` data bytes of the message pointed to by `mp`. If `len` equals -1, all data are concatenated. If `len` bytes of the same message type cannot be found, pullupmsg() fails and returns 0.

**RETURN VALUES**
On success, 1 is returned; on failure, 0 is returned.

**CONTEXT**
pullupmsg() can be called from user or interrupt context.

**EXAMPLES**
This is a driver write srv(9E) (service) routine for a device that does not support scatter/gather DMA. For all M_DATA messages, the data will be transferred to the device with DMA. First, try to pull up the message into one message block with the pullupmsg() function (line 12). If successful, the transfer can be accomplished in one DMA job. Otherwise, it must be done one message block at a time (lines 19–22). After the data has been transferred to the device, free the message and continue processing messages on the queue.

```c
1 xxxwrsrv(q)
2 queue_t *q;
3 {
4     mblk_t *mp;
5     mblk_t *tmp;
6     caddr_t dma_addr;
7     ssize_t dma_len;
8     while ((mp = getq(q)) != NULL) {
9         switch (mp->b_datap->db_type) {
10             case M_DATA:
11                 if (pullupmsg(mp, -1)) {
12                     dma_addr = vtop(mp->b_rptr);
13                     dma_len = mp->b_wptr - mp->b_rptr;
14                     xxx_do_dma(dma_addr, dma_len);
15                     freemsg(mp);
16                     break;
17                 }
18             }
19             for (tmp = mp; tmp; tmp = tmp->b_cont) {
20                 dma_addr = vtop(tmp->b_rptr);
21                 dma_len = tmp->b_wptr - tmp->b_rptr;
22                 xxx_do_dma(dma_addr, dma_len);
23             }
24         }
25     }
26 }
```
EXAMPLE 1 Using pullupmsg() (Continued)

```c
23 } 
24 freemsg(mp); 
25 break; 
...
26 } 
27 } 
28 }
```

SEE ALSO

srv(9E), allocb(9F), msgpullup(9F), msgb(9S)

Writing Device Drivers

STREAMS Programming Guide

NOTES

pullupmsg() is not included in the DKI and will be removed from the system in a future release. Device driver writers are strongly encouraged to use msgpullup(9F) instead of pullupmsg().
NAME  put – call a STREAMS put procedure

SYNOPSIS  
#include <sys/stream.h>
#include <sys/ddi.h>

void put(queue_t *q, mblk_t *mp);

INTERFACE
LEVEL
PARAMETERS
Architecture independent level 1 (DDI/DKI).

q  Pointer to a STREAMS queue.

mp  Pointer to message block being passed into queue.

DESCRIPTION  
put() calls the put procedure (put(9E) entry point) for the STREAMS queue
specified by q, passing it the message block referred to by mp. It is typically used by a
driver or module to call its own put procedure.

CONTEXT  
put() can be called from a STREAMS module or driver put or service routine, or
from an associated interrupt handler, timeout, bufcall, or esballoc call-back. In the
latter cases, the calling code must guarantee the validity of the q argument.

Since put() may cause re-entry of the module (as it is intended to do), mutexes or
other locks should not be held across calls to it, due to the risk of single-party
deadlock (put(9E), putnext(9F), putctl(9F), qreply(9F)). This function is provided
as a DDI/DKI conforming replacement for a direct call to a put procedure.

SEE ALSO  
put(9E), freezestr(9F), putctl(9F), putctll(9F), putnext(9F), putnextctl(9F),
putnextctl1(9F), qprocson(9F), qreply(9F)

Writing Device Drivers

STREAMS Programming Guide

NOTES  
The caller cannot have the stream frozen when calling this function. See
freezestr(9F).

DDI/DKI conforming modules and drivers are no longer permitted to call put
procedures directly, but must call through the appropriate STREAMS utility function,
for example, put(9E), putnext(9F), putctl(9F), and qreply(9F). This function is
provided as a DDI/DKI conforming replacement for a direct call to a put procedure.

The put() and putnext() functions should be called only after qprocson() is
finished.
### NAME
putbq – place a message at the head of a queue

### SYNOPSIS
```c
#include <sys/stream.h>

int putbq(queue_t *q, mblk_t *bp);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- **q**
  - Pointer to the queue.
- **bp**
  - Pointer to the message block.

### DESCRIPTION
`putbq()` places a message at the beginning of the appropriate section of the message queue. There are always sections for high priority and ordinary messages. If other priority bands are used, each will have its own section of the queue, in priority band order, after high priority messages and before ordinary messages. `putbq()` can be used for ordinary, priority band, and high priority messages. However, unless precautions are taken, using `putbq()` with a high priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.

This function is usually called when `bcanput(9F)` or `canput(9F)` determines that the message cannot be passed on to the next stream component. The flow control parameters are updated to reflect the change in the queue’s status. If `QNOENB` is not set, the service routine is enabled.

### RETURN VALUES
`putbq()` returns 1 upon success and 0 upon failure.

**Note** – Upon failure, the caller should call `freemsg(9F)` to free the pointer to the message block.

### CONTEXT
`putbq()` can be called from user or interrupt context.

### EXAMPLES
See the `bufcall(9F)` function page for an example of `putbq()`.

### SEE ALSO
- `bcanput(9F)`, `bufcall(9F)`, `canput(9F)`, `getq(9F)`, `putq(9F)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
putctl1(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>putctl1 – send a control message with a one-byte parameter to a queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int putctl1(queue_t *q, int type, int p);</td>
</tr>
<tr>
<td>INTERFACE LEVEL PARAMETERS</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>q</td>
<td>Queue to which the message is to be sent.</td>
</tr>
<tr>
<td>type</td>
<td>Type of message.</td>
</tr>
<tr>
<td>p</td>
<td>One-byte parameter.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>putctl1(), like putctl(9F), tests the type argument to make sure a data type has not been specified, and attempts to allocate a message block. The p parameter can be used, for example, to specify how long the delay will be when an M_DELAY message is being sent. putctl1() fails if type is M_DATA, M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful, putctl1() calls the put(9E) routine of the queue pointed to by q with the newly allocated and initialized message.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>On success, 1 is returned. 0 is returned if type is a data type, or if a message block cannot be allocated.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>putctl1() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>See the putctl(9F) function page for an example of putctl1().</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>put(9E), allocb(9F), datamsg(9F), putctl(9F), putnextctl1(9F)</td>
</tr>
</tbody>
</table>

Writing Device Drivers

STREAMS Programming Guide
NAME
putctl – send a control message to a queue

SYNOPSIS
#include <sys/stream.h>

int putctl(queue_t *q, int type);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
q Queue to which the message is to be sent.

type Message type (must be control, not data type).

DESCRIPTION
putctl() tests the type argument to make sure a data type has not been specified,
and then attempts to allocate a message block. putctl() fails if type is M_DATA,
M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful,
putctl() calls the put(9E) routine of the queue pointed to by q with the newly
allocated and initialized messages.

RETURN VALUES
On success, 1 is returned. If type is a data type, or if a message block cannot be
allocated, 0 is returned.

CONTEXT
putctl() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using putctl()

The send_ctl() routine is used to pass control messages downstream. M_BREAK
messages are handled with putctl() (line 11). putct1(9F) (line 16) is used for
M_DELAY messages, so that parm can be used to specify the length of the delay. In
either case, if a message block cannot be allocated a variable recording the number of
allocation failures is incremented (lines 12, 17). If an invalid message type is detected,
cmn_err(9F) panics the system (line 21).

```c
1 void
2    send_ctl(wrq, type, parm)
3         queue_t *wrq;
4         uchar_t type;
5         uchar_t parm;
6 {
7     extern int num_alloc_fail;
8     switch (type) {
9         case M_BREAK:
10            if (!putctl(wrq->q_next, M_BREAK))
11               num_alloc_fail++;
12               break;
13        case M_DELAY:
14            if (!putct1(wrq->q_next, M_DELAY, parm))
15               num_alloc_fail++;
16               break;
17         default:
18                 cmn_err(CE_PANIC, "send_ctl: bad message type passed");
19                   break;
20             
```
putctl(9F)

EXAMPLE 1 Using putctl()  (Continued)

23    }
24    }

SEE ALSO  put(9E), cmn_err(9F), datamsg(9F), putctl1(9F), putnextctl(9F)

Writing Device Drivers
STREAMS Programming Guide
**NAME**
putnext – send a message to the next queue

**SYNOPSIS**
```c
#include <sys/stream.h>
#include <sys/ddi.h>

void putnext(queue_t *q, mblk_t *mp);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `q`  
  Pointer to the queue from which the message `mp` will be sent.
- `mp`  
  Message to be passed.

**DESCRIPTION**
`putnext()` is used to pass a message to the `put()` routine of the next queue in the stream.

**RETURN VALUES**
None.

**CONTEXT**
`putnext()` can be called from user or interrupt context.

**EXAMPLES**
See `allocb()` for an example of using `putnext()`.

**SEE ALSO**
`put()`, `allocb()`, `put()`, `qprocson()`,

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**
The `put()` and `putnext()` functions should be called only after `qprocson()` is finished.
putnextctl1(9F)

| NAME       | putnextctl1 – send a control message with a one-byte parameter to a queue |
| SYNOPSIS   | #include <sys/stream.h> |
|            | int putnextctl1(queue_t *q, int type, int p); |
| INTERFACE  | Architecture independent level 1 (DDI/DKI). |
| LEVEL      | |
| PARAMETERS | q | Queue to which the message is to be sent. |
|            | type | Type of message. |
|            | p | One-byte parameter. |
| DESCRIPTION | putnextctl1(), like putctl1(9F), tests the type argument to make sure a data type has not been specified, and attempts to allocate a message block. The p parameter can be used, for example, to specify how long the delay will be when an M_DELAY message is being sent. putnextctl1() fails if type is M_DATA, M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful, putnextctl1() calls the put(9E) routine of the queue pointed to by q with the newly allocated and initialized message. A call to putnextctl1(q,type, p) is an atomic equivalent of putctl1(q->q_next, type, p). The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing q through its q_next field and then invoking putctl1(9F) proceeds without interference from other threads. putnextctl1() should always be used in preference to putctl1(9F) |
| RETURN VALUES | On success, 1 is returned. 0 is returned if type is a data type, or if a message block cannot be allocated. |
| CONTEXT | putnextctl1() can be called from user or interrupt context. |
| EXAMPLES | See the putnextctl(9F) function page for an example of putnextctl1(). |
| SEE ALSO | put(9E), allocb(9F), datamsg(9F), putctl1(9F), putnextctl1(9F) |

Writing Device Drivers

STREAMS Programming Guide

1310  man pages section 9: DDI and DKI Kernel Functions • Last Revised 29 Mar 1993
**NAME**

putnextctl – send a control message to a queue

---

**SYNOPSIS**

```c
#include <sys/stream.h>

int putnextctl(queue_t *q, int type);
```

---

**INTERFACE LEVEL PARAMETERS**

- `q` Queue to which the message is to be sent.
- `type` Message type (must be control, not data type).

---

**DESCRIPTION**

`putnextctl()` tests the `type` argument to make sure a data type has not been specified, and then attempts to allocate a message block. `putnextctl()` fails if `type` is M_DATA, M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful, `putnextctl()` calls the `put(9E)` routine of the queue pointed to by `q` with the newly allocated and initialized messages.

A call to `putnextctl(q, type)` is an atomic equivalent of `putctl(q->q_next,type)`. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing `q` through its `q_next` field and then invoking `putctl(9F)` proceeds without interference from other threads.

`putnextctl()` should always be used in preference to `putctl(9F)` on success, 1 is returned. If `type` is a data type, or if a message block cannot be allocated, 0 is returned.

**CONTEXT**

`putnextctl()` can be called from user or interrupt context.

---

**EXAMPLES**

**EXAMPLE 1**

The `send_ctl` routine is used to pass control messages downstream. M_BREAK messages are handled with `putnextctl()` (line 8). `putnextctl(9F)` (line 13) is used for M_DELAY messages, so that `parm` can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 9, 14). If an invalid message type is detected, `cmn_err(9F)` panics the system (line 18).

```c
void send_ctl(queue_t *wrq, uchar_t type, uchar_t parm)
{
    extern int num_alloc_fail;

    switch (type) {
    case M_BREAK:
        if (!putnextctl(wrq, M_BREAK))
            num_alloc_fail++;
        break;
    case M_DELAY:
        if (!putnextctl1(wrq, M_DELAY, parm))
            num_alloc_fail++;
        break;
    }
```

---

**RETURN VALUES**

On success, 1 is returned. If `type` is a data type, or if a message block cannot be allocated, 0 is returned.
EXAMPLE 1  (Continued)

    16     default:
    17         cmn_err(CB_PANIC, "send_ctl: bad message type passed");
    18         break;
    19     }
    20 }

SEE ALSO  put(9E), cmn_err(9F), datamsg(9F), putctl(9F), putnextctl(9F)

Writing Device Drivers

STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>putq – put a message on a queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int putq(queue_t *q, mblk_t *bp);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>q: Pointer to the queue to which the message is to be added.</td>
</tr>
<tr>
<td></td>
<td>bp: Message to be put on the queue.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>putq() is used to put messages on a driver’s queue after the module’s put routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. If QNOENB is not set, the service routine is enabled. If no other processing is done, putq() can be used as the module’s put routine.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>putq() returns 1 on success and 0 on failure.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>putq() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>See the datamsg(9F) function page for an example of putq().</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>datamsg(9F), putbq(9F), qenable(9F), rmvq(9F)</td>
</tr>
</tbody>
</table>

*Writing Device Drivers*

*STREAMS Programming Guide*
**NAME**
qassociate – associate STREAMS queue with driver instance

**SYNOPSIS**
```c
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int qassociate(queue_t *q, int instance);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI). This entry point is required for drivers which export cb_ops(9S) entry points.

**PARAMETERS**
- `queue_t *q`
  Pointer to a queue(9S) structure. Either the read or write queue can be used.
- `int instance`
  Driver instance number or -1.

**DESCRIPTION**
The qassociate() function associates the specified STREAMS queue with the specified instance of the bottom driver in the queue. Upon successful return, the stream is associated with the instance with any prior association dissolved.

A DLPI style-2 driver calls qassociate() while processing the DL_ATTACH_REQ message. The driver is also expected to call this interface while performing stream associations through other means, such as ndd(1M) ioctl commands.

If `instance` is -1, the stream is left unassociated with any hardware instance.

If the interface returns failure, the stream is not associated with the specified instance. Any prior association is left untouched.

The interface typically fails because of failure to locate and attach the device instance. The interface never fails if the specified instance is -1.

**CONTEXT**
qassociate() can be called from the stream’s put(9E) entry point.

**RETURN VALUES**
- 0
  Success.
- -1
  Failure.

**EXAMPLES**
A Style-2 network driver’s DL_ATTACH_REQ code would specify:
```c
if (qassociate(q, instance) != 0)
goto fail;
```

The association prevents Dynamic Reconfiguration (DR) from detaching the instance.

A Style-2 network driver’s DL_DETACH code would specify:
```c
(void) qassociate(q, -1);
```
This dissolves the queue’s association with any device instance.

A Style-2 network driver’s open(9E) code must call:

```
qassociate(q, -1);
```

This informs the framework that this driver has been modified to be DDI-compliant.

**SEE ALSO**

dlpi(7P), open(9E), put(9E), ddi_no_info(9F), queue(9S)
qbufcall(9F)

NAME
qbufcall – call a function when a buffer becomes available

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

bufcall_id_t qbufcall(queue_t *q, size_t size, uint_t pri,
                      void *func, void *arg, void *arg);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS
q Pointer to STREAMS queue structure.
size Number of bytes required for the buffer.
pri Priority of the allocb(9F) allocation request (not used).
func Function or driver routine to be called when a buffer becomes available.
arg Argument to the function to be called when a buffer becomes available.

DESCRIPTION
qbufcall() serves as a qtimeout(9F) call of indeterminate length. When a buffer allocation request fails, qbufcall() can be used to schedule the routine func to be called with the argument arg when a buffer becomes available. func may call allocb() or it may do something else.

The qbufcall() function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. See mt-streams(9F). qbufcall() schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to qbufcall(). All outstanding timeouts and bufcalls must be cancelled (using, respectively, quntimeout(9F) and qunbufcall(9F)) before a driver close routine can block and before the close routine calls qprocsoff(9F).

qprocson(9F) must be called before calling either qbufcall() or qtimeout(9F).

RETURN VALUES
If successful, qbufcall() returns a qbufcall ID that can be used in a call to qunbufcall(9F) to cancel the request. If the qbufcall() scheduling fails, func is never called and 0 is returned.

CONTEXT
qbufcall() can be called from user or interrupt context.

SEE ALSO
allocb(9F), mt-streams(9F), qprocson(9F), qtimeout(9F), qunbufcall(9F), quntimeout(9F)

Writing Device Drivers

STREAMS Programming Guide

WARNINGS
Even when func is called by qbufcall(), allocb(9F) can fail if another module or driver had allocated the memory before func was able to call allocb(9F).
NAME  qenable – enable a queue

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

void qenable(queue_t *q);

INTERFACE LEVEL
PARAMETERS  q  Pointer to the queue to be enabled.

DESCRIPTION  qenable() adds the queue pointed to by q to the list of queues whose service routines are ready to be called by the STREAMS scheduler.

CONTEXT  qenable() can be called from user or interrupt context.

EXAMPLES  See the dupb(9F) function page for an example of the qenable().

SEE ALSO  dupb(9F)

Writing Device Drivers

STREAMS Programming Guide
`qprocsoff(9F)`

**NAME**  
`qprocson, qprocsoff` – enable, disable put and service routines

**SYNOPSIS**  
```c
#include <sys/stream.h>
#include <sys/ddi.h>

void qprocson(queue_t *q);
void qprocsoff(queue_t *q);
```

**INTERFACE LEVEL**  
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**  
`q`  
Pointer to the RD side of a STREAMS queue pair.

**DESCRIPTION**  
`qprocson()` enables the put and service routines of the driver or module whose read queue is pointed to by `q`. Threads cannot enter the module instance through the put and service routines while they are disabled.

`qprocson()` must be called by the open routine of a driver or module before returning, and after any initialization necessary for the proper functioning of the put and service routines.

`qprocson()` must be called before calling `put(9F)`, `putnext(9F)`, `qbufcall(9F)`, `qtimeout(9F)`, `qwait(9F)`, or `qwait_sig(9F)`.

`qprocsoff()` must be called by the close routine of a driver or module before returning, and before deallocating any resources necessary for the proper functioning of the put and service routines. It also removes the queue’s service routines from the service queue, and blocks until any pending service processing completes.

The module or driver instance is guaranteed to be single-threaded before `qprocson()` is called and after `qprocsoff()` is called, except for threads executing asynchronous events such as interrupt handlers and callbacks, which must be handled separately.

**CONTEXT**  
These routines can be called from user or interrupt context.

**SEE ALSO**  
`close(9E)`, `open(9E)`, `put(9E)`, `srv(9E)`, `put(9F)`, `putnext(9F)`, `qbufcall(9F)`, `qtimeout(9F)`, `qwait(9F)`, `qwait_sig(9F)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**  
The caller may not have the STREAM frozen during either of these calls.
qprocson(9F)

NAME
qprocson, qprocsoff – enable, disable put and service routines

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

void qprocson(queue_t *q);
void qprocsoff(queue_t *q);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
qPointer to the RD side of a STREAMS queue pair.

DESCRIPTION
qprocson() enables the put and service routines of the driver or module whose read
queue is pointed to by q. Threads cannot enter the module instance through the put
and service routines while they are disabled.

qprocson() must be called by the open routine of a driver or module before
returning, and after any initialization necessary for the proper functioning of the put
and service routines.

qprocson() must be called before calling put(9F), putnext(9F), qbufcall(9F),
qtimeout(9F), qwait(9F), or qwait_sig(9F),

qprocsoff() must be called by the close routine of a driver or module before
returning, and before deallocating any resources necessary for the proper functioning
of the put and service routines. It also removes the queue’s service routines from the
service queue, and blocks until any pending service processing completes.

The module or driver instance is guaranteed to be single-threaded before
qprocson() is called and after qprocsoff() is called, except for threads executing
asynchronous events such as interrupt handlers and callbacks, which must be handled
separately.

CONTEXT
These routines can be called from user or interrupt context.

SEE ALSO
close(9E), open(9E), put(9E), srv(9E), put(9F), putnext(9F), qbufcall(9F),
qtimeout(9F), qwait(9F), qwait_sig(9F)

Writing Device Drivers
STREAMS Programming Guide

NOTES
The caller may not have the STREAM frozen during either of these calls.
NAME
qreply – send a message on a stream in the reverse direction

SYNOPSIS
#include <sys/stream.h>

void qreply(queue_t *q, mblk_t *mp);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
q Pointer to the queue.

DESCRIPTION
qreply() sends messages in the reverse direction of normal flow. That is, qreply(q, mp) is equivalent to putnext(OTHERQ(q), mp).

CONTEXT
qreply() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Canonical Flushing Code for STREAMS Drivers.

This example depicts the canonical flushing code for STREAMS drivers. Assume that the driver has service procedures so that there may be messages on its queues. See srv(9E). Its write-side put procedure handles M_FLUSH messages by first checking the FLUSHW bit in the first byte of the message, then the write queue is flushed (line 8) and the FLUSHW bit is turned off (line 9). See put(9E). If the FLUSHR bit is on, then the read queue is flushed (line 12) and the message is sent back up the read side of the stream with the qreply(9F) function (line 13). If the FLUSHR bit is off, then the message is freed (line 15). See the example for flushq(9F) for the canonical flushing code for modules.

1 xxxwput(q, mp)
2 queue_t *q;
3 mblk_t *mp;
4 {
5 switch(mp->b_datap->db_type) {
6 case M_FLUSH:
7 if (*mp->b_rptr & FLUSHW) {
8 flushq(q, FLUSHALL);
9 *mp->b_rptr &= ~FLUSHW;
10 }
11 if (*mp->b_rptr & FLUSHR) {
12 flushq(RD(q), FLUSHALL);
13 qreply(q, mp);
14 } else {
15 freemsg(mp);
16 }
17 break;
18 }
19 }

SEE ALSO
put(9E), srv(9E), flushq(9F), OTHERQ(9F), putnext(9F)

Writing Device Drivers
### qsize(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>qsize – find the number of messages on a queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/stream.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int qsize(queue_t *q);</code></td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td><code>q</code> Queue to be evaluated.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td><code>qsize()</code> evaluates the queue <code>q</code> and returns the number of messages it contains.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>If there are no message on the queue, <code>qsize()</code> returns 0. Otherwise, it returns the integer representing the number of messages on the queue.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td><code>qsize()</code> can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td></td>
<td>STREAMS Programming Guide</td>
</tr>
</tbody>
</table>

```c
#include <sys/stream.h>

int qsize(queue_t *q);
```
NAME
qtimeout – execute a function after a specified length of time

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

timeout_id_t qtimeout(queue_t *q, void *func, void *arg, clock_t ticks);

INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI).

q Pointer to STREAMS queue structure.
func Kernel function to invoke when the time increment expires.
arg Argument to the function.
ticks Number of clock ticks to wait before the function is called. Use
drv_usectohz(9F) to convert microseconds to clock ticks.

DESCRIPTION
The qtimeout() function schedules the specified function func to be called after a
specified time interval. func is called with arg as a parameter. Control is immediately
returned to the caller. This is useful when an event is known to occur within a specific
time frame, or when you want to wait for I/O processes when an interrupt is not
available or might cause problems. The exact time interval over which the timeout
takes effect cannot be guaranteed, but the value given is a close approximation.

The qtimeout() function is tailored to be used with the enhanced STREAMS
framework interface which is based on the concept of perimeters. (See
mt-streams(9F).) qtimeout() schedules the specified function to execute after
entering the perimeters associated with the queue passed in as the first parameter to
qtimeout(). All outstanding timeouts and bufcalls must be cancelled (using,
respectively, quntimeout(9F) and qunbufcall(9F)) before a driver close routine can
block and before the close routine calls qprocsoff(9F).

qprocson(9F) must be called before calling qtimeout().

RETURN VALUES
qtimeout() returns an opaque non-zero timeout identifier that can be passed to
quntimeout(9F) to cancel the request. Note: No value is returned from the called
function.

CONTEXT
qtimeout() can be called from user or interrupt context.

SEE ALSO
drv_usectohz(9F), mt-streams(9F), qbufcall(9F), qprocson(9F),
qubufcall(9F), quntimeout(9F)

Writing Device Drivers

STREAMS Programming Guide
qunbufcall(9F)

NAME
qunbufcall – cancel a pending qbufcall request

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

void qunbufcall(queue_t *q, bufcall_id_t id);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
q Pointer to STREAMS queue_t structure.

id Identifier returned from qbufcall(9F)

DESCRIPTION
qunbufcall() cancels a pending qbufcall() request. The argument id is a non-zero identifier of the request to be cancelled. id is returned from the qbufcall() function used to issue the cancel request.

The qunbufcall() function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See mt-streams(9F).) qunbufcall() returns when the bufcall has been cancelled or finished executing. The bufcall will be cancelled even if it is blocked at the perimeters associated with the queue. All outstanding timeouts and bufcalls must be cancelled before a driver close routine can block and before the close routine calls qprocsoff(9F).

CONTEXT
qunbufcall() can be called from user or interrupt context.

SEE ALSO
mt-streams(9F), qbufcall(9F), qtimeout(9F), quntimeout(9F)

Writing Device Drivers

STREAMS Programming Guide
quntimeout(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>quntimeout – cancel previous qtimeout function call</th>
</tr>
</thead>
</table>
| SYNOPSIS        | #include <sys/stream.h>  
|                 | #include <sys/ddi.h>    |
|                 | clock_t quntimeout(queue_t *q, timeout_id_t id); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| PARAMETERS      | q Pointer to a STREAMS queue structure. |
|                 | id Opaque timeout ID a previous qtimeout(9F) call. |
| DESCRIPTION     | quntimeout() cancels a pending qtimeout(9F) request. The quntimeout() function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See mt-streams(9F).) quntimeout() returns when the timeout has been cancelled or finished executing. The timeout will be cancelled even if it is blocked at the perimeters associated with the queue. quntimeout() should be executed for all outstanding timeouts before a driver or module close returns. All outstanding timeouts and bufcalls must be cancelled before a driver close routine can block and before the close routine calls qprocsoff(9F). |
| RETURN VALUES   | quntimeout() returns -1 if the id is not found. Otherwise, quntimeout() returns a 0 or positive value. |
| CONTEXT         | quntimeout() can be called from user or interrupt context. |
| SEE ALSO        | mt-streams(9F), qbufcall(9F), qtimeout(9F), qunbufcall(9F) |

Making Device Drivers

STREAMS Programming Guide
qwait(9F)

NAME
qwait, qwait_sig – STREAMS wait routines

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

void qwait(queue_t *q);
int qwait_sig(queue_t *q);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS
$qp$ Pointer to the queue that is being opened or closed.

DESCRIPTION
qwait() and qwait_sig() are used to wait for a message to arrive to the put(9E) or
srv(9E) procedures. qwait() and qwait_sig() can also be used to wait for
qbufcall(9F) or qtimeout(9F) callback procedures to execute. These routines can be
used in the open(9E) and close(9E) procedures in a STREAMS driver or module.
qwait() and qwait_sig() atomically exit the inner and outer perimeters associated
with the queue, and wait for a thread to leave the module's put(9E), srv(9E), or
qbufcall(9F) / qtimeout(9F) callback procedures. Upon return they re-enter the
inner and outer perimeters.

This can be viewed as there being an implicit wakeup when a thread leaves a put(9E)
or srv(9E) procedure or after a timeout(9F) or qbufcall(9F) callback procedure
has been run in the same perimeter.

qprocson(9F) must be called before calling qwait() or qwait_sig().

qwait() is not interrupted by a signal, whereas qwait_sig() is interrupted by a
signal. qwait_sig() normally returns non-zero, and returns zero when the waiting
was interrupted by a signal.

qwait() and qwait_sig() are similar to cv_wait() and cv_wait_sig() except
that the mutex is replaced by the inner and outer perimeters and the signalling is
implicit when a thread leaves the inner perimeter. See condvar(9F).

RETURN VALUES
0 For qwait_sig(), indicates that the condition was not necessarily
signaled, and the function returned because a signal was pending.

CONTEXT
These functions can only be called from an open(9E) or close(9E) routine.

EXAMPLES
EXAMPLE 1 Using qwait()

The open routine sends down a T_INFO_REQ message and waits for the
T_INFO_ACK. The arrival of the T_INFO_ACK is recorded by resetting a flag in the
unit structure (WAIT_INFO_ACK). The example assumes that the module is
D_MTQPAIR or D_MTPERMOD.

xxopen(qp, . . .)
{ queue_t *qp;
    struct xxdata *xx;
    /* Allocate xxdata structure */
EXAMPLE 1 Using qwait() (Continued)

qprocson(qp);
/* Format T_INFO_ACK in mp */
putnext(qp, mp);
xx->xx_flags |= WAIT_INFO_ACK;
while (xx->xx_flags & WAIT_INFO_ACK)
    qwait(qp);
return (0);
}

xxrput(qp, mp)
queue_t *qp;
mblk_t *mp;
{
    struct xxdata *xx = (struct xxdata *)q->q_ptr;

    ...

case T_INFO_ACK:
    if (xx->xx_flags & WAIT_INFO_ACK) {
        /* Record information from info ack */
        xx->xx_flags &= ~WAIT_INFO_ACK;
        freemsg(mp);
        return;
    }

    ...
}

SEE ALSO close(9E), open(9E), put(9E), srv(9E), condvar(9F), mt-streams(9F), qbufcall(9F), qprocson(9F), qtimeout(9F)

STREAMS Programming Guide
Writing Device Drivers
NAME | qwait, qwait_sig – STREAMS wait routines  
SYNOPSIS |  
#include <sys/stream.h>  
#include <sys/ddi.h>  

void qwait(queue_t *q);  
int qwait_sig(queue_t *q);  

INTERFACE | Solaris DDI specific (Solaris DDI).  
LEVEL |  
PARAMETERS | qp | Pointer to the queue that is being opened or closed.  
DESCRIPTION | qwait() and qwait_sig() are used to wait for a message to arrive to the put(9E) or srv(9E) procedures. qwait() and qwait_sig() can also be used to wait for qbufcall(9F) or qtimeout(9F) callback procedures to execute. These routines can be used in the open(9E) and close(9E) procedures in a STREAMS driver or module. qwait() and qwait_sig() atomically exit the inner and outer perimeters associated with the queue, and wait for a thread to leave the module’s put(9E), srv(9E), or qbufcall(9F) / qtimeout(9F) callback procedures. Upon return they re-enter the inner and outer perimeters.  
This can be viewed as there being an implicit wakeup when a thread leaves a put(9E) or srv(9E) procedure or after a qtimeout(9F) or qbufcall(9F) callback procedure has been run in the same perimeter.  
qprocson(9F) must be called before calling qwait() or qwait_sig().  
qwait() is not interrupted by a signal, whereas qwait_sig() is interrupted by a signal. qwait_sig() normally returns non-zero, and returns zero when the waiting was interrupted by a signal.  
qwait() and qwait_sig() are similar to cv_wait() and cv_wait_sig() except that the mutex is replaced by the inner and outer perimeters and the signalling is implicit when a thread leaves the inner perimeter. See condvar(9F).  
RETURN VALUES |  
0 | For qwait_sig(), indicates that the condition was not necessarily signaled, and the function returned because a signal was pending.  
CONTEXT | These functions can only be called from an open(9E) or close(9E) routine.  
EXAMPLES | EXAMPLE 1 Using qwait()  
The open routine sends down a T_INFO_REQ message and waits for the T_INFO_ACK. The arrival of the T_INFO_ACK is recorded by resetting a flag in the unit structure (WAIT_INFO_ACK). The example assumes that the module is D_MTQPAIR or D_MTPERMOD.  
  
xopen(qp, ...)  
queue_t *qp;  
{  
    struct xxdata *xx;  
    /* Allocate xxdata structure */
EXAMPLE 1 Using qwait()

Using qwait()

(Continued)

qprocson(qp);
/* Format T_INFO_ACK in mp */
putnext(qp, mp);
xx->xx_flags |= WAIT_INFO_ACK;
while (xx->xx_flags & WAIT_INFO_ACK)
    qwait(qp);
return (0);
}

xxrput(qp, mp)
queue_t *qp;
mblk_t *mp;
{
    struct xxdata *xx = (struct xxdata *)q->q_ptr;
    ...

    case T_INFO_ACK:
        if (xx->xx_flags & WAIT_INFO_ACK) {
            /* Record information from info ack */
            xx->xx_flags &-= WAIT_INFO_ACK;
            freemsg(mp);
            return;
        }
    ...
}

SEE ALSO

close(9E), open(9E), put(9E), srv(9E), condvar(9F), mt-streams(9F),
qbufcall(9F), qprocson(9F), qtimeout(9F)

STREAMS Programming Guide

Writing Device Drivers
NAME | qwriter – asynchronous STREAMS perimeter upgrade

SYNOPSIS | 

```c
#include <sys/stream.h>
#include <sys/ddi.h>

void qwriter(queue_t *qp, mblk_t *mp, void (*func)(), int perimeter);
```

INTERFACE LEVEL PARAMETERS | Solaris DDI specific (Solaris DDI).

- **qp**: Pointer to the queue.
- **mp**: Pointer to a message that will be passed in to the callback function.
- **func**: A function that will be called when exclusive (writer) access has been acquired at the specified perimeter.
- **perimeter**: Either PERIM_INNER or PERIM_OUTER.

DESCRIPTION | qwriter() is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive and call the specified callback function when the upgrade has succeeded. See mt-streams(9F). The callback function is called as:

```c
(*func)(queue_t *qp, mblk_t *mp);
```

qwriter() will acquire exclusive access immediately if possible, in which case the specified callback function will be executed before qwriter() returns. If this is not possible, qwriter() will defer the upgrade until later and return before the callback function has been executed. Modules should not assume that the callback function has been executed when qwriter() returns. One way to avoid dependencies on the execution of the callback function is to immediately return after calling qwriter() and let the callback function finish the processing of the message.

When qwriter() defers calling the callback function, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the upgrade has completed and the callback function has finished executing.

CONTEXT | qwriter() can only be called from a put(9E) or srv(9E) routine, or from a qwriter(), qtimeout(9F), or qbufcall(9F) callback function.

SEE ALSO | put(9E), srv(9E), mt-streams(9F), qbufcall(9F), qtimeout(9F)

STREAMS Programming Guide
Writing Device Drivers
RD(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>RD, rd – get pointer to the read queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>queue_t *RD(queue_t *q);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>PARAMETER</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>Pointer to the write queue whose read queue is to be returned.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>The RD() function accepts a write queue pointer as an argument and returns a pointer to the read queue of the same module.</td>
</tr>
<tr>
<td></td>
<td>CAUTION: Make sure the argument to this function is a pointer to a write queue. RD() will not check for queue type, and a system panic could result if it is not the right type.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The pointer to the read queue.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>RD() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>EXAMPLE 1 Function page reference</td>
</tr>
<tr>
<td></td>
<td>See the qreply(9F) function page for an example of RD().</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>qreply(9F), WR(9F)</td>
</tr>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td></td>
<td>STREAMS Programming Guide</td>
</tr>
</tbody>
</table>
NAME | RD, rd – get pointer to the read queue

SYNOPSIS | 
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *RD(queue_t *q);

INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI).
PARAMETERS | q Pointer to the write queue whose read queue is to be returned.

DESCRIPTION | 
The RD() function accepts a write queue pointer as an argument and returns a pointer to the read queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a write queue. RD() will not check for queue type, and a system panic could result if it is not the right type.

RETURN VALUES | The pointer to the read queue.

CONTEXT | RD() can be called from user or interrupt context.

EXAMPLES | EXAMPLE 1 Function page reference
See the qreply(9F) function page for an example of RD().

SEE ALSO | qreply(9F), WR(9F)

*Writing Device Drivers
STREAMS Programming Guide*
NAME
inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned char inb(int port);
unsigned short inw(int port);
unsigned long inl(int port);
void repinsb(int port, unsigned char *addr, int count);
void repinsw(int port, unsigned short *addr, int count);
void repinsd(int port, unsigned long *addr, int count);

INTERFACE
The functions described here are obsolete. For the inb(), inw(), and inl() functions, use, respectively, ddi_get8(9F), ddi_get16(9F), and ddi_get32(9F) instead. For repinsb(), repinsw(), and repinsd(), use, respectively, ddi_rep_get8(9F), ddi_rep_get16(9F), and ddi_rep_get32(9F) instead.

LEVEL
PARAMETERS
port A valid I/O port address.
addr The address of a buffer where the values will be stored.
count The number of values to be read from the I/O port.

DESCRIPTION
These routines read data of various sizes from the I/O port with the address specified by port.

The inb(), inw(), and inl() functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The repinsb(), repinsw(), and repinsd() functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold count values of the requested size.

RETURN VALUES
inb(), inw(), and inl() return the value that was read from the I/O port.

CONTEXT
These functions may be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
eisa(4), isa(4), attributes(5), ddi_get8(9F), ddi_get16(9F), ddi_get32(9F), ddi_rep_get8(9F), ddi_rep_get16(9F), ddi_rep_get32(9F), outb(9F)
The functions described here are obsolete. For the `inb()`, `inw()`, and `inl()` functions, use, respectively, `ddi_get8(9F)`, `ddi_get16(9F)`, and `ddi_get32(9F)` instead. For `repinsb()`, `repinsw()`, and `repinsd()`, use, respectively, `ddi_rep_get8(9F)`, `ddi_rep_get16(9F)`, and `ddi_rep_get32(9F)` instead.

These routines read data of various sizes from the I/O port with the address specified by `port`.

The `inb()`, `inw()`, and `inl()` functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The `repinsb()`, `repinsw()`, and `repinsd()` functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. `count` specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold `count` values of the requested size.

`inb()`, `inw()`, and `inl()` return the value that was read from the I/O port.

These functions may be called from user or interrupt context.

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

See also `eisa(4)`, `isa(4)`, `attributes(5)`, `ddi_get8(9F)`, `ddi_get16(9F)`, `ddi_get32(9F)`, `ddi_rep_get8(9F)`, `ddi_rep_get16(9F)`, `ddi_rep_get32(9F)`, `outb(9F)`
The functions described here are obsolete. For the `inb()`, `inw()`, and `inl()` functions, use, respectively, `ddi_get8(9F)`, `ddi_get16(9F)`, and `ddi_get32(9F)` instead. For `repinsb()`, `repinsw()`, and `repinsd()`, use, respectively, `ddi_rep_get8(9F)`, `ddi_rep_get16(9F)`, and `ddi_rep_get32(9F)` instead.

**PARAMETERS**
- `port` A valid I/O port address.
- `addr` The address of a buffer where the values will be stored.
- `count` The number of values to be read from the I/O port.

**RETURN VALUES**
- `inb()`, `inw()`, and `inl()` return the value that was read from the I/O port.

**CONTEXT**
These functions may be called from user or interrupt context.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**
eisa(4), isa(4), attributes(5), ddi_get8(9F), ddi_get16(9F), ddi_get32(9F), ddi_rep_get8(9F), ddi_rep_get16(9F), ddi_rep_get32(9F), outb(9F)
NAME | outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);
```

INTERFACE LEVEL

The functions described here are obsolete. For the `outb()`, `outw()`, and `outl()` functions use, respectively, `ddi_put8(9F)`, `ddi_put16(9F)`, and `ddi_put32(9F)` instead. For `repoutsb()`, `repoutsw()`, and `repoutsd()`, use, respectively, `ddi_rep_put8(9F)`, `ddi_rep_put16(9F)`, and `ddi_rep_put32(9F)` instead.

PARAMETERS

- `port` | A valid I/O port address.
- `value` | The data to be written to the I/O port.
- `addr` | The address of a buffer from which the values will be fetched.
- `count` | The number of values to be written.

DESCRIPTION

These routines write data of various sizes to the I/O port with the address specified by `port`.

The `outb()`, `outw()`, and `outl()` functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by `value`.

The `repoutsb()`, `repoutsw()`, and `repoutsd()` functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. `count` specifies the number of values to be written. `addr` is a pointer to a buffer from which the output values are fetched.

CONTEXT

These functions may be called from user or interrupt context.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO | eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)

Writing Device Drivers
outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

#include <sys/ddi.h>
#include <sys/sunddi.h>

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);

The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsd(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.

PARAMETERS
port A valid I/O port address.
value The data to be written to the I/O port.
addr The address of a buffer from which the values will be fetched.
count The number of values to be written.

DESCRIPTION
These routines write data of various sizes to the I/O port with the address specified by port.

The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.

The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

CONTEXT
These functions may be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE | ATTRIBUTE VALUE
---|---
Architecture | IA
Stability Level | Obsolete

SEE ALSO
elsa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)

Writing Device Drivers
repoutsw(9F)

NAME
outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);

INTERFACE
The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsd(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.

LEVEL

PARAMETERS
port A valid I/O port address.
value The data to be written to the I/O port.
addr The address of a buffer from which the values will be fetched.
count The number of values to be written.

DESCRIPTION
These routines write data of various sizes to the I/O port with the address specified by port.

The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.

The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

CONTEXT
These functions may be called from user or interrupt context.

ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IA</td>
</tr>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)

Writing Device Drivers
rmalloc(9F)

NAME
rmalloc – allocate space from a resource map

SYNOPSIS
#include <sys/map.h>
#include <sys/ddi.h>

unsigned long rmalloc(struct map *mp, size_t size);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
mp Resource map from where the resource is drawn.
size Number of units of the resource.

DESCRIPTION
rmalloc() is used by a driver to allocate space from a previously defined and initialized resource map. The map itself is allocated by calling the function rmallocmap(9F). rmalloc() is one of five functions used for resource map management. The other functions include:

rmalloc_wait(9F) Allocate space from a resource map, wait if necessary.
rmfree(9F) Return previously allocated space to a map.
rmallocmap(9F) Allocate a resource map and initialize it.
rmfreemap(9F) Deallocate a resource map.

rmalloc() allocates space from a resource map in terms of arbitrary units. The system maintains the resource map by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. The normal return value is an unsigned long set to the value of the index where sufficient free space in the resource was found.

RETURN VALUES
Under normal conditions, rmalloc() returns the base index of the allocated space. Otherwise, rmalloc() returns a 0 if all resource map entries are already allocated.

CONTEXT
rmalloc() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Illustrating the principles of map management

The following example is a simple memory map, but it illustrates the principles of map management. A driver allocates and initializes the map by calling both the rmallocmap(9F) and rmfree(9F) functions. rmallocmap(9F) is called to establish the number of slots or entries in the map, and rmfree(9F) to initialize the resource area the map is to manage. The following example is a fragment from a hypothetical start routine and illustrates the following procedures:

- Panics the system if the required amount of memory can not be allocated (lines 11–15).
- Uses rmallocmap(9F) to configure the total number of entries in the map, and rmfree(9F) to initialize the total resource area.

1 #define XX_MAPSIZE 12
2 #define XX_BUFSIZE 2560
3 static struct map *xx_mp; /* Private buffer space map */
EXAMPLE 1 Illustrating the principles of map management (Continued)

```c
4  xxstart( )
5  /*
6   * Allocate private buffer. If insufficient memory,
7   * display message and halt system.
8   */
9  {
10   register caddr_t bp;
11   if ((bp = kmem_alloc(XX_BUFSIZE, KM_NOSLEEP) == 0) {
12     cmn_err(CE_PANIC, "xxstart: kmem_alloc failed before %d buffer"
13              "allocation", XX_BUFSIZE);
14   }
15   /*
16    * Initialize the resource map with number
17    * of slots in map.
18    */
19   xx_mp = rmallocmap(XX_MAPSIZE);
20   /*
21    * Initialize space management map with total
22    * buffer area it is to manage.
23    */
24   rmfree(xx_mp, XX_BUFSIZE, bp);
25   ...
```

EXAMPLE 2 Allocating buffers

The `rmalloc()` function is then used by the driver's read or write routine to allocate buffers for specific data transfers. The `uiomove(9F)` function is used to move the data between user space and local driver memory. The device then moves data between itself and local driver memory through DMA.

The next example illustrates the following procedures:

- The size of the I/O request is calculated and stored in the `size` variable (line 10).
- Buffers are allocated through the `rmalloc()` function using the `size` value (line 15).
- If the allocation fails the system will panic.
- The `uiomove(9F)` function is used to move data to the allocated buffer (line 23).
- If the address passed to `uiomove(9F)` is invalid, `rmfree(9F)` is called to release the previously allocated buffer, and an `EFAULT` error is returned.

```c
1 #define XX_BUFSIZE 2560
2 #define XX_MAXSIZE (XX_BUFSIZE / 4)
3
4 static struct map *xx_mp; /* Private buffer space map */
5 ...
6 xxread(dev_t dev, uiot_t *uiop, cred_t *credp)
```
EXAMPLE 2 Allocating buffers (Continued)

```c
7 register caddr_t addr;
8 register int size;
9
10 size = min(COUNT, XX_MAXSIZE); /* Break large I/O request */
11 /* into small ones */
12 /*
13 * Get buffer.
14 */
15 if ((addr = (caddr_t)rmalloc(xx_mp, size)) == 0)
16    cmn_err(CE_PANIC, "read: rmalloc failed allocation of size %d", 
17             size);
18
19 /*
20 * Move data to buffer. If invalid address is found,
21 * return buffer to map and return error code.
22 */
23 if (uiomove(addr, size, UIO_READ, uiop) == -1) {
24    rmfree(xx_mp, size, addr);
25    return(EFAULT);
26 }
27 }
```

SEE ALSO kmem_alloc(9F), rmalloc_wait(9F), rmallocmap(9F), rmfree(9F), 
rmfreemap(9F), uiomove(9F)

Writing Device Drivers
rmallocmap(9F)

NAME  
rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);

INTERFACE LEVEL
Parameters

PARAMETERS

DESCRIPTION  
rmallocmap() dynamically allocates a resource map structure. The argument mapsize defines the total number of entries in the map. In particular, it is the total number of allocations that can be outstanding at any one time.

rmallocmap() initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource, a call to rmfree(9F) is used to make the entirety of the actual resource available for allocation, starting from the first index into the resource. Typically, the call to rmallocmap() is followed by a call to rmfree(9F), passing the address of the map returned from rmallocmap(), the total size of the resource, and the first index into the actual resource.

The resource map allocated by rmallocmap() can be used to describe an arbitrary resource in whatever allocation units are appropriate, such as blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to rmalloc(9F), rmalloc_wait(9F), and rmfree(9F).

rmallocmap_wait() is similar to rmallocmap(), with the exception that it will wait for space to become available if necessary.

rmfreemap() deallocates a resource map structure previously allocated by rmallocmap() or rmallocmap_wait(). The argument mp is a pointer to the map structure to be deallocated.

RETURN VALUES
Upon successful completion, rmallocmap() and rmallocmap_wait() return a pointer to the newly allocated map structure. Upon failure, rmallocmap() returns a NULL pointer.

CONTEXT  
rmallocmap() and rmfreemap() can be called from user, kernel, or interrupt context.

rmallocmap_wait() can only be called from user or kernel context.

SEE ALSO  
rmalloc(9F), rmalloc_wait(9F), rmfree(9F)

Writing Device Drivers
### NAME
rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps

### SYNOPSIS
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- **mapsize**: Number of entries for the map.
- **mp**: A pointer to the map structure to be deallocated.

### DESCRIPTION
rmallocmap() dynamically allocates a resource map structure. The argument `mapsize` defines the total number of entries in the map. In particular, it is the total number of allocations that can be outstanding at any one time.

rmallocmap() initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource, a call to `rmfree(9F)` is used to make the entirety of the actual resource available for allocation, starting from the first index into the resource. Typically, the call to `rmallocmap()` is followed by a call to `rmfree(9F)`, passing the address of the map returned from `rmallocmap()`, the total size of the resource, and the first index into the actual resource.

The resource map allocated by `rmallocmap()` can be used to describe an arbitrary resource in whatever allocation units are appropriate, such as blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to `rmalloc(9F)`, `rmalloc_wait(9F)`, and `rmfree(9F)`.

rmallocmap_wait() is similar to `rmallocmap()`, with the exception that it will wait for space to become available if necessary.

rmfreemap() deallocates a resource map structure previously allocated by `rmallocmap()` or `rmallocmap_wait()`. The argument `mp` is a pointer to the map structure to be deallocated.

### RETURN VALUES
Upon successful completion, `rmallocmap()` and `rmallocmap_wait()` return a pointer to the newly allocated map structure. Upon failure, `rmallocmap()` returns a NULL pointer.

### CONTEXT
`rmallocmap()` and `rmfreemap()` can be called from user, kernel, or interrupt context.

`rmallocmap_wait()` can only be called from user or kernel context.

### SEE ALSO
`rmalloc(9F), rmalloc_wait(9F), rmfree(9F)`

*Writing Device Drivers*
NAME
rmalloc_wait – allocate space from a resource map, wait if necessary

SYNOPSIS
#include <sys/map.h>
#include <sys/ddi.h>

unsigned long rmalloc_wait(struct map *mp, size_t size);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp</td>
<td>Pointer to the resource map from which space is to be allocated.</td>
</tr>
<tr>
<td>size</td>
<td>Number of units of space to allocate.</td>
</tr>
</tbody>
</table>

DESCRIPTION
rmalloc_wait() requests an allocation of space from a resource map.
rmalloc_wait() is similar to the rmalloc(9F) function with the exception that it will wait for space to become available if necessary.

RETURN VALUES
rmalloc_wait() returns the base of the allocated space.

CONTEXT
This function can be called from user or interrupt context. However, in most cases rmalloc_wait() should be called from user context only.

SEE ALSO
rmalloc(9F), rmallocmap(9F), rmfree(9F), rmfreemap(9F)

Writing Device Drivers
### NAME
rmfree – free space back into a resource map

### SYNOPSIS
```c
#include <sys/map.h>
#include <sys/ddi.h>

void rmfree(struct map *mp, size_t size, ulong_t index);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- **mp** Pointer to the map structure.
- **size** Number of units being freed.
- **index** Index of the first unit of the allocated resource.

### DESCRIPTION
rmfree() releases space back into a resource map. It is the opposite of rmalloc(9F), which allocates space that is controlled by a resource map structure.

When releasing resources using rmfree() the size and index passed to rmfree() must exactly match the size and index values passed to and returned from a previous call to rmalloc(). Resources cannot be returned piecemeal.

Drivers may define resource maps for resource allocation, in terms of arbitrary units, using the rmallocmap(9F) function. The system maintains the resource map structure by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. rmfree() frees up unallocated space for re-use.

rmfree() can also be used to initialize a resource map, in which case the size and index should cover the entire resource area.

### CONTEXT
rmfree() can be called from user or interrupt context.

### SEE ALSO
rmalloc(9F), rmalloc_wait(9F), rmallocmap(9F), rmfreemap(9F)

Writing Device Drivers
rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps

#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);

NAME
rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
mapsize Number of entries for the map.
mp A pointer to the map structure to be deallocated.

DESCRIPTION
rmallocmap() dynamically allocates a resource map structure. The argument mapsize defines the total number of entries in the map. In particular, it is the total number of allocations that can be outstanding at any one time.

rmallocmap() initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource, a call to rmfree(9F) is used to make the entirety of the actual resource available for allocation, starting from the first index into the resource. Typically, the call to rmallocmap() is followed by a call to rmfree(9F), passing the address of the map returned from rmallocmap(), the total size of the resource, and the first index into the actual resource.

The resource map allocated by rmallocmap() can be used to describe an arbitrary resource in whatever allocation units are appropriate, such as blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to rmalloc(9F), rmalloc_wait(9F), and rmfree(9F).

rmallocmap_wait() is similar to rmallocmap(), with the exception that it will wait for space to become available if necessary.

rmfreemap() deallocates a resource map structure previously allocated by rmallocmap() or rmallocmap_wait(). The argument mp is a pointer to the map structure to be deallocated.

RETURN VALUES
Upon successful completion, rmallocmap() and rmallocmap_wait() return a pointer to the newly allocated map structure. Upon failure, rmallocmap() returns a NULL pointer.

CONTEXT
rmallocmap() and rmfreemap() can be called from user, kernel, or interrupt context.
rmallocmap_wait() can only be called from user or kernel context.

SEE ALSO
rmalloc(9F), rmalloc_wait(9F), rmfree(9F)

Writing Device Drivers
rmvb(9F)

NAME  rmvb – remove a message block from a message

SYNOPSIS  
#include <sys/stream.h>

mblk_t *rmvb(mblk_t *mp, mblk_t *bp);

INTERFACE LEVEL
PARAMETERS

Architecture independent level 1 (DDI/DKI).

mp  Message from which a block is to be removed. mblk_t is an instance of the
    msgb(9S) structure.

bp  Message block to be removed.

DESCRIPTION  
rmvb() removes a message block (bp) from a message (mp), and returns a pointer to
the altered message. The message block is not freed, merely removed from the
message. It is the module or driver’s responsibility to free the message block.

RETURN VALUES  
If successful, a pointer to the message (minus the removed block) is returned. The
pointer is NULL if bp was the only block of the message before rmvb() was called. If
the designated message block (bp) does not exist, -1 is returned.

CONTEXT  
rmvb() can be called from user or interrupt context.

EXAMPLES  
This routine removes all zero-length M_DATA message blocks from the given message.
For each message block in the message, save the next message block (line 10). If the
current message block is of type M_DATA and has no data in its buffer (line 11), then
remove it from the message (line 12) and free it (line 13). In either case, continue with
the next message block in the message (line 16).

1  void
2  xxclean(mp)
3  mblk_t *mp;
4  {
5      mblk_t *tmp;
6      mblk_t *nmp;
7      
8      tmp = mp;
9      while (tmp) {
10         nmp = tmp->b_cont;
11         if (((tmp->b_datap->db_type == M_DATA) &&
12             (tmp->b_rptr == tmp->b_wptr)) {
13             (void) rmvb(mp, tmp);
14         }
15         tmp = nmp;
16     }
17 }

SEE ALSO  
freeb(9F), msgb(9S)

Writing Device Drivers

STREAMS Programming Guide
NAME | rmvq – remove a message from a queue

SYNOPSIS | 
#include <sys/stream.h> 

void rmvq(queue_t *q, mblk_t *mp);

INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI).
PARAMETERS | 
q | Queue containing the message to be removed.
mp | Message to remove.

DESCRIPTION | rmvq() removes a message from a queue. A message can be removed from anywhere on a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either rmvq() or getq(9F) should be used to remove a message from a queue.

CONTEXT | rmvq() can be called from user or interrupt context.

EXAMPLES | This code fragment illustrates how one may flush one type of message from a queue. In this case, only M_PROTO T_DATA_IND messages are flushed. For each message on the queue, if it is an M_PROTO message (line 8) of type T_DATA_IND (line 10), save a pointer to the next message (line 11), remove the T_DATA_IND message (line 12) and free it (line 13). Continue with the next message in the list (line 19).

```c
1 mblk_t *mp, *nmp;
2 queue_t *q;
3 union T_primitives *tp;
4
5 /* Insert code here to protect queue and message block */
6 mp = q->q_first;
7 while (mp) {
8     if (mp->b_datap->db_type == M_PROTO) {
9         tp = (union T_primitives *)mp->b_rptr;
10         if (tp->type == T_DATA_IND) {
11             nmp = mp->b_next;
12             rmvq(q, mp);
13             freemsg(mp);
14             mp = nmp;
15         } else {
16             mp = mp->b_next;
17         }
18     } else {
19         mp = mp->b_next;
20     }
21 }
22 /* End of region that must be protected */
```

When using rmvq(), you must ensure that the queue and the message block is not modified by another thread at the same time. You can achieve this either by using STREAMS functions or by implementing your own locking.

SEE ALSO | freemsg(9F), getq(9F), insq(9F)

| Writing Device Drivers |
WARNINGS
Make sure that the message \textit{mp} is linked onto \textit{q} to avoid a possible system panic.
rw_init (9F)

rw_lock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade,
rw_tryupgrade, rw_read_locked – readers/writer lock functions

#include <sys/ksynch.h>

void rw_init (krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy (krwlock_t *rwlp);
void rw_enter (krwlock_t *rwlp, krw_t enter_type);
void rw_exit (krwlock_t *rwlp);
int rw_tryenter (krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade (krwlock_t *rwlp);
int rw_tryupgrade (krwlock_t *rwlp);
int rw_read_locked (krwlock_t *rwlp);

Solaris DDI specific (Solaris DDI).

rwlp Pointer to a krwlock_t readers/writer lock.
name Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)
type Type of readers/writer lock.
arg Type-specific argument for initialization function.
enter_type One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rw_tryenter() could not obtain the lock without blocking.</td>
</tr>
<tr>
<td>0</td>
<td>rw_tryupgrade() was unable to perform the upgrade because of other threads holding or waiting to hold the lock.</td>
</tr>
<tr>
<td>0</td>
<td>rw_read_locked() returns 0 if the lock is held by the caller for write.</td>
</tr>
<tr>
<td>non-zero</td>
<td>from rw_read_locked() if the lock is held by the caller for read.</td>
</tr>
<tr>
<td>non-zero</td>
<td>successful return from rw_tryenter() or rw_tryupgrade().</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
rw_downgrade(9F)

NAME  rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions

SYNOPSIS  
#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
rwlp  Pointer to a krwlock_t readers/writer lock.
name  Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)
type  Type of readers/writer lock.
arg  Type-specific argument for initialization function.
enter_type  One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

DESCRIPTION
A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
**rw_destroy()** releases any resources that might have been allocated by **rw_init().** It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

**rw_enter()** acquires the lock, and blocks if necessary. If **enter_type** is **RW_READER**, the caller blocks if there is a writer or a thread attempting to enter for writing. If **enter_type** is **RW_WRITER**, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

**rw_exit()** releases the lock and may wake up one or more threads waiting on the lock.

**rw_tryenter()** attempts to enter the lock, like **rw_enter()**, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with **RW_WRITER**), may call **rw_downgrade()** to convert to holding the lock non-exclusively (as if entered with **RW_READER**). One or more waiting readers may be unblocked.

**rw_tryupgrade()** can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

**rw_read_locked()** returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if **rw_read_locked()** is called for a lock that isn’t held by the caller.

**RETURN VALUES**

- 0: **rw_tryenter()** could not obtain the lock without blocking.
- 0: **rw_tryupgrade()** was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
- 0: **rw_read_locked()** returns 0 if the lock is held by the caller for write.
- non-zero: from **rw_read_locked()** if the lock is held by the caller for read.
- non-zero: successful return from **rw_tryenter()** or **rw_tryupgrade()**.

**CONTEXT** These functions can be called from user or interrupt context, except for **rw_init()** and **rw_destroy()**, which can be called from user context only.
rw_downgrade(9F)

SEE ALSO  condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES  Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade,
rw_tryupgrade, rw_read_locked – readers/writer lock functions

#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

Solaris DDI specific (Solaris DDI).

<table>
<thead>
<tr>
<th>NAME</th>
<th>rwlp</th>
<th>Pointer to a krwlock_t readers/writer lock.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>Type of readers/writer lock.</td>
</tr>
<tr>
<td></td>
<td>arg</td>
<td>Type-specific argument for initialization function.</td>
</tr>
<tr>
<td></td>
<td>enter_type</td>
<td>One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).</td>
</tr>
</tbody>
</table>

A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>non-zero</td>
</tr>
<tr>
<td>non-zero</td>
</tr>
</tbody>
</table>

CONTEXT

These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO: condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES: Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
rw_init(9F)

NAME rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions

SYNOPSIS #include <sys/ksynch.h>
void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE Solaris DDI specific (Solaris DDI).
LEVEL PARAMETERS
PARAMETERS
rwlp Pointer to a krwlock_t readers/writer lock.
name Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)
type Type of readers/writer lock.
arg Type-specific argument for initialization function.
enter_type One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

DESCRIPTION A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

RETURN VALUES
0 rw_tryenter() could not obtain the lock without blocking.
0 rw_tryupgrade() was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
0 rw_read_locked() returns 0 if the lock is held by the caller for write.
non-zero from rw_read_locked() if the lock is held by the caller for read.
non-zero successful return from rw_tryenter() or rw_tryupgrade().

CONTEXT These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
rw_init
rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade,
rw_tryupgrade, rw_read_locked – readers/writer lock functions

#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

rwlp Pointer to a krwlock_t readers/writer lock.
name Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they're a waste of kernel memory.)
type Type of readers/writer lock.
arg Type-specific argument for initialization function.
enter_type One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

DESCRIPTION

A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
**rw_destroy()** releases any resources that might have been allocated by **rw_init()**. It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

**rw_enter()** acquires the lock, and blocks if necessary. If **enter_type** is **RW_READER**, the caller blocks if there is a writer or a thread attempting to enter for writing. If **enter_type** is **RW_WRITER**, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

**rw_exit()** releases the lock and may wake up one or more threads waiting on the lock.

**rw_tryenter()** attempts to enter the lock, like **rw_enter()**, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with **RW_WRITER**), may call **rw_downgrade()** to convert to holding the lock non-exclusively (as if entered with **RW_READER**). One or more waiting readers may be unblocked.

**rw_tryupgrade()** can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

**rw_read_locked()** returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if **rw_read_locked()** is called for a lock that isn't held by the caller.

### RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>rw_tryenter()</strong> could not obtain the lock without blocking.</td>
</tr>
<tr>
<td>0</td>
<td><strong>rw_tryupgrade()</strong> was unable to perform the upgrade because of other threads holding or waiting to hold the lock.</td>
</tr>
<tr>
<td>0</td>
<td><strong>rw_read_locked()</strong> returns 0 if the lock is held by the caller for write.</td>
</tr>
<tr>
<td>non-zero</td>
<td>from <strong>rw_read_locked()</strong> if the lock is held by the caller for read.</td>
</tr>
<tr>
<td>non-zero</td>
<td>successful return from <strong>rw_tryenter()</strong> or <strong>rw_tryupgrade()</strong>.</td>
</tr>
</tbody>
</table>

### CONTEXT

These functions can be called from user or interrupt context, except for **rw_init()** and **rw_destroy()**, which can be called from user context only.
rw_init(9F)

SEE ALSO condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
rwlock(9F)

NAME    rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions

SYNOPSIS    #include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE LEVEL PARAMETERS    Solaris DDI specific (Solaris DDI).

rwlp    Pointer to a krwlock_t readers/writer lock.
name    Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)
type    Type of readers/writer lock.
arg    Type-specific argument for initialization function.
enter_type    One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

DESCRIPTION    A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

RETURN VALUES
0    rw_tryenter() could not obtain the lock without blocking.
0    rw_tryupgrade() was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
0    rw_read_locked() returns 0 if the lock is held by the caller for write.

non-zero   from rw_read_locked() if the lock is held by the caller for read.
non-zero   successful return from rw_tryenter() or rw_tryupgrade().

CONTEXT These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO

condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES

Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions

#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);

void rw_destroy(krwlock_t *rwlp);

void rw_enter(krwlock_t *rwlp, krw_t enter_type);

void rw_exit(krwlock_t *rwlp);

int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);

void rw_downgrade(krwlock_t *rwlp);

int rw_tryupgrade(krwlock_t *rwlp);

int rw_read_locked(krwlock_t *rwlp);

Solaris DDI specific (Solaris DDI).

rwlp Pointer to a krwlock_t readers/writer lock.

name Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they’re a waste of kernel memory.)

type Type of readers/writer lock.

arg Type-specific argument for initialization function.

enter_type One of the values RW_READER or RW_WRITER, indicating whether the lock is to be acquired non-exclusively (RW_READER) or exclusively (RW_WRITER).

A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rw_tryenter() could not obtain the lock without blocking.</td>
</tr>
<tr>
<td>0</td>
<td>rw_tryupgrade() was unable to perform the upgrade because of other threads holding or waiting to hold the lock.</td>
</tr>
<tr>
<td>0</td>
<td>rw_read_locked() returns 0 if the lock is held by the caller for write.</td>
</tr>
<tr>
<td>non-zero</td>
<td>from rw_read_locked() if the lock is held by the caller for read.</td>
</tr>
<tr>
<td>non-zero</td>
<td>successful return from rw_tryenter() or rw_tryupgrade().</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO：condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES：Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
rw_tryenter(9F)

NAME
rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions

SYNOPSIS
#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS

type
- Type of readers/writer lock.

DESCRIPTION

A multiple-readers, single-writer lock is represented by the krwlock_t data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than once. The type argument should be set to RW_DRIVER. If the lock is used by the interrupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not used by any interrupt handler, the argument should be NULL.
rw_destroy() releases any resources that might have been allocated by rw_init(). It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is RW_WRITER, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

rw_exit() releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter() attempts to enter the lock, like rw_enter(), but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with RW_WRITER), may call rw_downgrade() to convert to holding the lock non-exclusively (as if entered with RW_READER). One or more waiting readers may be unblocked.

rw_tryupgrade() can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked() returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if rw_read_locked() is called for a lock that isn’t held by the caller.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rw_tryenter() could not obtain the lock without blocking.</td>
</tr>
<tr>
<td>0</td>
<td>rw_tryupgrade() was unable to perform the upgrade because of other threads holding or waiting to hold the lock.</td>
</tr>
<tr>
<td>0</td>
<td>rw_read_locked() returns 0 if the lock is held by the caller for write.</td>
</tr>
<tr>
<td>non-zero</td>
<td>from rw_read_locked() if the lock is held by the caller for read.</td>
</tr>
<tr>
<td>non-zero</td>
<td>successful return from rw_tryenter() or rw_tryupgrade().</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context, except for rw_init() and rw_destroy(), which can be called from user context only.
SEE ALSO  condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
- **rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade,**
  - **rw_tryupgrade, rw_read_locked** – readers/writer lock functions

```c
#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);
```

**NAME**
- `rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked` – readers/writer lock functions

**SYNOPSIS**
- `rw_init`, `rw_destroy`, `rw_enter`, `rw_exit`, `rw_tryenter`, `rw_downgrade`, `rw_tryupgrade`, `rw_read_locked`

**INTERFACE LEVEL PARAMETERS**
- `rwlp` – Pointer to a `krwlock_t` readers/writer lock.
- `name` – Descriptive string. This is obsolete and should be `NULL`. (Non-null strings are legal, but they’re a waste of kernel memory.)
- `type` – Type of readers/writer lock.
- `arg` – Type-specific argument for initialization function.
- `enter_type` – One of the values `RW_READER` or `RW_WRITER`, indicating whether the lock is to be acquired non-exclusively (`RW_READER`) or exclusively (`RW_WRITER`).

**DESCRIPTION**
- A multiple-readers, single-writer lock is represented by the `krwlock_t` data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

- `rw_init()` initializes a readers/writer lock. It is an error to initialize a lock more than once. The `type` argument should be set to `RW_DRIVER`. If the lock is used by the interrupt handler, the type-specific argument, `arg`, should be the `ddi_iblock_cookie` returned from `ddi_get_iblock_cookie(9F)` or `ddi_get_soft_iblock_cookie(9F)`. If the lock is not used by any interrupt handler, the argument should be `NULL`.

**Kernel Functions for Drivers** 1377
**rw_destroy()** releases any resources that might have been allocated by **rw_init()**. It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

**rw_enter()** acquires the lock, and blocks if necessary. If **enter_type** is **RW_READER**, the caller blocks if there is a writer or a thread attempting to enter for writing. If **enter_type** is **RW_WRITER**, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

**rw_exit()** releases the lock and may wake up one or more threads waiting on the lock.

**rw_tryenter()** attempts to enter the lock, like **rw_enter()**, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with **RW_WRITER**), may call **rw_downgrade()** to convert to holding the lock non-exclusively (as if entered with **RW_READER**). One or more waiting readers may be unblocked.

**rw_tryupgrade()** can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

**rw_read_locked()** returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if **rw_read_locked()** is called for a lock that isn't held by the caller.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>rw_tryenter()</strong> could not obtain the lock without blocking.</td>
</tr>
<tr>
<td>0</td>
<td><strong>rw_tryupgrade()</strong> was unable to perform the upgrade because of other threads holding or waiting to hold the lock.</td>
</tr>
<tr>
<td>0</td>
<td><strong>rw_read_locked()</strong> returns 0 if the lock is held by the caller for write.</td>
</tr>
<tr>
<td>non-zero</td>
<td>from <strong>rw_read_locked()</strong> if the lock is held by the caller for read.</td>
</tr>
<tr>
<td>non-zero</td>
<td>successful return from <strong>rw_tryenter()</strong> or <strong>rw_tryupgrade()</strong>.</td>
</tr>
</tbody>
</table>

**CONTEXT**

These functions can be called from user or interrupt context, except for **rw_init()** and **rw_destroy()**, which can be called from user context only.
SEE ALSO
condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F),
ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)

Writing Device Drivers

NOTES
Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather
lock statistics, see lockstat(1M).
SAMESTR(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>SAMESTR, samestr – test if next queue is in the same stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int SAMESTR(queue_t *q);</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>q Pointer to the queue.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>The SAMESTR() function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This function accounts for the twisted queue connections that occur in a STREAMS pipe and should be used in preference to direct examination of the q_next field of queue(9S) to see if the stream continues beyond q.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>SAMESTR() returns 1 if the next queue is the same type as the current queue. It returns 0 if the next queue does not exist or if it is not the same type.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>SAMESTR() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>OTHERQ(9F)</td>
</tr>
</tbody>
</table>

Writing Device Drivers

STREAMS Programming Guide
### NAME
SAMESTR, samestr – test if next queue is in the same stream

### SYNOPSIS
```
#include <sys/stream.h>

int SAMESTR(queue_t *q);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### PARAMETERS
- `q` Pointer to the queue.

### DESCRIPTION
The `SAMESTR()` function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This function accounts for the twisted queue connections that occur in a STREAMS pipe and should be used in preference to direct examination of the `q_next` field of `queue(9S)` to see if the stream continues beyond `q`.

### RETURN VALUES
- `SAMESTR()` returns 1 if the next queue is the same type as the current queue. It returns 0 if the next queue does not exist or if it is not the same type.

### CONTEXT
`SAMESTR()` can be called from user or interrupt context.

### SEE ALSO
- OTHERQ(9F)
- *Writing Device Drivers*
- *STREAMS Programming Guide*
### NAME
scsi_abort – abort a SCSI command

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

int scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **ap**: Pointer to a `scsi_address` structure.
- **pkt**: Pointer to a `scsi_pkt(9S)` structure.

### DESCRIPTION
`scsi_abort()` terminates a command that has been transported to the host adapter driver. A NULL `pkt` causes all outstanding packets to be aborted. On a successful abort, the `pkt_reason` is set to `CMD_ABORTED` and `pkt_statistics` is OR'ed with `STAT_ABORTED`.

### RETURN VALUES
`scsi_abort()` returns:
- **1**: on success.
- **0**: on failure.

### CONTEXT
`scsi_abort()` can be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1** Terminating a command.
```
if (scsi_abort(&devp->sd_address, pkt) == 0) {
    (void) scsi_reset(&devp->sd_address, RESET_ALL);
}
```

### SEE ALSO
tran_abort(9E), scsi_reset(9F), scsi_pkt(9S)

*Writing Device Drivers*
scsi_alloc_consistent_buf - allocate an I/O buffer for SCSI DMA

**SYNOPSIS**

```c
#include <sys/scsi/scsi.h>

struct buf *scsi_alloc_consistent_buf(struct scsi_address *ap,
    struct buf *bp, size_t datalen, uint_t bflags, int (*callback,
    caddr_t), caddr_t arg);
```

**INTERFACE LEVEL PARAMETERS**

- **ap**
  - Pointer to the `scsi_address(9S)` structure.
- **bp**
  - Pointer to the `buf(9S)` structure.
- **datalen**
  - Number of bytes for the data buffer.
- **bflags**
  - Flags setting for the allocated buffer header. This should either be `B_READ` or `B_WRITE`.
- **callback**
  - A pointer to a callback function, `NULL_FUNC` or `SLEEP_FUNC`.
- **arg**
  - The callback function argument.

**DESCRIPTION**

`scsi_alloc_consistent_buf()` allocates a buffer header and the associated data buffer for direct memory access (DMA) transfer. This buffer is allocated from the `iobp` space, which is considered consistent memory. For more details, see `ddi_dma_mem_alloc(9F)` and `ddi_dma_sync(9F)`.

For buffers allocated via `scsi_alloc_consistent_buf()`, and marked with the `PKT_CONSISTENT` flag via `scsi_init_pkt(9F)`, the HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver's command completion callback is performed.

If `bp` is `NULL`, a new buffer header will be allocated using `getrbuf(9F)`. In addition, if `datalen` is non-zero, a new buffer will be allocated using `ddi_dma_mem_alloc(9F)`.

**callback** indicates what the allocator routines should do when direct memory access (DMA) resources are not available; the valid values are:

- **NULL_FUNC**
  - Do not wait for resources. Return a `NULL` pointer.
- **SLEEP_FUNC**
  - Wait indefinitely for resources.
- **Other Values**
  - `callback` points to a function that is called when resources may become available. `callback` must return either `0` (indicating that it attempted to allocate resources but failed to do so), in which case it is put back on a list to be called again later, or `1` indicating either success in allocating resources or indicating that it no longer cares for a retry. The last argument `arg` is supplied to the `callback` function when it is invoked.
scsi_alloc_consistent_buf(9F)

RETURN VALUES  
scsi_alloc_consistent_buf() returns a pointer to a buf(9S) structure on success. It returns NULL if resources are not available even if waitfunc was not SLEEP_FUNC.

CONTEXT  
If callback is SLEEP_FUNC, then this routine may be called only from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block.

EXAMPLES  
**EXAMPLE 1** Allocate a request sense packet with consistent DMA resources attached.

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
       SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
rgpkt = scsi_init_pkt(&devp->sd_address,
            NULL, bp, CDB_GROUP0, 1, 0,
            PKT_CONSISTENT, SLEEP_FUNC, NULL);
```

**EXAMPLE 2** Allocate an inquiry packet with consistent DMA resources attached.

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
       SUN_INQSIZE, B_READ, canwait, NULL);
if (bp) {
    pkt = scsi_init_pkt(&devp->sd_address, NULL, bp,
             CDB_GROUP0, 1, PP_LEN, PKT_CONSISTENT,
             canwait, NULL);
}
```

SEE ALSO  
ddi_dma_mem_alloc(9F), ddi_dma_sync(9F), getrbuf(9F),
scsi_destroy_pkt(9F), scsi_init_pkt(9F), scsi_free_consistent_buf(9F),
buf(9S), scsi_address(9S)

Writing Device Drivers
### NAME
scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);
```

### INTERFACE
Solaris DDI specific (Solaris DDI).

### LEVEL

### PARAMETERS
- **cmd**
  A SCSI command value.
- **cmdvec**
  Pointer to an array of command strings.
- **dtype**
  Device type.
- **msg**
  A message value.
- **reason**
  A packet reason value.
- **sense_key**
  A SCSI sense key value.

### DESCRIPTION
`scsi_cname()` decodes SCSI commands. `cmdvec` is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

`scsi_dname()` decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

`scsi_mname()` decodes SCSI messages.

`scsi_rname()` decodes packet completion reasons.

`scsi_sname()` decodes SCSI sense keys.

### RETURN VALUES
These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

### CONTEXT
These functions can be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1** Decoding SCSI tape commands.

`scsi_cname()` decodes SCSI tape commands as follows:

```c
static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
    "\003request sense",
    "\010read",
    "\012write",
```
EXAMPLE 1 Decoding SCSI tape commands.  (Continued)

    "\020write file mark",
    "\021space",
    "\022inquiry",
    "\025mode select",
    "\031erase tape",
    "\032mode sense",
    "\033load tape",
    NULL
};  

SEE ALSO  Writing Device Drivers
### NAME
scsi_destroy_pkt – free an allocated SCSI packet and its DMA resource

### SYNOPSIS
```c
#include <sys/scsi/scsi.h>

void scsi_destroy_pkt(struct scsi_pkt *pkt);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **pkt**: Pointer to a scsi_pkt(9S) structure.

### DESCRIPTION
scsi_destroy_pkt() releases all necessary resources, typically at the end of an I/O transfer. The data is synchronized to memory, then the DMA resources are deallocated and `pkt` is freed.

### CONTEXT
scsi_destroy_pkt() may be called from user or interrupt context.

### EXAMPLES
**EXAMPLE 1** Releasing resources.
```
scsi_destroy_pkt(un->un_rqs);
```

### SEE ALSO
- tran_destroy_pkt(9E), scsi_init_pkt(9F), scsi_pkt(9S)
- *Writing Device Drivers*
NAME
scsi_dmgaget, scsi_dmafree – SCSI dma utility routines

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *
scsi_dmgaget(struct scsi_pkt *pkt, opaque_t
dmatoken, int (*callback)(void));

void scsi_dmafree(struct scsi_pkt *pkt);

INTERFACE LEVEL
These interfaces are obsolete. Use scsi_init_pkt(9F) instead of scsi_dmgaget().
Use scsi_destroy_pkt(9F) instead of scsi_dmafree().

PARAMETERS
pkt A pointer to a scsi_pkt(9S) structure.
dmatoken Pointer to an implementation dependent object
callback Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.

DESCRIPTION
scsi_dmgaget() allocates DMA resources for an already allocated SCSI packet. pkt is
a pointer to the previously allocated SCSI packet (see scsi_pktalloc(9F)).

dmatoken is a pointer to an implementation dependent object which defines the length,
direction, and address of the data transfer associated with this SCSI packet
(command). The dmatoken must be a pointer to a buf(9S) structure. If dmatoken is
NULL, no resources are allocated.

callback indicates what scsi_dmgaget() should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC Wait indefinitely for resources.

Other Values callback points to a function which is called when resources may
have become available. callback must return either 0 (indicating
that it attempted to allocate resources but failed to do so again), in
which case it is put back on a list to be called again later, or 1
indicating either success in allocating resources or indicating that it
no longer cares for a retry.

scsi_dmafree() frees the DMA resources associated with the SCSI packet. The
packet itself remains allocated.

RETURN VALUES
scsi_dmgaget() returns a pointer to a scsi_pkt on success. It returns NULL if
resources are not available.

CONTEXT
If callback is SLEEP_FUNC, then this routine may only be called from user-level code.
Otherwise, it may be called from either user or interrupt level. The callback function
may not block or call routines that block.

scsi_dmafree() can be called from user or interrupt context.
ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), scsi_pktalloc(9F), scsi_pktfree(9F), scsi_resalloc(9F), scsi_resfree(9F), buf(9S), scsi_pkt(9S)

Writing Device Drivers

NOTES The scsi_dmaget() and scsi_dmafree() functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, scsi_init_pkt(9F) and scsi_destroy_pkt(9F).
NAME | scsi_dmaget, scsi_dmafree – SCSI dma utility routines
SYNOPSIS | #include <sys/scsi/scsi.h>

 struct scsi_pkt *scsi_dmaget(struct scsi_pkt *pkt, opaque_t
dmatoken, int (*callback)(void));

 void scsi_dmafree(struct scsi_pkt *pkt);

INTERFACE LEVEL | These interfaces are obsolete. Use scsi_init_pkt(9F) instead of scsi_dmaget(). Use scsi_destroy_pkt(9F) instead of scsi_dmafree().
PARAMETERS | pkt | A pointer to a scsi_pkt(9S) structure.
dmatoken | Pointer to an implementation dependent object
callback | Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.

DESCRIPTION | scsi_dmaget() allocates DMA resources for an already allocated SCSI packet. pkt is a pointer to the previously allocated SCSI packet (see scsi_pktalloc(9F)).

dmatoken is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The dmatoken must be a pointer to a buf(9S) structure. If dmatoken is NULL, no resources are allocated.

callback indicates what scsi_dmaget() should do when resources are not available:

 NULL_FUNC | Do not wait for resources. Return a NULL pointer.
 SLEEP_FUNC | Wait indefinitely for resources.

 Other Values | callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

RETURN VALUES | scsi_dmaget() returns a pointer to a scsi_pkt on success. It returns NULL if resources are not available.

CONTEXT | If callback is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block.

 scsi_dmafree() can be called from user or interrupt context.
ATTRIBUTES

See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), scsi_pktalloc(9F), scsi_pktfree(9F), scsi_resalloc(9F), scsi_resfree(9F), buf(9S), scsi_pkt(9S)

Writing Device Drivers

NOTES

The scsi_dmaget() and scsi_dmafree() functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, scsi_init_pkt(9F) and scsi_destroy_pkt(9F).
NAME

scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

SYNOPSIS

#include <sys/scsi/scsi.h>

char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

cmd          A SCSI command value.
cmdvec       Pointer to an array of command strings.
dtype        Device type.
msg          A message value.
reason       A packet reason value.
sense_key    A SCSI sense key value.

DESCRIPTION

scsi_cname() decodes SCSI commands. cmdvec is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

csci_dname() decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

scsi_mname() decodes SCSI messages.

scsi_rname() decodes packet completion reasons.

scsi_sname() decodes SCSI sense keys.

RETURN VALUES

These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLES

EXAMPLE 1 Decoding SCSI tape commands.

scsi_cname() decodes SCSI tape commands as follows:

static char *st_cmds[] = {
  "\000test unit ready",
  "\001rewind",
  "\003request sense",
  "\010read",
  "\012write",
  "\020write file mark",
};

1392  man pages section 9: DDI and DKI Kernel Functions • Last Revised 21 Dec 1992
EXAMPLE 1 Decoding SCSI tape commands.  (Continued)

    "\021space",
    "\022inquiry",
    "\025mode select",
    "\031erase tape",
    "\032mode sense",
    "\033load tape",
    NULL
};

    cmn_err(CR_CONT, "st: cmd=%s", scsi_cname(cmd, st_cmds));
    ...

SEE ALSO  Writing Device Drivers
### NAME
scsi_errmsg – display a SCSI request sense message

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

void scsi_errmsg(struct scsi_device *devp, struct scsi_pkt *pktp,
                  char *drv_name, int severity, daddr_t blkno, daddr_t err_blkno,
                  struct scsi_key_strings *cmdlist, struct scsi_extended_sense
                  *sensep);
```

### INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI).

- **devp**
  Pointer to the `scsi_device(9S)` structure.

- **pktp**
  Pointer to a `scsi_pkt(9S)` structure.

- **drv_name**
  String used by `scsi_log(9F)`.

- **severity**
  Error severity level, maps to severity strings below.

- **blkno**
  Requested block number.

- **err_blkno**
  Error block number.

- **cmdlist**
  An array of SCSI command description strings.

- **sensep**
  A pointer to a `scsi_extended_sense(9S)` structure.

### DESCRIPTION

`scsi_errmsg()` interprets the request sense information in the `sensep` pointer and generates a standard message that is displayed using `scsi_log(9F)`. The first line of the message is always a `CE_WARN`, with the continuation lines being `CE_CONT`. `sensep` may be `NULL`, in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The `scsi_device(9S)` structure denoted by `devp` supplies the identification of the device that requested the display. `severity` selects which string is used in the "Error Level:" reporting, according to the following table:

<table>
<thead>
<tr>
<th>Severity Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_ERR_ALL</td>
<td>All</td>
</tr>
<tr>
<td>SCSI_ERR_UNKNOWN</td>
<td>Unknown</td>
</tr>
<tr>
<td>SCSI_ERR_INFO</td>
<td>Informational</td>
</tr>
<tr>
<td>SCSI_ERR_RECOVER</td>
<td>Recovered</td>
</tr>
<tr>
<td>SCSI_ERR_RETRYABLE</td>
<td>Retryable</td>
</tr>
<tr>
<td>SCSI_ERR_FATAL</td>
<td>Fatal</td>
</tr>
</tbody>
</table>
blkno is the block number of the original request that generated the error. err_blkno is the block number where the error occurred. cmdlist is a mapping table for translating the SCSI command code in pkt to the actual command string.

The cmdlist is described in the structure below:

```
struct scsi_key_strings {
    int key;
    char *message;
};
```

For a basic SCSI disk, the following list is appropriate:

```
static struct scsi_key_strings scsi_cmds[] = {
    0x00, "test unit ready",
    0x01, "rezero/rewind",
    0x03, "request sense",
    0x04, "format",
    0x07, "reassign",
    0x08, "read",
    0x0a, "write",
    0x0b, "seek",
    0x12, "inquiry",
    0x15, "mode select",
    0x16, "reserve",
    0x17, "release",
    0x18, "copy",
    0x1a, "mode sense",
    0x1b, "start/stop",
    0x1e, "door lock",
    0x28, "read(10)",
    0x2a, "write(10)",
    0x2f, "verify",
    0x37, "read defect data",
    0x3b, "write buffer",
    -1, NULL
};
```

Kernel Functions for Drivers 1395
scsi_errmsg(9F)

SEE ALSO cmn_err(9F), scsi_log(9F), scsi_device(9S), scsi_extended_sense(9S), scsi_pkt(9S)

Writing Device Drivers
scsi_free_consistent_buf(9F)

NAME
scsi_free_consistent_buf – free a previously allocated SCSI DMA I/O buffer

SYNOPSIS
#include <sys/scsi/scsi.h>

void scsi_free_consistent_buf(struct buf *bp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
bp Pointer to the buf(9S) structure.

DESCRIPTION
scsi_free_consistent_buf() frees a buffer header and consistent data buffer that was previously allocated using scsi_alloc_consistent_buf(9F).

CONTEXT
scsi_free_consistent_buf() may be called from either the user or the interrupt levels.

SEE ALSO
freerbuf(9F), scsi_alloc_consistent_buf(9F), buf(9S)

Writing Device Drivers

WARNING
scsi_free_consistent_buf() will call freerbuf(9F) to free the buf(9S) that was allocated before or during the call to scsi_alloc_consistent_buf(9F).

If consistent memory is bound to a scsi_pkt(9S), the pkt should be destroyed before freeing the consistent memory.
The `scsi_get_device_type_scsi_options()` function looks up the property `device-type-scsi-options-list`, which can be specified in the HBA's `driver.conf(4)` file. This property allows specification of `scsi-options` on a per-device-type basis.

The formal syntax is:

```
device-type-scsi-options-list = <duplet> [, <duplet> *];
```

where:

```
<duplet> := <vid+pid>, <scsi-options-property-name>
```

and:

```
<scsi-options-property-name> = <value>;
```

The string `<vid+pid>` is returned by the device on a SCSI inquiry command. This string can contain any character in the range 0x20-0x7e. Characters such as double quote ("), or single quote (’), which are not permitted in property value strings, are represented by their octal equivalent (for example, \042 and \047). Trailing spaces can be truncated.

For example:

```
device-type-scsi-options-list=
  "SEAGATE ST32550W", "seagate-options",
  "EXABYTE EXB-2501", "exabyte-options",
  "IBM OEM DPHSS4S", "ibm-options";
```

```
seagate-options = 0x78;
exabyte-options = 0x58;
ibm-options = 0x378;
```

The `scsi_get_device_type_scsi_options()` function searches the list of duplets for a matching INQUIRY string. If a match is found, `scsi_get_device_type_scsi_options()` returns the corresponding value.
| **RETURN VALUES**  | `scsi_get_device_type_scsi_options()` returns the `scsi-options` value found, or if no match is found the `default_scsi_options` value passed in. |
| **CONTEXT**       | This function can be called from kernel or interrupt context. |
| **SEE ALSO**      | *Writing Device Drivers* |
#include <sys/scsi/scsi.h>

int scsi_hba_attach_setup(dev_info_t *dip, ddi_dma_attr_t *hba_dma_attr, scsi_hba_tran_t *hba_tran, int hba_flags);

int scsi_hba_attach(dev_info_t *dip, ddi_dma_lim_t *hba_lim, scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);

int scsi_hba_detach(dev_info_t *dip);

Solaris architecture specific (Solaris DDI).

dip
A pointer to the dev_info_t structure, referring to the instance of the HBA device.

hba_lim
A pointer to a ddi_dma_lim(9S) structure.

hba_tran
A pointer to a scsi_hba_tran(9S) structure.

hba_flags
Flag modifiers. The only defined flag value is SCSI_HBA_TRAN_CLONE.

hba_options
Optional features provided by the HBA driver for future extensions; must be NULL.

hba_dma_attr
A pointer to a ddi_dma_attr(9S) structure.

scsi_hba_attach_setup() is the recommended interface over scsi_hba_attach().

For scsi_hba_attach_setup() and scsi_hba_attach():

scsi_hba_attach() registers the DMA limits hba_lim and the transport vectors hba_tran of each instance of the HBA device defined by dip.

scsi_hba_attach_setup() registers the DMA attributes hba_dma_attr and the transport vectors hba_tran of each instance of the HBA device defined by dip. The HBA driver can pass different DMA limits or DMA attributes, and transport vectors for each instance of the device, as necessary, to support any constraints imposed by the HBA itself.

scsi_hba_attach() and scsi_hba_attach_setup() use the dev_bus_ops field in the dev_ops(9S) structure. The HBA driver should initialize this field to NULL before calling scsi_hba_attach() or scsi_hba_attach_setup().

If SCSI_HBA_TRAN_CLONE is requested in hba_flags, the hba_tran structure will be cloned once for each target attached to the HBA. The cloning of the structure will occur before the tran_tgt_init(9E) entry point is called to initialize a target. At all subsequent HBA entry points, including tran_tgt_init(9E), the scsi_hba_tran_t structure passed as an argument or found in a scsi_address structure will be the ‘cloned’ scsi_hba_tran_t structure, thus allowing the HBA to
use the tran_tgt_private field in the scsi_hba_tran_t structure to point to per-target data. The HBA must take care to free only the same scsi_hba_tran_t structure it allocated when detaching; all ‘cloned’ scsi_hba_tran_t structures allocated by the system will be freed by the system.

scsi_hba_attach() and scsi_hba_attach_setup() attach a number of integer-valued properties to dip, unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via ddi_prop_get_int(9F), and respect any settings for features provided the HBA.

**scsi-options**

Optional SCSI configuration bits

- **SCSI_OPTIONS_DR**: If not set, the HBA should not grant Disconnect privileges to target devices.
- **SCSI_OPTIONS_LINK**: If not set, the HBA should not enable Linked Commands.
- **SCSI_OPTIONS_TAG**: If not set, the HBA should not operate in Command Tagged Queueing mode.
- **SCSI_OPTIONS_FAST**: If not set, the HBA should not operate the bus in FAST SCSI mode.
- **SCSI_OPTIONS_FAST20**: If not set, the HBA should not operate the bus in FAST20 SCSI mode.
- **SCSI_OPTIONS_WIDE**: If not set, the HBA should not operate the bus in WIDE SCSI mode.
- **SCSI_OPTIONS_SYNC**: If not set, the HBA should not operate the bus in synchronous transfer mode.

**scsi-reset-delay**

SCSI bus or device reset recovery time, in milliseconds.

**scsi-selection-timeout**

Default SCSI selection phase timeout value, in milliseconds. Please refer to individual HBA man pages for any HBA-specific information

For scsi_hba_detach():

scsi_hba_detach() removes the reference to the DMA limits or attributes structure and the transport vector for the given instance of an HBA driver.
scsi_hba_attach() \hline
\text{name} \hspace{1cm} \text{value} \\
\hline
\text{scsi_hba_attach()}, \text{scsi_hba_attach_setup()}, \text{and scsi_hba_detach()} return DDI_SUCCESS if the function call succeeds, and return DDI_FAILURE on failure.

\text{CONTEXT} \hspace{1cm} \text{scsi_hba_attach()} and \text{scsi_hba_attach_setup()} should be called from attach(9E). \text{scsi_hba_detach()} should be called from detach(9E).

\text{SEE ALSO} \hspace{1cm} attach(9E), detach(9E), tran_tgt_init(9E), ddi_prop_get_int(9F), ddi_dma_attr(9S), ddi_dma_lim(9S), dev_ops(9S), scsi_address(9S), scsi_hba_tran(9S)

\text{Writing Device Drivers}

\text{NOTES} \hspace{1cm} It is the HBA driver’s responsibility to ensure that no more transport requests will be taken on behalf of any SCSI target device driver after \text{scsi_hba_detach()} is called.

The \text{scsi_hba_attach()} function is obsolete and will be discontinued in a future release. This function is replaced by \text{scsi_hba_attach_setup()}.\hline
NAME | scsi_hba_attach_setup, scsi_hba_attach, scsi_hba_detach – SCSI HBA attach and detach routines

SYNOPSIS | #include <sys/scsi/scsi.h>

int scsi_hba_attach_setup (dev_info_t *dip, ddi_dma_attr_t *hba_dma_attr, scsi_hba_tran_t *hba_tran, int hba_flags);

int scsi_hba_attach (dev_info_t *dip, ddi_dma_lim_t *hba_lim, scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);

int scsi_hba_detach (dev_info_t *dip);

INTERFACE LEVEL | Solaris architecture specific (Solaris DDI).

PARAMETERS |

dip | A pointer to the dev_info_t structure, referring to the instance of the HBA device.

hba_lim | A pointer to a ddi_dma_lim(9S) structure.

hba_tran | A pointer to a scsi_hba_tran(9S) structure.

hba_flags | Flag modifiers. The only defined flag value is SCSI_HBA_TRAN_CLONE.

hba_options | Optional features provided by the HBA driver for future extensions; must be NULL.

hba_dma_attr | A pointer to a ddi_dma_attr(9S) structure.

DESCRIPTION | scsi_hba_attach_setup() is the recommended interface over scsi_hba_attach().

For scsi_hba_attach_setup() and scsi_hba_attach():

scsi_hba_attach() registers the DMA limits hba_lim and the transport vectors hba_tran of each instance of the HBA device defined by dip.

scsi_hba_attach_setup() registers the DMA attributes hba_dma_attr and the transport vectors hba_tran of each instance of the HBA device defined by dip. The HBA driver can pass different DMA limits or DMA attributes, and transport vectors for each instance of the device, as necessary, to support any constraints imposed by the HBA itself.

scsi_hba_attach() and scsi_hba_attach_setup() use the dev_bus_ops field in the dev_ops(9S) structure. The HBA driver should initialize this field to NULL before calling scsi_hba_attach() or scsi_hba_attach_setup().

If SCSI_HBA_TRAN_CLONE is requested in hba_flags, the hba_tran structure will be cloned once for each target attached to the HBA. The cloning of the structure will occur before the tran_tgt_init(9E) entry point is called to initialize a target. At all subsequent HBA entry points, including tran_tgt_init(9E), the scsi_hba_tran_t structure passed as an argument or found in a scsi_address structure will be the ‘cloned’ scsi_hba_tran_t structure, thus allowing the HBA to
use the `tran_tgt_private` field in the `scsi_hba_tran_t` structure to point to per-target data. The HBA must take care to free only the same `scsi_hba_tran_t` structure it allocated when detaching; all ‘cloned’ `scsi_hba_tran_t` structures allocated by the system will be freed by the system.

`scsi_hba_attach()` and `scsi_hba_attach_setup()` attach a number of integer-valued properties to `dip`, unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via `ddi_prop_get_int(9F)`, and respect any settings for features provided the HBA.

**scsi-options**

Optional SCSI configuration bits

- **SCSI_OPTIONS_DR**
  If not set, the HBA should not grant Disconnect privileges to target devices.

- **SCSI_OPTIONS_LINK**
  If not set, the HBA should not enable Linked Commands.

- **SCSI_OPTIONS_TAG**
  If not set, the HBA should not operate in Command Tagged Queueing mode.

- **SCSI_OPTIONS_FAST**
  If not set, the HBA should not operate the bus in FAST SCSI mode.

- **SCSI_OPTIONS_FAST20**
  If not set, the HBA should not operate the bus in FAST20 SCSI mode.

- **SCSI_OPTIONS_WIDE**
  If not set, the HBA should not operate the bus in WIDE SCSI mode.

- **SCSI_OPTIONS_SYNC**
  If not set, the HBA should not operate the bus in synchronous transfer mode.

**scsi-reset-delay**

SCSI bus or device reset recovery time, in milliseconds.

**scsi-selection-timeout**

Default SCSI selection phase timeout value, in milliseconds. Please refer to individual HBA man pages for any HBA-specific information.

For `scsi_hba_detach()`:

`scsi_hba_detach()` removes the reference to the DMA limits or attributes structure and the transport vector for the given instance of an HBA driver.
scsi_hba_attach(), scsi_hba_attach_setup(), and scsi_hba_detach() return DDI_SUCCESS if the function call succeeds, and return DDI_FAILURE on failure.

scsi_hba_attach() and scsi_hba_attach_setup() should be called from attach(9E). scsi_hba_detach() should be called from detach(9E).

attach(9E), detach(9E), tran_tgt_init(9E), ddi_prop_get_int(9F), ddi_dma_attr(9S), ddi_dma_lim(9S), dev_ops(9S), scsi_address(9S), scsi_hba_tran(9S)

Writing Device Drivers

It is the HBA driver’s responsibility to ensure that no more transport requests will be taken on behalf of any SCSI target device driver after scsi_hba_detach() is called.

The scsi_hba_attach() function is obsolete and will be discontinued in a future release. This function is replaced by scsi_hba_attach_setup().
NAME

scsi_hba_attach_setup, scsi_hba_attach, scsi_hba_detach – SCSI HBA attach and detach routines

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_attach_setup(dev_info_t *dip, ddi_dma_attr_t *hba_dma_attr, scsi_hba_tran_t *hba_tran, int hba_flags);

int scsi_hba_attach(dev_info_t *dip, ddi_dma_lim_t *hba_lim, scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);

int scsi_hba_detach(dev_info_t *dip);

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

PARAMETERS

dip A pointer to the dev_info_t structure, referring to the instance of the HBA device.

hba_lim A pointer to a ddi_dma_lim(9S) structure.

hba_tran A pointer to a scsi_hba_tran(9S) structure.

hba_flags Flag modifiers. The only defined flag value is SCSI_HBA_TRAN_CLONE.

hba_options Optional features provided by the HBA driver for future extensions; must be NULL.

hba_dma_attr A pointer to a ddi_dma_attr(9S) structure.

DESCRIPTION

scsi_hba_attach_setup() is the recommended interface over scsi_hba_attach().

For scsi_hba_attach_setup() and scsi_hba_attach():

scsi_hba_attach() registers the DMA limits hba_lim and the transport vectors hba_tran of each instance of the HBA device defined by dip.

scsi_hba_attach_setup() registers the DMA attributes hba_dma_attr and the transport vectors hba_tran of each instance of the HBA device defined by dip. The HBA driver can pass different DMA limits or DMA attributes, and transport vectors for each instance of the device, as necessary, to support any constraints imposed by the HBA itself.

scsi_hba_attach() and scsi_hba_attach_setup() use the dev_bus_ops field in the dev_ops(9S) structure. The HBA driver should initialize this field to NULL before calling scsi_hba_attach() or scsi_hba_attach_setup().

If SCSI_HBA_TRAN_CLONE is requested in hba_flags, the hba_tran structure will be cloned once for each target attached to the HBA. The cloning of the structure will occur before the tran_tgt_init(9E) entry point is called to initialize a target. At all subsequent HBA entry points, including tran_tgt_init(9E), the scsi_hba_tran_t structure passed as an argument or found in a scsi_address structure will be the ‘cloned’ scsi_hba_tran_t structure, thus allowing the HBA to
use the tran_tgt_private field in the scsi_hba_tran_t structure to point to per-target data. The HBA must take care to free only the same scsi_hba_tran_t structure it allocated when detaching; all ‘cloned’ scsi_hba_tran_t structures allocated by the system will be freed by the system.

scsi_hba_attach() and scsi_hba_attach_setup() attach a number of integer-valued properties to dip, unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via ddi_prop_get_int(9F), and respect any settings for features provided the HBA.

scsi-options

Optional SCSI configuration bits

SCSI_OPTIONS_DR
  If not set, the HBA should not grant Disconnect privileges to target devices.

SCSI_OPTIONS_LINK
  If not set, the HBA should not enable Linked Commands.

SCSI_OPTIONS_TAG
  If not set, the HBA should not operate in Command Tagged Queueing mode.

SCSI_OPTIONS_FAST
  If not set, the HBA should not operate the bus in FAST SCSI mode.

SCSI_OPTIONS_FAST20
  If not set, the HBA should not operate the bus in FAST20 SCSI mode.

SCSI_OPTIONS_WIDE
  If not set, the HBA should not operate the bus in WIDE SCSI mode.

SCSI_OPTIONS_SYNC
  If not set, the HBA should not operate the bus in synchronous transfer mode.

scsi-reset-delay

SCSI bus or device reset recovery time, in milliseconds.

scsi-selection-timeout

Default SCSI selection phase timeout value, in milliseconds. Please refer to individual HBA man pages for any HBA-specific information

For scsi_hba_detach():

scsi_hba_detach() removes the reference to the DMA limits or attributes structure and the transport vector for the given instance of an HBA driver.
scsi_hba_detach(9F)

RETURN VALUES  

scsi_hba_attach(), scsi_hba_attach_setup(), and scsi_hba_detach() return DDI_SUCCESS if the function call succeeds, and return DDI_FAILURE on failure.

CONTEXT  

scsi_hba_attach() and scsi_hba_attach_setup() should be called from attach(9E). scsi_hba_detach() should be called from detach(9E).

SEE ALSO  

attach(9E), detach(9E), tran_tgt_init(9E), ddi_prop_get_int(9F), ddi_dma_attr(9S), ddi_dma_lim(9S), dev_ops(9S), scsi_address(9S), scsi_hba_tran(9S)

Writing Device Drivers

NOTES  

It is the HBA driver’s responsibility to ensure that no more transport requests will be taken on behalf of any SCSI target device driver after scsi_hba_detach() is called.

The scsi_hba_attach() function is obsolete and will be discontinued in a future release. This function is replaced by scsi_hba_attach_setup().
### NAME
scsi_hba_init, scsi_hba_fini – SCSI Host Bus Adapter system initialization and completion routines

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

int scsi_hba_init(struct modlinkage *modlp);
void scsi_hba_fini(struct modlinkage *modlp);
```

### INTERFACE LEVEL
Solaris architecture specific (Solaris DDI).

### PARAMETERS
- **modlp**
  Pointer to the Host Bus Adapters module linkage structure.

**scsi_hba_init()**
- `scsi_hba_init()` is the system-provided initialization routine for SCSI HBA drivers. The `scsi_hba_init()` function registers the HBA in the system and allows the driver to accept configuration requests on behalf of SCSI target drivers. The `scsi_hba_init()` routine must be called in the HBA’s `_init(9E)` routine before `mod_install(9F)` is called. If `mod_install(9F)` fails, the HBA’s `_init(9E)` should call `scsi_hba_fini()` before returning failure.

**scsi_hba_fini()**
- `scsi_hba_fini()` is the system provided completion routine for SCSI HBA drivers. `scsi_hba_fini()` removes all of the system references for the HBA that were created in `scsi_hba_init()`. The `scsi_hba_fini()` routine should be called in the HBA’s `_fini(9E)` routine if `mod_remove(9F)` is successful.

### RETURN VALUES
- `scsi_hba_init()` returns 0 if successful, and a non-zero value otherwise. If `scsi_hba_init()` fails, the HBA’s `_init()` entry point should return the value returned by `scsi_hba_init()`.

### CONTEXT
- `scsi_hba_init()` and `scsi_hba_fini()` should be called from `_init(9E)` or `_fini(9E)`, respectively.

### SEE ALSO
- `_fini(9E), _init(9E), mod_install(9F), mod_remove(9F), scsi_pktalloc(9F), scsi_pktfree(9F), scsi_hba_tran(9S)

### Writing Device Drivers

### NOTES
- The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once `scsi_hba_fini()` is called.
s HDMI интерфейса

NAME

s HDMI инициализации и процесса завершения.

SYNOPSIS

#include <sys/scsi/scsi.h>

int s HDMI_init(struct modlinkage *modlp);

void s HDMI_fini(struct modlinkage *modlp);

INTERFACE LEVEL

PARAMETERS

modlp

Pointer to the Host Bus Adapters module linkage structure.

scsi_hba_init() — s HDMI инициализации — это системное предоставление инициализации для SCSI HBA драйверов.

scsi_hba_init() функция регистрирует HBA в системе и позволяет драйверу принять запросы конфигурации от имени SCSI target drivers. The s HDMI_init() функция должна вызываться в HBA ‘s _init(9E) routine перед mod_install(9F) вызывается. Если mod_install(9F) возвращает ошибку, HBA’s _init(9E) должна вызывать s HDMI_fini() до возврата ошибки.

scsi_hba_fini() — s HDMI завершения — это системное предоставление завершения для SCSI HBA драйверов.

scsi_hba_fini() удаляет все системные ссылки на HBA, которые были созданы в s HDMI_init(). The s HDMI_fini() функция должна быть вызвана в HBA’s _fini(9E) routine if mod_remove(9F) успешный.

RETURN VALUES

scsi_hba_init() возвращает 0 если успешный, и ненулевой значение в противном случае. Если s HDMI_init() неудачный, HBA’s _init() entry point should return the value returned by scsi_hba_init().

CONTEXT

s HDMI_init() и s HDMI_fini() должны быть вызваны из _init(9E) или _fini(9E), соответственно.

SEE ALSO

_mod_install(9F), mod_remove(9F), scsi_pktalloc(9F), scsi_pktfree(9F), scsi_hba_tran(9S)

Writing Device Drivers

NOTES

The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once s HDMI_fini() is called.
NAME
scsi_hba_lookup_capstr – return index matching capability string

SYNOPSIS
#include <sys/scsi/scsi.h>

text

INTERFACE

LEVEL

PARAMETERS

capstr Pointer to a string.

DESCRIPTION
scsi_hba_lookup_capstr() attempts to match capstr against a known set of
capability strings, and returns the defined index for the matched capability, if found.

The set of indices and capability strings is:

SCSI_CAP_DMA_MAX
    "dma-max" or "dma_max"

SCSI_CAP_MSG_OUT
    "msg-out" or "msg_out"

SCSI_CAP_DISCONNECT
    "disconnect"

SCSI_CAP_SYNCHRONOUS
    "synchronous"

SCSI_CAP_WIDE_XFER
    "wide-xfer" or "wide_xfer"

SCSI_CAP_PARITY
    "parity"

SCSI_CAP_INITIATOR_ID
    "initiator-id"

SCSI_CAP_UNTAGGED_QING
    "untagged-qing"

SCSI_CAP_TAGGED_QING
    "tagged-qing"

SCSI_CAP_ARQ
    "auto-rqsense"

SCSI_CAP_LINKED_CMDS
    "linked-cmds"

SCSI_CAP_SECTOR_SIZE
    "sector-size"

SCSI_CAP_TOTAL_SECTORS
    "total-sectors"

SCSI_CAP_GEOMETRY
    "geometry"
scci_hba_lookup_capstr(9F)

<table>
<thead>
<tr>
<th>Capability String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_CAP_RESET_NOTIFICATION</td>
<td>&quot;reset-notification&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_QFULL_RETRIES</td>
<td>&quot;qfull-retries&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_QFULL_RETRY_INTERVAL</td>
<td>&quot;qfull-retry-interval&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_LUN_RESET</td>
<td>&quot;lun-reset&quot;</td>
</tr>
</tbody>
</table>

RETURN VALUES
scci_hba_lookup_capstr() returns a non-negative index value corresponding to the capability string, or -1 if the string does not match any known capability.

CONTEXT
scci_hba_lookup_capstr() can be called from user or interrupt context.

SEE ALSO
tran_getcap(9E), tran_setcap(9E), scsi_ifgetcap(9F), scci_ifsetcap(9F), scci_reset_notify(9F)

Writing Device Drivers
NAME

scsi_hba_pkt_alloc, scsi_hba_pkt_free – allocate and free a scsi_pkt structure

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_hba_pkt_alloc(dev_info_t *dip, struct scsi_address *ap, int cmdlen, int statuslen, int tgtlen, int hbalen, int (*callback, caddr_t arg, caddr_t arg);

void scsi_hba_pkt_free(struct scsi_address *ap, struct scsi_pkt *pkt);

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

PARAMETERS

dip
Pointer to a dev_info_t structure, defining the HBA driver instance.

ap
Pointer to a scsi_address(9S) structure, defining the target instance.

cmdlen
Length in bytes to be allocated for the SCSI command descriptor block (CDB).

statuslen
Length in bytes to be allocated for the SCSI status completion block (SCB).

tgtlen
Length in bytes to be allocated for a private data area for the target driver’s exclusive use.

hbalen
Length in bytes to be allocated for a private data area for the HBA driver’s exclusive use.

callback
Indicates what scsi_hba_pkt_alloc() should do when resources are not available:

NULL_FUNC
Do not wait for resources. Return a NULL pointer.

SLEEP_FUNC
Wait indefinitely for resources.

arg
Must be NULL.

pkt
A pointer to a scsi_pkt(9S) structure.

DESCRIPTION

For scsi_hba_pkt_alloc():

scsi_hba_pkt_alloc() allocates space for a scsi_pkt structure. HBA drivers should use this interface when allocating a scsi_pkt from their tran_init_pkt(9E) entry point.

If callback is NULL_FUNC, scsi_hba_pkt_alloc() may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.

scsi_hba_pkt_alloc() copies the scsi_address(9S) structure pointed to by ap to the pkt_address field in the scsi_pkt(9S).
`scsi_hba_pkt_alloc()` also allocates memory for these `scsi_pkt(9S)` data areas, and sets these fields to point to the allocated memory:

- `pkt_ha_private`          HBA private data area.
- `pkt_private`             Target driver private data area.
- `pkt_scbp`                SCSI status completion block.
- `pkt_cdbp`                SCSI command descriptor block.

For `scsi_hba_pkt_free()`:

`scsi_hba_pkt_free()` frees the space allocated for the `scsi_pkt(9S)` structure.

**RETURN VALUES**

`scsi_hba_pkt_alloc()` returns a pointer to the `scsi_pkt` structure, or `NULL` if no space is available.

**CONTEXT**

`scsi_hba_pkt_alloc()` can be called from user or interrupt context. Drivers must not allow `scsi_hba_pkt_alloc()` to sleep if called from an interrupt routine.

`scsi_hba_pkt_alloc()` can be called from user or interrupt context.

**SEE ALSO**

`tran_init_pkt(9E), scsi_address(9S), scsi_pkt(9S)`

*Writing Device Drivers*
**NAME**
scci_hba_pkt_alloc, scci_hba_pkt_free – allocate and free a scsi_pkt structure

**SYNOPSIS**
#include <sys/scsi/scsi.h>

struct scsi_pkt *scci_hba_pkt_alloc(dev_info_t *dip, struct
  scsi_address *ap, int cmdlen, int statuslen, int tgtlen, int hbalen,
  int (*callback, caddr_t arg, caddr_t arg);

void scci_hba_pkt_free(struct scsi_address *ap, struct scsi_pkt *
pkt);

**INTERFACE LEVEL**
Solaris architecture specific (Solaris DDI).

**PARAMETERS**
dip Pointer to a dev_info_t structure, defining the HBA driver instance.
ap Pointer to a scsi_address(9S) structure, defining the target instance.
cmdlen Length in bytes to be allocated for the SCSI command descriptor block (CDB).
statuslen Length in bytes to be allocated for the SCSI status completion block (SCB).
tgtlen Length in bytes to be allocated for a private data area for the target driver’s exclusive use.
hbalen Length in bytes to be allocated for a private data area for the HBA driver’s exclusive use.
callback Indicates what scci_hba_pkt_alloc() should do when resources are not available:
  NULL_FUNC
    Do not wait for resources. Return a NULL pointer.
  SLEEP_FUNC
    Wait indefinitely for resources.
arg Must be NULL.
pkt A pointer to a scsi_pkt(9S) structure.

**DESCRIPTION**
For scci_hba_pkt_alloc():

scci_hba_pkt_alloc() allocates space for a scsi_pkt structure. HBA drivers should use this interface when allocating a scsi_pkt from their tran_init_pkt(9E) entry point.

If callback is NULL_FUNC, scci_hba_pkt_alloc() may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.

scci_hba_pkt_alloc() copies the scsi_address(9S) structure pointed to by ap to the pkt_address field in the scsi_pkt(9S).
scsi_hba_pkt_alloc() also allocates memory for these scsi_pkt(9S) data areas, and sets these fields to point to the allocated memory:

- **pkt_ha_private**: HBA private data area.
- **pkt_private**: Target driver private data area.
- **pkt_scbp**: SCSI status completion block.
- **pkt_cdbp**: SCSI command descriptor block.

For scsi_hba_pkt_free():

- **scsi_hba_pkt_free()** frees the space allocated for the scsi_pkt(9S) structure.

**RETURN VALUES**

- **scsi_hba_pkt_alloc()** returns a pointer to the scsi_pkt structure, or NULL if no space is available.

**CONTEXT**

- **scsi_hba_pkt_alloc()** can be called from user or interrupt context. Drivers must not allow scsi_hba_pkt_alloc() to sleep if called from an interrupt routine.

- **scsi_hba_pkt_free()** can be called from user or interrupt context.

**SEE ALSO**

- tran_init_pkt(9E), scsi_address(9S), scsi_pkt(9S)

*Writing Device Drivers*
# Name
scci_hba_probe – default SCSI HBA probe function

## Synopsis
```c
#include <sys/scsi/scsi.h>

int scci_hba_probe(struct scsi_device *sd, int (*waitfunc)(void));
```

## Interface Level
Solaris architecture specific (Solaris DDI).

### Parameters
- `sd` Pointer to a `scsi_device(9S)` structure describing the target.
- `waitfunc` NULL_FUNC or SLEEP_FUNC.

## Description
`scci_hba_probe()` is a function providing the semantics of `scci_probe(9F)`. An HBA driver may call `scci_hba_probe()` from its `tran_tgt_probe(9E)` entry point, to probe for the existence of a target on the SCSI bus, or the HBA may set `tran_tgt_probe(9E)` to point to `scci_hba_probe` directly.

## Return Values
See `scci_probe(9F)` for the return values from `scci_hba_probe()`.

## Context
`scci_hba_probe()` should only be called from the HBA’s `tran_tgt_probe(9E)` entry point.

## See Also
- `tran_tgt_probe(9E), scci_probe(9F), scci_device(9S)`
- *Writing Device Drivers*
NAME
csci_hba_tran_alloc, csci_hba_tran_free – allocate and free transport structures

SYNOPSIS
#include <sys/scsi/scsi.h>
scsi_hba_tran_t *csci_hba_tran_alloc(dev_info_t *dip, int flags);
void csci_hba_tran_free(scsi_hba_tran_t *hba_tran);

INTERFACE LEVEL
PARAMETERS
Solaris architecture specific (Solaris DDI).
dip Pointer to a dev_info structure, defining the HBA driver instance.
flag Flag modifiers. The only possible flag value is
SCSI_HBA_CANSLEEP (memory allocation may sleep).
hba_tran Pointer to a scsi_hba_tran(9S) structure.

DESCRIPTION
For csci_hba_tran_alloc():

csci_hba_tran_alloc() allocates a scsi_hba_tran(9S) structure for a HBA
driver. The HBA must use this structure to register its transport vectors with the
system by using csci_hba_attach_setup(9F).

If the flag SCSI_HBA_CANSLEEP is set in flags, csci_hba_tran_alloc() may sleep
when allocating resources; otherwise it may not sleep, and callers should be prepared
to deal with allocation failures.

For csci_hba_tran_free():

csci_hba_tran_free() is used to free the scsi_hba_tran(9S) structure allocated
by csci_hba_tran_alloc().

RETURN VALUES
For csci_hba_tran_alloc():

Returns a pointer to the allocated transport structure, or
NULL if no space is available.

For csci_hba_tran_free():

Can be called from user or interrupt context. Drivers must
not allow csci_hba_tran_alloc() to sleep if called from an interrupt routine.

SEE ALSO
For csci_hba_tran_alloc():

csci_hba_attach_setup(9F), csci_hba_tran(9S)

Writing Device Drivers
scsi_hba_tran_alloc, scsi_hba_tran_free – allocate and free transport structures

#include <sys/scsi/scsi.h>

scsi_hba_tran_t *scsi_hba_tran_alloc(dev_info_t *dip, int flags);
void scsi_hba_tran_free(scsi_hba_tran_t *hba_tran);

Solaris architecture specific (Solaris DDI).

dip Pointer to a dev_info structure, defining the HBA driver instance.

flag Flag modifiers. The only possible flag value is SCSI_HBA_CANSLEEP (memory allocation may sleep).

hba_tran Pointer to a scsi_hba_tran(9S) structure.

For scsi_hba_tran_alloc():

scsi_hba_tran_alloc() allocates a scsi_hba_tran(9S) structure for a HBA driver. The HBA must use this structure to register its transport vectors with the system by using scsi_hba_attach_setup(9F).

If the flag SCSI_HBA_CANSLEEP is set in flags, scsi_hba_tran_alloc() may sleep when allocating resources; otherwise it may not sleep, and callers should be prepared to deal with allocation failures.

For scsi_hba_tran_free():

scsi_hba_tran_free() is used to free the scsi_hba_tran(9S) structure allocated by scsi_hba_tran_alloc().

scsi_hba_tran_alloc() returns a pointer to the allocated transport structure, or NULL if no space is available.

scsi_hba_tran_alloc() can be called from user or interrupt context. Drivers must not allow scsi_hba_tran_alloc() to sleep if called from an interrupt routine.

scsi_hba_tran_free() can be called from user or interrupt context.

scsi_hba_attach_setup(9F), scsi_hba_tran(9S)

Writing Device Drivers
NAME
scsi_ifgetcap, scsi_ifsetcap – get/set SCSI transport capability

SYNOPSIS
#include <sys/scsi/scsi.h>

int scsi_ifgetcap(struct scsi_address *ap, char *cap, int whom);
int scsi_ifsetcap(struct scsi_address *ap, char *cap, int value, int whom);

INTERFACE LEVEL PARAMETERS
Solaris DDI specific (Solaris DDI).
ap
Pointer to the scsi_address structure.
cap
Pointer to the string capability identifier.
value
Defines the new state of the capability.
whom
Determines if all targets or only the specified target is affected.

DESCRIPTION
The target drivers use scsi_ifsetcap() to set the capabilities of the host adapter driver. A cap is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using scsi_ifgetcap(). If whom is 0 all targets are affected, else the target specified by the scsi_address structure pointed to by ap is affected.

A device may support only a subset of the capabilities listed below. It is the responsibility of the driver to make sure that these functions are called with a cap supported by the device.

The following capabilities have been defined:
dma-max
Maximum dma transfer size supported by host adapter.
msg-out
Message out capability supported by host adapter: 0 disables, 1 enables.
disconnect
Disconnect capability supported by host adapter: 0 disables, 1 enables.
synchronous
Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.
wide-xfer
Wide transfer capability supported by host adapter: 0 disables, 1 enables.
parity
Parity checking by host adapter: 0 disables, 1 enables.
initiator-id
The host’s bus address is returned.
untagged-qing
The host adapter’s capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.
tagged-qing
The host adapter’s capability to support tagged queueing: 0 disables, 1 enables.
auto-rqsense | The host adapter's capability to support auto request sense on check conditions: 0 disables, 1 enables.

sector-size | The target driver sets this capability to inform the HBA of the granularity, in bytes, of DMA breakup; the HBA's DMA limit structure will be set to reflect this limit (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). It should be set to the physical disk sector size. This capability defaults to 512.

total-sectors | The target driver sets this capability to inform the HBA of the total number of sectors on the device, as returned from the SCSI get capacity command. This capability must be set before the target driver "gets" the geometry capability.

gridometry | This capability returns the HBA geometry of a target disk. The target driver must set the total-sectors capability before "getting" the geometry capability. The geometry is returned as a 32-bit value: the upper 16 bits represent the number of heads per cylinder; the lower 16 bits represent the number of sectors per track. The geometry capability cannot be "set."

If geometry is not relevant or appropriate for this target disk, because (for example) the HBA BIOS supports Logical Block Addressing for this drive, it is acceptable for scsi_ifgetcap() to return -1, indicating that the geometry is not defined. This will cause failure of attempts to retrieve the "virtual geometry" from the target driver (the DKIOC_G_VIRTGEOM ioctl will fail). See dkio(7I) for more information about DKIOC_G_VIRTGEOM.

reset-notification | The host adapter's capability to support bus reset notification: 0 disables, 1 enables. Refer to scsi_reset_notify(9F).

linked-cmds | The host adapter's capability to support linked commands: 0 disables, 1 enables.

qfull-retries | This capability enables/disables QUEUE FULL handling. If 0, the HBA will not retry a command when a QUEUE FULL status is returned. If greater than 0, then the HBA driver will retry the command at specified number of times at an interval determined by the "qfull-retry-interval". The range for qfull-retries is 0-255.
This capability sets the retry interval (in ms) for commands that were completed with a QUEUEFULL status. The range for qfull-retry-intervals is 0-1000 ms.

This capability is created with a value of zero by HBA drivers that support the RESET_LUN flag in their tran_reset(9E) routine. If it exists, its value can be set to 1 by target drivers that want to allow use of LOGICAL UNIT RESET on a specific target instance. If lun-reset does not exist or has a value of zero, scsi_reset(9F) is prevented from passing the RESET_LUN flag to the HBA driver’s tran_reset() routine. If lun-reset exists and has a value of 1, then the HBA driver’s tran_reset() routine can be called with the RESET_LUN flag.

scsi_ifsetcap() returns:

1 If the capability was successfully set to the new value.
0 If the capability is not variable.
-1 If the capability was not defined, or setting the capability to a new value failed.

scsi_ifgetcap() returns the current value of a capability, or:

-1 If the capability was not defined.

These functions can be called from user or interrupt context.

EXAMPLE 1 Using scsi_ifgetcap()

```c
if (scsi_ifgetcap(&sd->sd_address, "auto-rqsense", 1) == 1) {
    un->un_arq_enabled = 1;
} else {
    un->un_arq_enabled =
        ((scsi_ifsetcap(&sd->sd_address, "auto-rqsense", 1, 1) == 1) ?
         1 : 0);
}

if (scsi_ifsetcap(&devp->sd_address, "tagged-qing", 1, 1) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = MAX_THROTTLE;
} else if (scsi_ifsetcap(&devp->sd_address, "untagged-qing", 0) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = 3;
} else {
    un->un_dp->options &= ~SD_QUEUEING;
    un->un_throttle = 1;
}
```
EXAMPLE 1 Using scsi_ifgetcap() (Continued)

SEE ALSO

tran_reset(9E), scsi_hba_lookup_capstr(9F), scsi_reset(9F),
scsi_reset_notify(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S),
scsi_address(9S), scsi_arq_status(9S)

Writing Device Drivers
**NAME**
scsi_ifgetcap, scsi_ifsetcap – get/set SCSI transport capability

**SYNOPSIS**
#include <sys/scsi/scsi.h>

int scsi_ifgetcap(struct scsi_address *ap, char *cap, int whom);
int scsi_ifsetcap(struct scsi_address *ap, char *cap, int value, int whom);

**INTERFACE LEVEL PARAMETERS**
Solaris DDI specific (Solaris DDI).

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>Pointer to the scsi_address structure.</td>
</tr>
<tr>
<td>cap</td>
<td>Pointer to the string capability identifier.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the new state of the capability.</td>
</tr>
<tr>
<td>whom</td>
<td>Determines if all targets or only the specified target is affected.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**
The target drivers use `scsi_ifsetcap()` to set the capabilities of the host adapter driver. A *cap* is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using `scsi_ifgetcap()`. If *whom* is 0 all targets are affected, else the target specified by the scsi_address structure pointed to by *ap* is affected.

A device may support only a subset of the capabilities listed below. It is the responsibility of the driver to make sure that these functions are called with a *cap* supported by the device.

The following capabilities have been defined:

<table>
<thead>
<tr>
<th>Cap</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dma-max</td>
<td>Maximum dma transfer size supported by host adapter.</td>
</tr>
<tr>
<td>msg-out</td>
<td>Message out capability supported by host adapter: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>disconnect</td>
<td>Disconnect capability supported by host adapter: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>synchronous</td>
<td>Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>wide-xfer</td>
<td>Wide transfer capability supported by host adapter: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>parity</td>
<td>Parity checking by host adapter: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>initiator-id</td>
<td>The host’s bus address is returned.</td>
</tr>
<tr>
<td>untagged-qing</td>
<td>The host adapter’s capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.</td>
</tr>
<tr>
<td>tagged-qing</td>
<td>The host adapter’s capability to support tagged queueing: 0 disables, 1 enables.</td>
</tr>
</tbody>
</table>
The host adapter’s capability to support auto request sense on check conditions: 0 disables, 1 enables.

The target driver sets this capability to inform the HBA of the granularity, in bytes, of DMA breakup; the HBA’s DMA limit structure will be set to reflect this limit (see `ddi_dma_lim_sparc(9S)` or `ddi_dma_lim_x86(9S)`). It should be set to the physical disk sector size. This capability defaults to 512.

The target driver sets this capability to inform the HBA of the total number of sectors on the device, as returned from the SCSI get capacity command. This capability must be set before the target driver “gets” the geometry capability.

This capability returns the HBA geometry of a target disk. The target driver must set the `total-sectors` capability before “getting” the geometry capability. The geometry is returned as a 32-bit value: the upper 16 bits represent the number of heads per cylinder; the lower 16 bits represent the number of sectors per track. The geometry capability cannot be “set.”

If geometry is not relevant or appropriate for this target disk, because (for example) the HBA BIOS supports Logical Block Addressing for this drive, it is acceptable for `scsi_ifgetcap()` to return -1, indicating that the geometry is not defined. This will cause failure of attempts to retrieve the “virtual geometry” from the target driver (the `DKIOCG_VIRTGEOM` ioctl will fail). See `dkio(7I)` for more information about `DKIOCG_VIRTGEOM`.

The host adapter’s capability to support bus reset notification: 0 disables, 1 enables. Refer to `scsi_reset_notify(9F)`.

The host adapter’s capability to support linked commands: 0 disables, 1 enables.

This capability enables/disables QUEUE FULL handling. If 0, the HBA will not retry a command when a QUEUE FULL status is returned. If greater than 0, then the HBA driver will retry the command at specified number of times at an interval determined by the "qfull-retry-interval". The range for `qfull-retries` is 0-255.
qfull-retry-interval  This capability sets the retry interval (in ms) for commands that were completed with a QUEUE FULL status. The range for qfull-retry-intervals is 0-1000 ms.

lun-reset  This capability is created with a value of zero by HBA drivers that support the RESET_LUN flag in their tran_reset(9E) routine. If it exists, its value can be set to 1 by target drivers that want to allow use of LOGICAL UNIT RESET on a specific target instance. If lun-reset does not exist or has a value of zero, scsi_reset(9F) is prevented from passing the RESET_LUN flag to the HBA driver’s tran_reset() routine. If lun-reset exists and has a value of 1, then the HBA driver’s tran_reset() routine can be called with the RESET_LUN flag.

RETURN VALUES  
scsi_ifsetcap() returns:

1  If the capability was successfully set to the new value.
0  If the capability is not variable.
-1  If the capability was not defined, or setting the capability to a new value failed.

scsi_ifgetcap() returns the current value of a capability, or:

-1  If the capability was not defined.

CONTEXT  
These functions can be called from user or interrupt context.

EXAMPLES  
EXAMPLE 1 Using scsi_ifgetcap()

if (scsi_ifgetcap(sd->sd_address, "auto-rqsense", 1) == 1) {
    un->un_arq_enabled = 1;
} else {
    un->un_arq_enabled =
        ((scsi_ifgetcap(sd->sd_address, "auto-rqsense", 1, 1) == 1) ?
        1 : 0);
}

if (scsi_ifsetcap(&dev->sd_address, "tagged-qing", 1, 1) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = MAX_THROTTLE;
} else if (scsi_ifgetcap(&dev->sd_address, "untagged-qing", 0) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = 3;
} else {
    un->un_dp->options &= ~SD_QUEUEING;
    un->un_throttle = 1;
}
EXAMPLE 1 Using `scsi_ifgetcap()` (Continued)

SEE ALSO

- `tran_reset(9E)`, `scsi_hba_lookup_capstr(9F)`, `scsi_reset(9F)`, `scsi_reset_notify(9F)`, `ddi_dma_lim_sparc(9S)`, `ddi_dma_lim_x86(9S)`, `scsi_address(9S)`, `scsi_arq_status(9S)

*Writing Device Drivers*
NAME
scsi_init_pkt – prepare a complete SCSI packet

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap, struct
  scsi_pkt *pktp, struct buf *bp, int cmdlen, int statuslen, int
  privatelen, int flags, int (*callback)(caddr_t), caddr_t arg);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS

DESCRIPTION
Target drivers use scsi_init_pkt() to request the transport layer to allocate and
initialize a packet for a SCSI command which possibly includes a data transfer. If pktp
is NULL, a new scsi_pkt(9S) is allocated using the HBA driver’s packet allocator.
The bp is a pointer to a buf(9S) structure. If bp is non-NULL and contains a valid byte
count, the buf(9S) structure is also set up for DMA transfer using the HBA driver
DMA resources allocator. When bp is allocated by
scsi_alloc_consistent_buf(9F), the PKT_CONSISTENT bit must be set in the
flags argument to ensure proper operation. If privatelen is non-zero then additional
space is allocated for the `pkt_private` area of the `scsi_pkt(9S)`. On return `pkt_private` points to this additional space. Otherwise `pkt_private` is a pointer that is typically used to store the `bp` during execution of the command. In this case `pkt_private` is `NULL` on return.

The `flags` argument is a set of bit flags. Possible bits include:

PKT_CONSISTENT
This must be set if the DMA buffer was allocated using `scsi_alloc_consistent_buf(9F)`. In this case, the HBA driver will guarantee that the data transfer is properly synchronized before performing the target driver’s command completion callback.

PKT_DMA_PARTIAL
This may be set if the driver can accept a partial DMA mapping. If set, `scsi_init_pkt()` will allocate DMA resources with the DDI_DMA_PARTIAL bit set in the `dmar_flag` element of the `ddi_dma_req(9S)` structure. The `pkt_resid` field of the `scsi_pkt(9S)` structure may be returned with a non-zero value, which indicates the number of bytes for which `scsi_init_pkt()` was unable to allocate DMA resources. In this case, a subsequent call to `scsi_init_pkt()` may be made for the same `pktp` and `bp` to adjust the DMA resources to the next portion of the transfer. This sequence should be repeated until the `pkt_resid` field is returned with a zero value, which indicates that with transport of this final portion the entire original request will have been satisfied.

When calling `scsi_init_pkt()` to move already-allocated DMA resources, the `cmdlen`, `statuslen`, and `privatelen` fields are ignored.

The last argument `arg` is supplied to the `callback` function when it is invoked.

`callback` indicates what the allocator routines should do when resources are not available:

- **NULL_FUNC**: Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC**: Wait indefinitely for resources.
- **Other Values**: `callback` points to a function which is called when resources may have become available. `callback` must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

When allocating DMA resources, `scsi_init_pkt()` returns the `scsi_pkt` field `pkt_resid` as the number of residual bytes for which the system was unable to allocate DMA resources. A `pkt_resid` of 0 means that all necessary DMA resources were allocated.
scsi_init_pkt() returns NULL if the packet or DMA resources could not be allocated. Otherwise, it returns a pointer to an initialized scsi_pkt(9S). If pktp was not NULL the return value will be pktp on successful initialization of the packet.

If callback is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block.

**EXAMPLE 1** Allocating a Packet Without DMA Resources Attached

To allocate a packet without DMA resources attached, use:

```c
pkt = scsi_init_pkt(&devp->sd_address, NULL, NULL, CDB_GROUP1,
    1, sizeof (struct my_pkt_private *), 0,
    sd_runout, sd_unit);
```

**EXAMPLE 2** Allocating a Packet With DMA Resources Attached

To allocate a packet with DMA resources attached use:

```c
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP1,
    sizeof(struct scsi_arq_status), 0, 0, NULL_FUNC, NULL);
```

**EXAMPLE 3** Attaching DMA Resources to a Preallocated Packet

To attach DMA resources to a preallocated packet, use:

```c
pkt = scsi_init_pkt(&devp->sd_address, old_pkt, bp, 0,
    0, 0, 0, sd_runout, (caddr_t) sd_unit);
```

**EXAMPLE 4** Allocating a Packet with Consistent DMA Resources Attached

Since the packet is already allocated, the cmdlen, statuslen and privatelen are 0. To allocate a packet with consistent DMA resources attached, use:

```c
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
    SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
    sizeof(struct scsi_arq_status), sizeof (struct my_pkt_private *),
    PKT_CONSISTENT, SLEEP_FUNC, NULL);
```

**EXAMPLE 5** Allocating a Packet with Partial DMA Resources Attached

To allocate a packet with partial DMA resources attached, use:

```c
my_pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
    1, sizeof (struct buf *), PKT_DMA_PARTIAL,
    SLEEP_FUNC, NULL);
```
EXAMPLE 5 Allocating a Packet with Partial DMA Resources Attached (Continued)

scsi_alloc_consistent_buf(9F),
scsi_destroy_pkt(9F),
scsi_dmaget(9F),
scsi_pktalloc(9F),
buf(9S),
ddi_dma_req(9S),
scsi_address(9S),
scsi_pkt(9S)

Writing Device Drivers

SEE ALSO

NOTES

If a DMA allocation request fails with DDI_DMA_NOMAPPING, the B_ERROR flag will be set in bp, and the b_error field will be set toEFAULT.

If a DMA allocation request fails with DDI_DMA_TOOBIG, the B_ERROR flag will be set in bp, and the b_error field will be set toEINVAL.
scsi_log(9F)

NAME
scsi_log – display a SCSI-device-related message

SYNOPSIS
#include <sys/scsi/scsi.h>
#include <sys/cmn_err.h>

void scsi_log(dev_info_t *dip, char *drv_name, uint_t level, const char *fmt, ...);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
dip Pointer to the dev_info structure.
drv_name String naming the device.
level Error level.
fmt Display format.

DESCRIPTION
scsi_log() is a utility function that displays a message via the cmn_err(9F) routine. The error levels that can be passed in to this function are CE_PANIC, CE_WARN, CE_NOTE, CE_CONT, and SCSI_DEBUG. The last level is used to assist in displaying debug messages to the console only. drv_name is the short name by which this device is known; example disk driver names are sd and cmdk. If the dev_info_t pointer is NULL, then the drv_name will be used with no unit or long name.

If the first character in fmt is:

■ An exclamation mark (!), the message goes only to the system buffer.
■ A caret (^), the message goes only to the console.
■ A question mark (?) and level is CE_CONT, the message is always sent to the system buffer, but is written to the console only when the system has been booted in verbose mode. See kernel(1M). If neither condition is met, the ? character has no effect and is simply ignored.

All formatting conversions in use by cmn_err() also work with scsi_log().

CONTEXT
scsi_log() may be called from user or interrupt context.

EXAMPLES
EXAMPLE 1
scsi_log(dev, "Disk Unit ", CE_PANIC, "Bad Value %d
", foo);
generates:
PANIC: /eisa/aha@330,0/cmdk@0,0 (Disk Unit 0): Bad Value 5

This is followed by a PANIC.

EXAMPLE 2
scsi_log(dev, "sd", CE_WARN, "Label Bad\n");
generates:
WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1): Label Bad
EXAMPLE 3
```
scsi_log((dev_info_t *) NULL, "Disk Unit ", CE_NOTE, "Disk Ejected\n");
generates:
Disk Unit: Disk Ejected
```

EXAMPLE 4
```
scsi_log(cmdk_unit, "Disk Unit ", CE_CONT, "Disk Inserted\n");
generates:
Disk Inserted
```

EXAMPLE 5
```
scsi_log(sd_unit, "sd", SCSI_DEBUG, "We really got here\n");
generates (only to the console):
DEBUG: sd1: We really got here
```

SEE ALSO
```
kernel(1M), sd(7D), cmn_err(9F), scsi_errmsg(9F)
```

Writing Device Drivers
scsi_mname(9F)

NAME
scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

SYNOPSIS
#include <sys/scsi/scsi.h>

char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
cmd A SCSI command value.
cmdvec Pointer to an array of command strings.
dtype Device type.
msg A message value.
reason A packet reason value.
sense_key A SCSI sense key value.

DESCRIPTION
scsi_cname() decodes SCSI commands. cmdvec is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

csci_dname() decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

scsi_mname() decodes SCSI messages.

scsi_rname() decodes packet completion reasons.

scsi_sname() decodes SCSI sense keys.

RETURN VALUES
These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Decoding SCSI tape commands.

scsi_cname() decodes SCSI tape commands as follows:

static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
    "\003request sense",
    "\010read",
    "\012write",

    };
EXAMPLE 1 Decoding SCSI tape commands.  (Continued)

    "\020write file mark",
    "\021space",
    "\022inquiry",
    "\025mode select",
    "\031erase tape",
    "\032mode sense",
    "\033load tape",
    NULL
};
...

SEE ALSO  Writing Device Drivers
scsi_pktalloc(9F)

NAME

scsi_pktalloc, scsi_resalloc, scsi_pktfree, scsi_resfree - SCSI packet utility routines

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *
scsi_pktalloc(struct scsi_address* ap, int cmdlen, int statuslen, int (*callback)(void));

struct scsi_pkt *
scsi_resalloc(struct scsi_address* ap, int cmdlen, int statuslen, opaque_t dmatoken, int (*callback)(void));

void
scsi_pktfree(struct scsi_pkt* pkt);

void
scsi_resfree(struct scsi_pkt* pkt);

INTERFACE LEVEL

The scsi_pktalloc(), scsi_pktfree(), scsi_resalloc(), and scsi_resfree() functions are obsolete. The scsi_pktalloc() and scsi_resalloc() functions have been replaced by scsi_init_pkt(9F). The scsi_pktfree() and scsi_resfree() functions have been replaced by scsi_destroy_pkt(9F).

PARAMETERS

ap Pointer to a scsi_address structure.

cmdlen The required length for the SCSI command descriptor block (CDB) in bytes.

statuslen The required length for the SCSI status completion block (SCB) in bytes.

dmatoken Pointer to an implementation-dependent object.

callback A pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.

pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION

scsi_pktalloc() requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, scsi_resalloc() should be used.

ap is a pointer to a scsi_address structure. Allocator routines use it to determine the associated host adapter.

cmdlen is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the pkt_cdbp field of the allocated scsi_pkt structure.

statuslen is the required length for the SCSI status completion block. The address of the allocated block is placed into the pkt_scbp field of the scsi_pkt structure.

dmatoken is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The dmatoken must be a pointer to a buf(9S) structure. If dmatoken is NULL, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data
transfer phase (either no data transfer phase expected, or the wrong direction encountered), the command is terminated with the pkt_reason set to CMD_DMA_DERR. dmatoken provides the information to determine if the transfer count is correct.

callback indicates what the allocator routines should do when resources are not available:

- **NULL_FUNC**  Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC**  Wait indefinitely for resources.
- **Other Values**  callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

`scsi_pktfree()` frees the packet.

`scsi_resfree()` free all resources held by the packet and the packet itself.

**RETURN VALUES** Both allocation routines return a pointer to a scsi_pkt structure on success, or NULL on failure.

**CONTEXT** If callback is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

**ATTRIBUTES** See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO** attributes(5), scsi_dmafree(9F), scsi_dmaget(9F), buf(9S), scsi_pkt(9S)

Writing Device Drivers

**NOTES** The `scsi_pktalloc()`, `scsi_pktfree()`, `scsi_resalloc()`, and `scsi_resfree()` functions are obsolete and will be discontinued in a future release. The `scsi_pktalloc()` and `scsi_resalloc()` functions have been replaced by `scsi_init_pkt(9F)`. The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by `scsi_destroy_pkt(9F)`.

`scsi_pktalloc(9F)`
SCSI packet utility routines

```
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_pktalloc(struct scsi_address* ap, int cmdlen, int statuslen, int (*callback)(void));

struct scsi_pkt *scsi_resalloc(struct scsi_address* ap, int cmdlen, int statuslen, opaque_t dmatoken, int (*callback)(void));

void scsi_pktfree(struct scsi_pkt* pkt);

void scsi_resfree(struct scsi_pkt* pkt);
```

The `scsi_pktalloc()`, `scsi_pktfree()`, `scsi_resalloc()`, and `scsi_resfree()` functions are obsolete. The `scsi_pktalloc()` and `scsi_resalloc()` functions have been replaced by `scsi_init_pkt(9F)`. The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by `scsi_destroy_pkt(9F).

- **ap** Pointer to a `scsi_address` structure.
- **cmdlen** The required length for the SCSI command descriptor block (CDB) in bytes.
- **statuslen** The required length for the SCSI status completion block (SCB) in bytes.
- **dmatoken** Pointer to an implementation-dependent object.
- **callback** A pointer to a callback function, or `NULL_FUNC` or `SLEEP_FUNC`.
- **pkt** Pointer to a `scsi_pkt(9S)` structure.

`salli_pktalloc()` requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, `scsi_resalloc()` should be used.

`ap` is a pointer to a `scsi_address` structure. Allocator routines use it to determine the associated host adapter.

`cmdlen` is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the `pkt_cdbp` field of the allocated `scsi_pkt` structure.

`statuslen` is the required length for the SCSI status completion block. The address of the allocated block is placed into the `pkt_scbp` field of the `scsi_pkt` structure.

`dmatoken` is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The `dmatoken` must be a pointer to a `buf(9S)` structure. If `dmatoken` is `NULL`, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data transfer
transfer phase (either no data transfer phase expected, or the wrong direction
encountered), the command is terminated with the pkt_reason set to
CMD_DMA_DERR. dmatoken provides the information to determine if the transfer count
is correct.

callback indicates what the allocator routines should do when resources are not
available:

NULL_FUNC    Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC   Wait indefinitely for resources.

Other Values   callback points to a function which is called when resources may
have become available. callback must return either 0 (indicating
that it attempted to allocate resources but again failed to do so), in
which case it is put back on a list to be called again later, or 1
indicating either success in allocating resources or indicating that it
no longer cares for a retry.

scsi_pktfree() frees the packet.

scsi_resfree() free all resources held by the packet and the packet itself.

RETURN VALUES
Both allocation routines return a pointer to a scsi_pkt structure on success, or NULL
on failure.

CONTEXT
If callback is SLEEP_FUNC, then this routine may only be called from user-level code.
Otherwise, it may be called from either user or interrupt level. The callback function
may not block or call routines that block. Both deallocation routines can be called from
user or interrupt context.

ATTRIBUTES
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), scsi_dmfree(9F), scsi_dmaget(9F), buf(9S), scsi_pkt(9S)

Writing Device Drivers

NOTES
The scsi_pktalloc(), scsi_pktfree(), scsi_resalloc(), and
scsi_resfree() functions are obsolete and will be discontinued in a future release.
The scsi_pktalloc() and scsi_resalloc() functions have been replaced by
scsi_init_pkt(9F). The scsi_pktfree() and scsi_resfree() functions have
been replaced by scsi_destroy_pkt(9F).
### NAME
scsi_poll - run a polled SCSI command on behalf of a target driver

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

int scsi_poll(struct scsi_pkt *pkt);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### PARAMETERS
- **pkt**
  Pointer to the `scsi_pkt(9S)` structure.

### DESCRIPTION
`scsi_poll()` requests the host adapter driver to run a polled command. Unlike `scsi_transport(9F)` which runs commands asynchronously, `scsi_poll()` runs commands to completion before returning. If the `pkt_time` member of `pkt` is 0, the value of `pkt_time` is defaulted to `SCSI_POLL_TIMEOUT` to prevent an indefinite hang of the system.

### RETURN VALUES
`scsi_poll()` returns:
- **0** command completed successfully.
- **-1** command failed.

### CONTEXT
`scsi_poll()` can be called from user or interrupt level. This function should not be called when the caller is executing `timeout(9F)` in the context of a thread.

### SEE ALSO
- `makecom(9F), scsi_transport(9F), scsi_pkt(9S)`
- *Writing Device Drivers*

### WARNINGS
Since `scsi_poll()` runs commands to completion before returning, it may require more time than is desirable when called from interrupt context. Therefore, calling `scsi_poll` from interrupt context is not recommended.
NAME | scsi_probe – utility for probing a scsi device

SYNOPSIS | `#include <sys/scsi/scsi.h>
int scsi_probe(struct scsi_device *devp, int (*waitfunc);`

INTERFACE | Solaris DDI specific (Solaris DDI).
LEVEL |
PARAMETERS | `devp` Pointer to a scsi_device(9S) structure
`waitfunc` NULL_FUNC or SLEEP_FUNC

DESCRIPTION | `scsi_probe()` determines whether a target/lun is present and sets up the scsi_device structure with inquiry data.

`scsi_probe()` uses the SCSI Inquiry command to test if the device exists. It can retry the Inquiry command as appropriate. If `scsi_probe()` is successful, it will allocate space for the scsi_inquiry structure and assign the address to the sd_inq member of the scsi_device(9S) structure. `scsi_probe()` will then fill in this scsi_inquiry(9S) structure and return SCSIPROBE_EXISTS. If `scsi_probe()` is unsuccessful, it returns SCSIPROBE_NOMEM in spite of callback set to SLEEP_FUNC.

`scsi_unprobe(9F)` is used to undo the effect of `scsi_probe()`.

If the target is a non-CCS device, SCSIPROBE_NONCCS will be returned.

`waitfunc` indicates what the allocator routines should do when resources are not available; the valid values are:

- NULL_FUNC Do not wait for resources. Return SCSIPROBE_NOMEM or SCSIPROBE_FAILURE
- SLEEP_FUNC Wait indefinitely for resources.

RETURN VALUES | `scsi_probe()` returns:

- SCSIPROBE_BUSY Device exists but is currently busy.
- SCSIPROBE_EXISTS Device exists and inquiry data is valid.
- SCSIPROBE_FAILURE Polled command failure.
- SCSIPROBE_NOMEM No space available for structures.
- SCSIPROBE_NOMEM_CB No space available for structures but callback request has been queued.
- SCSIPROBE_NONCCS Device exists but inquiry data is not valid.
- SCSIPROBE_NORESP Device does not respond to an INQUIRY.

CONTEXT | `scsi_probe()` is normally called from the target driver’s probe(9E) or attach(9E) routine. In any case, this routine should not be called from interrupt context, because it can sleep waiting for memory to be allocated.
EXPLANATION

EXAMPLE 1 Using scsi_probe()

```c
switch (scsi_probe(devp, NULL_FUNC)) {
    default:
        case SCSIPROBE_NORESP:
        case SCSIPROBE_NONCCS:
        case SCSIPROBE_NOMEM:
        case SCSIPROBE_FAILURE:
        case SCSIPROBE_BUSY:
            break;
        case SCSIPROBE_EXISTS:
            switch (devp->sd_inq->inq_dtype) {
                case DTYPE_DIRECT:
                    rval = DDI_PROBE_SUCCESS;
                    break;
                case DTYPE_RODIRECT:
                    rval = DDI_PROBE_SUCCESS;
                    break;
                case DTYPE_NOTPRESENT:
                    default:
                        break;
            }
    }
    scsi_unprobe(devp);
}
```

SEE ALSO

attach(9E), probe(9E), scsi_slave(9F), scsi_unprobe(9F), scsi_unslave(9F), scsi_device(9S), scsi_inquiry(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES

A `waitfunc` function other than `NULL_FUNC` or `SLEEP_FUNC` is not supported and may have unexpected results.
scsi_pktalloc(), scsi_resalloc(), scsi_pktfree(), scsi_resfree() - SCSI packet utility routines

#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_pktalloc(struct scsi_address *ap, int cmdlen, int statuslen, int (*callback)(void));

struct scsi_pkt *scsi_resalloc(struct scsi_address *ap, int cmdlen, int statuslen, opaque_t dmatoken, int (*callback)(void));

void scsi_pktfree(struct scsi_pkt *pkt);

void scsi_resfree(struct scsi_pkt *pkt);

The scsi_pktalloc(), scsi_pktfree(), scsi_resalloc(), and scsi_resfree() functions are obsolete. The scsi_pktalloc() and scsi_resalloc() functions have been replaced by scsi_init_pkt(9F). The scsi_pktfree() and scsi_resfree() functions have been replaced by scsi_destroy_pkt(9F).

PARAMETERS

- **ap**: Pointer to a scsi_address structure.
- **cmdlen**: The required length for the SCSI command descriptor block (CDB) in bytes.
- **statuslen**: The required length for the SCSI status completion block (SCB) in bytes.
- **dmatoken**: Pointer to an implementation-dependent object.
- **callback**: A pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.
- **pkt**: Pointer to a scsi_pkt(9S) structure.

DESCRIPTION

scsi_pktalloc() requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, scsi_resalloc() should be used.

*ap* is a pointer to a scsi_address structure. Allocator routines use it to determine the associated host adapter.

*cmdlen* is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the pkt_cdbp field of the allocated scsi_pkt structure.

*statuslen* is the required length for the SCSI status completion block. The address of the allocated block is placed into the pkt_scbp field of the scsi_pkt structure.

*dmatoken* is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The *dmatoken* must be a pointer to a buf(9S) structure. If *dmatoken* is NULL, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data
The transfer phase (either no data transfer phase expected, or the wrong direction encountered), the command is terminated with the pkt_reason set to CMD_DMA_DERR. dmatoken provides the information to determine if the transfer count is correct.

callback indicates what the allocator routines should do when resources are not available:

- **NULL_FUNC**: Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC**: Wait indefinitely for resources.
- **Other Values**: callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

scsi_pktfree() frees the packet.

scsi_resfree() free all resources held by the packet and the packet itself.

**RETURN VALUES**: Both allocation routines return a pointer to a scsi_pkt structure on success, or NULL on failure.

**CONTEXT**: If callback is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

**ATTRIBUTES**: See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**: attributes(5), scsi_dmafree(9F), scsi_dmaget(9F), buf(9S), scsi_pkt(9S)

**Writing Device Drivers**

The scsi_pktalloc(), scsi_pktfree(), scsi_resalloc(), and scsi_resfree() functions are obsolete and will be discontinued in a future release. The scsi_pktalloc() and scsi_resalloc() functions have been replaced by scsi_init_pkt(9F). The scsi_pktfree() and scsi_resfree() functions have been replaced by scsi_destroy_pkt(9F).
scsi_reset() — reset a SCSI bus or target

#include <sys/scsi/scsi.h>

int scsi_reset(struct scsi_address *ap, int level);

Solaris DDI specific (Solaris DDI).

**PARAMETERS**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>Pointer to the scsi_address structure.</td>
</tr>
<tr>
<td>level</td>
<td>The level of reset required.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

scsi_reset() asks the host adapter driver to reset the SCSI bus or a SCSI target as specified by level. If level equals RESET_ALL, the SCSI bus is reset. If it equals RESET_TARGET, ap is used to determine the target to be reset. If it equals RESET_LUN, ap is used to determine the logical unit to be reset.

When given the RESET_LUN level, scsi_reset() can return failure if the LOGICAL UNIT RESET message is not supported by the target device, or if the underlying HBA driver does not implement the ability to issue a LOGICAL UNIT RESET message.

Note that, at the point when scsi_reset() resets the logical unit (case RESET_LUN), or the target (case RESET_TARGET), or the bus (case RESET_ALL), there might be one or more command packets outstanding. That is, packets have been passed to scsi_transport(), and queued or possibly transported, but the commands have not been completed and the target completion routine has not been called for those packets.

The successful call to scsi_reset() has the side effect that any such commands currently outstanding are aborted, at which point the packets are marked with pkt_reason set to CMD_RESET, and the appropriate bit -- either STAT_BUS_RESET or STAT_DEV_RESET -- is set in pkt_statistics. Once thus appropriately marked, the aborted command packets are passed to the target driver command completion routine.

Also note that, at the moment that a thread executing scsi_reset() actually resets the target or the bus, it is possible that a second thread may have already called scsi_transport(), but not yet queued or transported its command. In this case the HBA will not yet have received the second thread’s packet and this packet will not be aborted.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_reset()</td>
<td>returns:</td>
</tr>
<tr>
<td>1</td>
<td>Upon success.</td>
</tr>
<tr>
<td>0</td>
<td>Upon failure.</td>
</tr>
</tbody>
</table>

**CONTEXT**

scsi_reset() can be called from user or interrupt context.

**SEE ALSO**

tran_reset(9E), tran_reset_notify(9E), scsi_abort(9F)

Writing Device Drivers
NAME | scsi_reset_notify – notify target driver of bus resets

SYNOPSIS | #include <sys/scsi/scsi.h>

void scsi_reset_notify(struct scsi_address *ap, int flag, void (*callback)(caddr_t), caddr_t arg);

INTERFACE LEVEL PARAMETERS | Solaris DDI specific (Solaris DDI).

ap | Pointer to the scsi_address structure.

flag | A flag indicating registration or cancellation of the notification request.

callback | A pointer to the target driver's reset notification function.

arg | The callback function argument.

DESCRIPTION | scsi_reset_notify() is used by a target driver when it needs to be notified of a bus reset. The bus reset could be issued by the transport layer (e.g., the host bus adapter (HBA) driver or controller) or by another initiator.

The argument flag is used to register or cancel the notification. The supported values for flag are as follows:

SCSI_RESET_NOTIFY | Register callback as the reset notification function for the target driver.

SCSI_RESET_CANCEL | Cancel the reset notification request.

Target drivers can find out whether the HBA driver and controller support reset notification by checking the reset-notification capability using the scsi_ifgetcap(9F) function.

RETURN VALUES | If flag is SCSI_RESET_NOTIFY, scsi_reset_notify() returns:

DDI_SUCCESS | The notification request has been accepted.

DDI_FAILURE | The transport layer does not support reset notification or could not accept this request.

If flag is SCSI_RESET_CANCEL, scsi_reset_notify() returns:

DDI_SUCCESS | The notification request has been canceled.

DDI_FAILURE | No notification request was registered.

CONTEXT | scsi_reset_notify() can be called from user or interrupt context.

SEE ALSO | scsi_address(9S), scsi_ifgetcap(9F)

Writing Device Drivers
The scsi_pktalloc(), scsi_pktfree(), scsi_resalloc(), and scsi_resfree() functions are obsolete. The scsi_pktalloc() and scsi_resalloc() functions have been replaced by scsi_init_pkt(9F). The scsi_pktfree() and scsi_resfree() functions have been replaced by scsi_destroy_pkt(9F).

### Parameters

- **ap**: Pointer to a scsi_address structure.
- **cmdlen**: The required length for the SCSI command descriptor block (CDB) in bytes.
- **statuslen**: The required length for the SCSI status completion block (SCB) in bytes.
- **dmatoken**: Pointer to an implementation-dependent object.
- **callback**: A pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.
- **pkt**: Pointer to a scsi_pkt(9S) structure.

### Description

**scsi_pktalloc()** requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, scsi_resalloc() should be used.

**ap** is a pointer to a scsi_address structure. Allocator routines use it to determine the associated host adapter.

**cmdlen** is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the pkt_cdbp field of the allocated scsi_pkt structure.

**statuslen** is the required length for the SCSI status completion block. The address of the allocated block is placed into the pkt_scbp field of the scsi_pkt structure.

**dmatoken** is a pointer to an implementation-dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The dmatoken must be a pointer to a buf(9S) structure. If dmatoken is NULL, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data
transfer phase (either no data transfer phase expected, or the wrong direction encountered), the command is terminated with the(pkt_reason set to CMD_DMA_DERR. dm atoken provides the information to determine if the transfer count is correct.

`callback` indicates what the allocator routines should do when resources are not available:

- **NULL_FUNC**: Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC**: Wait indefinitely for resources.
- **Other Values**: `callback` points to a function which is called when resources may have become available. `callback` must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

`scsi_pktfree()` frees the packet.

`scsi_resfree()` free all resources held by the packet and the packet itself.

**RETURN VALUES**
Both allocation routines return a pointer to a `scsi_pkt` structure on success, or NULL on failure.

**CONTEXT**
If `callback` is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

**ATTRIBUTES**
See attributes(5) for a description of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**
attributes(5), scsi_dmafree(9F), scsi_dmaget(9F), buf(9S), scsi_pkt(9S)

Writing Device Drivers

**NOTES**
The `scsi_pktalloc()`, `scsi_pktfree()`, `scsi_resalloc()`, and `scsi_resfree()` functions are obsolete and will be discontinued in a future release. The `scsi_pktalloc()` and `scsi_resalloc()` functions have been replaced by `scsi_init_pkt(9F)`. The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by `scsi_destroy_pkt(9F)`.
**NAME**
scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

**SYNOPSIS**
```
#include <sys/scsi/scsi.h>

char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- **cmd**
  A SCSI command value.
- **cmdvec**
  Pointer to an array of command strings.
- **dtype**
  Device type.
- **msg**
  A message value.
- **reason**
  A packet reason value.
- **sense_key**
  A SCSI sense key value.

**DESCRIPTION**
- `scsi_cname()` decodes SCSI commands. `cmdvec` is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.
- `scsi_dname()` decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.
- `scsi_mname()` decodes SCSI messages.
- `scsi_rname()` decodes packet completion reasons.
- `scsi_sname()` decodes SCSI sense keys.

**RETURN VALUES**
These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

**CONTEXT**
These functions can be called from user or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Decoding SCSI tape commands.

`scsi_cname()` decodes SCSI tape commands as follows:
```
static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
    "\003request sense",
    "\010read",
    "\012write",
    "\020write file mark",
    ...}
```

Kernel Functions for Drivers 1449
EXAMPLE 1 Decoding SCSI tape commands.  (Continued)

        "\021space",
        "\022inquiry",
        "\025mode select",
        "\031erase tape",
        "\032mode sense",
        "\033load tape",
        NULL
    }

    ..
    cmn_err(CE_CONT, "st: cmd=\%s", scsi_cname(cmd, st_cmds));
    ..

SEE ALSO  Writing Device Drivers
### NAME

scsi_setup_cdb – setup SCSI command descriptor block (CDB)

### SYNOPSIS

```c
int scsi_setup_cdb(union scsi_cdb *cdbp, uchar_t cmd, uint_t addr,
                   uint_t cnt, uint_t othr_cdb_data);
```

### INTERFACE LEVEL PARAMETERS

- **cdbp**: Pointer to command descriptor block.
- **cmd**: The first byte of the SCSI group 0, 1, 2, 4, or 5 CDB.
- **addr**: Pointer to the location of the data.
- **cnt**: Data transfer length in units defined by the SCSI device type. For sequential devices, `cnt` is the number of bytes. For block devices, `cnt` is the number of blocks.
- **othr_cdb_data**: Additional CDB data.

### DESCRIPTION

`scsi_setup_cdb()` function initializes a group 0, 1, 2, 4, or 5 type of command descriptor block pointed to by `cdbp` using `cmd`, `addr`, `cnt`, `othr_cdb_data`.

- `addr` should be set to 0 for commands having no addressing information (for example, group 0 READ command for sequential access devices). `othr_cdb_data` should be additional CDB data for Group 4 commands; otherwise, it should be set to 0.

- `scsi_setup_cdb()` function does not set the LUN bits in CDB[1] as the `makecom(9F)` functions do. Also, the fixed bit for sequential access device commands is not set.

### RETURN VALUES

`scsi_setup_cdb()` returns:

- `1` Upon success.
- `0` Upon failure.

### CONTEXT

These functions can be called from a user or interrupt context.

### SEE ALSO

- `makecom(9F)`, `scsi_pkt(9S)`
- **Writing Device Drivers**
  - *American National Standard SCSI-3 Primary Commands (SPC)*

---

**Kernel Functions for Drivers** 1451
### NAME
scsi_slave – utility for SCSI target drivers to establish the presence of a target

### SYNOPSIS
```
#include <sys/scsi/scsi.h>

int scsi_slave(struct scsi_device *devp, int (*callback)(void));
```

### INTERFACE LEVEL
The scsi_slave() function is obsolete. This function has been replaced by scsi_probe(9F).

### PARAMETERS
- **devp** Pointer to a scsi_device(9S) structure.
- **callback** Pointer to a callback function, NULL_FUNC or SLEEP_FUNC.

### DESCRIPTION
scsi_slave() checks for the presence of a SCSI device. Target drivers may use this function in their probe(9E) routines. scsi_slave() determines if the device is present by using a Test Unit Ready command followed by an Inquiry command. If scsi_slave() is successful, it will fill in the scsi_inquiry structure, which is the sd_inq member of the scsi_device(9S) structure, and return SCSI_PROBE_EXISTS. This information can be used to determine if the target driver has probed the correct SCSI device type. *callback* indicates what the allocator routines should do when DMA resources are not available:

- **NULL_FUNC** Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC** Wait indefinitely for resources.
- **Other Values** *callback* points to a function which is called when resources may have become available. *callback* must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

### RETURN VALUES
scsi_slave() returns:
- **SCSI_PROBE_NOMEM** No space available for structures.
- **SCSI_PROBE_EXISTS** Device exists and inquiry data is valid.
- **SCSI_PROBE_NONCCS** Device exists but inquiry data is not valid.
- **SCSI_PROBE_FAILURE** Polled command failure.
- **SCSI_PROBE_NORESP** No response to TEST UNIT READY.

### CONTEXT
scsi_slave() is normally called from the target driver’s probe(9E) or attach(9E) routine. In any case, this routine should not be called from interrupt context, because it can sleep waiting for memory to be allocated.

### ATTRIBUTES
See attributes(5) for a description of the following attributes:


<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

**SEE ALSO**
attributes(5), attach(9E), probe(9E), ddi_iopb_alloc(9F), makecom(9F), scsi_dmaget(9F), scsi_ifgetcap(9F), scsi_pktalloc(9F), scsi_poll(9F), scsi_probe(9F), scsi_device(9S)

**ANSI Small Computer System Interface-2 (SCSI-2)**

**Writing Device Drivers**

**NOTES**
The `scsi_slave()` function is obsolete and will be discontinued in a future release. This function has been replaced by `scsi_probe(9F)`.
scsi_sname(9F)

NAME
scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

SYNOPSIS
#include <sys/scsi/scsi.h>
char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

DESCRIPTION
scsi_cname() decodes SCSI commands. cmdvec is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

scsi_dname() decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

scsi_mname() decodes SCSI messages.

scsi_rname() decodes packet completion reasons.

scsi_sname() decodes SCSI sense keys.

RETURN VALUES
These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

CONTEXT
These functions can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Decoding SCSI tape commands.

scsi_cname() decodes SCSI tape commands as follows:

static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
    "\003request sense",
    "\010read",
    "\012write",
    "\020write file mark",
    "\022extend write",
    "\023extend read",
    "\026verify",
    "\027verify logical",
    "\028verify extended",
    "\029verify block",
    "\030scan"
};

NAME
SYNOPSIS
LEVEL
PARAMETERS
DESCRIPTION
RETURN VALUES
CONTEXT
EXAMPLES
EXAMPLE 1 Decoding SCSI tape commands. (Continued)

    "\021space",
    "\022inquiry",
    "\025mode select",
    "\031erase tape",
    "\032mode sense",
    "\033load tape",
    NULL
};

    cmn_err(CR_CONT, "st: cmd=%s", scsi_cname(cmd, st_cmds));

SEE ALSO Writing Device Drivers
### Name

`scsi_sync_pkt` - synchronize CPU and I/O views of memory

### Synopsis

```c
#include <sys/scsi/scsi.h>

void scsi_sync_pkt(struct scsi_pkt *pktp);
```

### Interface Level

Solaris DDI specific (Solaris DDI).

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pktp</code></td>
<td>Pointer to a <code>scsi_pkt(9S)</code> structure.</td>
</tr>
</tbody>
</table>

### Description

`scsi_sync_pkt()` is used to selectively synchronize a CPU’s or device’s view of the data associated with the SCSI packet that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When a SCSI packet is mapped for I/O using `scsi_init_pkt(9F)` and destroyed using `scsi_destroy_pkt(9F)`, then an implicit `scsi_sync_pkt()` will be performed. However, if the memory object has been modified by either the device or a CPU after the mapping by `scsi_init_pkt(9F)`, then a call to `scsi_sync_pkt()` is required.

If the same `scsi_pkt` is reused for a data transfer from memory to a device, then `scsi_sync_pkt()` must be called before calling `scsi_transport(9F)`. If the same packet is reused for a data transfer from a device to memory `scsi_sync_pkt()` must be called after the completion of the packet but before accessing the data in memory.

### Context

`scsi_sync_pkt()` may be called from user or interrupt context.

### See Also

- `tran_sync_pkt(9E)`, `ddi_dma_sync(9F)`, `scsi_destroy_pkt(9F)`, `scsi_init_pkt(9F)`, `scsi_transport(9F)`, `scsi_pkt(9S)`

*Writing Device Drivers*
NAME
scsi_transport – request by a SCSI target driver to start a command

SYNOPSIS
#include <sys/scsi/scsi.h>

int scsi_transport(struct scsi_pkt *pkt);

INTERFACE
Solaris DDI specific (Solaris DDI).

PARAMETERS
pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION
Target drivers use scsi_transport() to request the host adapter driver to transport
a command to the SCSI target device specified by pkt. The target driver must obtain
resources for the packet using scsi_init_pkt(9F) prior to calling this function. The
packet may be initialized using one of the makecom(9F) functions.
scsi_transport() does not wait for the SCSI command to complete. See
scsi_poll(9F) for a description of polled SCSI commands. Upon completion of the
SCSI command the host adapter calls the completion routine provided by the target
driver in the pkt_comp member of the scsi_pkt pointed to by pkt.

RETURN VALUES
scsi_transport() returns:
TRAN_ACCEPT The packet was accepted by the transport layer.
TRAN_BUSY The packet could not be accepted because there was
already a packet in progress for this target/lun, the
host adapter queue was full, or the target device queue
was full.
TRAN_BADPKT The DMA count in the packet exceeded the DMA
engine’s maximum DMA size.
TRAN_FATAL_ERROR A fatal error has occurred in the transport layer.

CONTEXT
scsi_transport() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using scsi_transport()

if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {
    scsi_log(pdev, sd_label, CE_WARN,
             "transport of request sense pkt fails (0x%x)\n", status);
}

SEE ALSO
tran_start(9E), makecom(9F), scsi_init_pkt(9F), scsi_pktalloc(9F),
scsi_poll(9F), scsi_pkt(9S)

Writing Device Drivers
scsi_unprobe(9F)

NAME  scsi_unprobe, scsi_unslave – free resources allocated during initial probing

SYNOPSIS  
#include <sys/scsi/scsi.h>

void scsi_unslave(struct scsi_device *devp);
void scsi_unprobe(struct scsi_device *devp);

INTERFACE
Solaris DDI specific (Solaris DDI). The scsi_unslave() interface is obsolete. Use
scsi_unprobe() instead.

LEVEL
PARAMETERS  devp  Pointer to a scsi_device(9S) structure.

DESCRIPTION  scsi_unprobe() and scsi_unslave() are used to free any resources that were
allocated on the driver’s behalf during scsi_slave(9F) and scsi_probe(9F)
activity.

CONTEXT  scsi_unprobe() and scsi_unslave() must not be called from an interrupt
context.

SEE ALSO  scsi_probe(9F), scsi_slave(9F), scsi_device(9S)

Writing Device Drivers

NOTES  The scsi_unslave() function is obsolete and will be discontinued in a future
release. This function has been replaced by scsi_unprobe().
scsi_unslave(9F)

NAME  
scci_unprobe, scsi_unslave – free resources allocated during initial probing

SYNOPSIS  
#include <sys/scsi/scsi.h>

void scsi_unslave(struct scsi_device *devp);
void scsi_unprobe(struct scsi_device *devp);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI). The scsi_unslave() interface is obsolete. Use scsi_unprobe() instead.

PARAMETERS  
devp  Pointer to a scsi_device(9S) structure.

DESCRIPTION  
scsi_unprobe() and scsi_unslave() are used to free any resources that were allocated on the driver’s behalf during scsi_slave(9F) and scsi_probe(9F) activity.

CONTEXT  
scsi_unprobe() and scsi_unslave() must not be called from an interrupt context.

SEE ALSO  
scsi_probe(9F), scsi_slave(9F), scsi_device(9S)

Writing Device Drivers

NOTES  
The scsi_unslave() function is obsolete and will be discontinued in a future release. This function has been replaced by scsi_unprobe().
The following parameters are supported:

devp
   Pointer to the `scsi_device(9S)` structure.

pktp
   Pointer to a `scsi_pkt(9S)` structure.

drv_name
   String used by `scsi_log(9F)`.

severity
   Error severity level, maps to severity strings below.

blkno
   Requested block number.

err_blkno
   Error block number.

cmdlist
   An array of SCSI command description strings.

sensep
   A pointer to a `scsi_extended_sense(9S)` structure.

asc_list
   A pointer to a array of asc and ascq message list. The list must be terminated with -1 asc value.

decode_fru
   This is a function pointer that will be called after the entire sense information has been decoded. The parameters will be the `scsi_device` structure to identify the device. Second argument will be a pointer to a buffer of length specified by third argument. The fourth argument will be the FRU byte. `decode_fru` might be NULL if no special decoding is required. `decode_fru` is expected to return pointer to a char string if decoding possible and NULL if no decoding is possible.

This function is very similar to `scsi_errmsg(9F)` but allows decoding of vendor-unique ASC/ASCQ and FRU information.
scsi_vu_errmsg() interprets the request sense information in the sensep pointer and generates a standard message that is displayed using scsi_log(9F). It first searches the list array for a matching vendor unique code if supplied. If it does not find one in the list then the standard list is searched. The first line of the message is always a CE_WARN, with the continuation lines being CE_CONT. sensep may be NULL, in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The scsi_device(9S) structure denoted by devp supplies the identification of the device that requested the display. severity selects which string is used in the “Error Level:” reporting, according to the table below:

<table>
<thead>
<tr>
<th>Severity Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_ERR_ALL</td>
<td>All</td>
</tr>
<tr>
<td>SCSI_ERR_UNKNOWN</td>
<td>Unknown</td>
</tr>
<tr>
<td>SCSI_ERR_INFO</td>
<td>Information</td>
</tr>
<tr>
<td>SCSI_ERR_RECOVERED</td>
<td>Recovered</td>
</tr>
<tr>
<td>SCSI_ERR_RETRYABLE</td>
<td>Retryable</td>
</tr>
<tr>
<td>SCSI_ERR_FATAL</td>
<td>Fatal</td>
</tr>
</tbody>
</table>

blkno is the block number of the original request that generated the error. err_blkno is the block number where the error occurred. cmdlist is a mapping table for translating the SCSI command code in pktp to the actual command string.

The cmdlist is described in the structure below:

```c
struct scsi_key_strings {
    int key;
    char *message;
};
```

For a basic SCSI disk, the following list is appropriate:

```c
static struct scsi_key_strings scsi_cmds[] = {
    0x00, "test unit ready",
    0x01, "rezero/rewind",
    0x03, "request sense",
    0x04, "format",
    0x07, "reassign",
    0x08, "read",
    0x0a, "write",
    0x0b, "seek",
    0x12, "inquiry",
    0x15, "mode select",
    0x16, "reserve",
    0x17, "release",
    0x18, "copy",
    0x1a, "mode sense",
    0x1b, "start/stop",
    0x1e, "door lock",
    0x28, "read(10)",
    0x2a, "write(10)",
    0x2f, "verify",
};
```
**EXCEPTIONS**

**EXAMPLES**

**EXAMPLE 1 Using `scsi_vu_errmsg()`**

```c
    struct scsi_asc_key_strings cd_list[] = {
        0x81, 0, "Logical Unit is inaccessible",
        -1, 0, NULL,
    };

    scsi_vu_errmsg(devp, pkt, "sd",
                   SCSI_ERR_INFO, bp->b_blkno, err_blkno,
                   sd_cmds, rqsense, cd_list,
                   my_decode_fru);
```

This generates the following console warning:

```
WARNING: /sbus@1,f8000000/esp@0,800000/sd@1,0 (sd1):
Error for Command: read  Error Level: Informational
Requested Block: 23936  Error Block: 23936
Vendor: XYZ  Serial Number: 123456
Sense Key: Unit Attention
ASC: 0x81 (Logical Unit is inaccessible), ASCQ: 0x0
FRU: 0x11 (replace LUN 1, located in slot 1)
```

**SEE ALSO**

`cmn_err(9F), scsi_errmsg(9F), scsi_log(9F), scsi_errmsg(9F),
scsi_asc_key_strings(9S), scsi_device(9S), scsi_extended_sense(9S),
scsi_pkt(9S)`

`Writing Device Drivers`

`STREAMS Programming Guide`
semaphore functions

#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);

Solaris DDI specific (Solaris DDI).

sp A pointer to a semaphore, type ksema_t.

val Initial value for semaphore.

name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)

type Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.

arg Type-specific argument; should be NULL.

These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by sema_p() and atomically incremented by sema_v(). The value must always be greater than or equal to zero. If sema_p() is called and the value is zero, the calling thread is blocked until another thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by sema_init(). For this reason, sema_destroy() should be called before deallocating the storage containing the semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(). However, if the semaphore value is zero, sema_p_sig() will return without decrementing the value if a signal (that is, from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and will not block.

RETURN VALUES

0 sema_tryp() could not decrement the semaphore value because it was zero.
sema_destroy(9F)

| 1  | sema_p_sig() was not able to decrement the semaphore value and detected a pending signal. |

**CONTEXT**

These functions can be called from user or interrupt context, except for `sema_init()` and `sema_destroy()`, which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, `sema_v()` and `sema_p()` should not be called from any interrupt context.

If `sema_p()` is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the `sema_v()` becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the `sema_v()` may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

**SEE ALSO**

`kill(2)`, `condvar(9F)`, `mutex(9F)`

*Writing Device Drivers*
**NAME**
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp –
semaphore functions

**SYNOPSIS**
```
#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);
```

**INTERFACE LEVEL PARAMETERS**
- sp: A pointer to a semaphore, type ksema_t.
- val: Initial value for semaphore.
- name: Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
- type: Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.
- arg: Type-specific argument; should be NULL.

**DESCRIPTION**
These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by `sema_p()` and atomically incremented by `sema_v()`. The value must always be greater than or equal to zero. If `sema_p()` is called and the value is zero, the calling thread is blocked until another thread performs a `sema_v()` operation on the semaphore.

Semaphores are initialized by calling `sema_init()`. The argument, `val`, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by `sema_init()`. For this reason, `sema_destroy()` should be called before deallocating the storage containing the semaphore.

`sema_p_sig()` decrements the semaphore, as does `sema_p()`. However, if the semaphore value is zero, `sema_p_sig()` will return without decrementing the value if a signal (that is, from `kill(2)`) is pending for the thread.

`sema_tryp()` will decrement the semaphore value only if it is greater than zero, and will not block.

**RETURN VALUES**
- 0: `sema_tryp()` could not decrement the semaphore value because it was zero.
sema_init(9F)

1 sema_p_sig() was not able to decrement the semaphore value and detected a pending signal.

CONTEXT These functions can be called from user or interrupt context, except for sema_init() and sema_destroy(), which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, sema_v() and sema_p() should not be called from any interrupt context.

If sema_p() is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the sema_v() becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the sema_v() may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO kill(2), condvar(9F), mutex(9F)

Writing Device Drivers
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

SYNOPSIS
#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);
void sema_destroy(ksema_t *sp);
void sema_p(ksema_t *sp);
void sema_v(ksema_t *sp);
int sema_p_sig(ksema_t *sp);
int sema_tryp(ksema_t *sp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).
PARAMETERS
sp A pointer to a semaphore, type ksema_t.
val Initial value for semaphore.
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
type Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.
arg Type-specific argument; should be NULL.

DESCRIPTION
These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by sema_p() and atomically incremented by sema_v(). The value must always be greater than or equal to zero. If sema_p() is called and the value is zero, the calling thread is blocked until another thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by sema_init(). For this reason, sema_destroy() should be called before deallocating the storage containing the semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(). However, if the semaphore value is zero, sema_p_sig() will return without decrementing the value if a signal (that is, from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and will not block.

RETURN VALUES
0 sema_tryp() could not decrement the semaphore value because it was zero.
sema_p(9F)

1  sema_p_sig() was not able to decrement the semaphore value and
detected a pending signal.

CONTEXT  These functions can be called from user or interrupt context, except for sema_init() and sema_destroy(), which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, sema_v() and sema_p() should not be called from any interrupt context.

If sema_p() is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the sema_v() becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the sema_v() may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO  kill(2), condvar(9F), mutex(9F)

Writing Device Drivers
semaphore(9F)

NAME  semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp –
       semaphore functions

SYNOPSIS  
#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t
type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);

INTERFACE

LEVEL  Solaris DDI specific (Solaris DDI).

PARAMETERS

sp   A pointer to a semaphore, type ksema_t.

val  Initial value for semaphore.

name  Descriptive string. This is obsolete and should be NULL.
       (Non-NULL strings are legal, but they are a waste of kernel
       memory.)

type  Variant type of the semaphore. Currently, only SEMA_DRIVER is
       supported.

arg   Type-specific argument; should be NULL.

DESCRIPTION  These functions implement counting semaphores as described by Dijkstra. A
semaphore has a value which is atomically decremented by sema_p() and atomically
incremented by sema_v(). The value must always be greater than or equal to zero. If
sema_p() is called and the value is zero, the calling thread is blocked until another
thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the
initial value for the semaphore. The semaphore storage is provided by the caller but
more may be dynamically allocated, if necessary, by sema_init(). For this reason,
sema_destroy() should be called before deallocating the storage containing the
semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(). However, if the
semaphore value is zero, sema_p_sig() will return without decrementing the value
if a signal (that is, from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and
will not block.

RETURN VALUES

0     sema_tryp() could not decrement the semaphore value because it was
       zero.
sema_p() was not able to decrement the semaphore value and detected a pending signal.

These functions can be called from user or interrupt context, except for sema_init() and sema_destroy(), which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, sema_v() and sema_p() should not be called from any interrupt context.

If sema_p() is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the sema_v() becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the sema_v() may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO kill(2), condvar(9F), mutex(9F)

Writing Device Drivers
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);

Solaris DDI specific (Solaris DDI).

INTERFACE LEVEL

PARAMETERS

| sp | A pointer to a semaphore, type ksema_t. |
| val | Initial value for semaphore. |
| name | Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.) |
| type | Variant type of the semaphore. Currently, only SEMA_DRIVER is supported. |
| arg | Type-specific argument; should be NULL. |

DESCRIPTION

These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by sema_p() and atomically incremented by sema_v(). The value must always be greater than or equal to zero. If sema_p() is called and the value is zero, the calling thread is blocked until another thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by sema_init(). For this reason, sema_destroy() should be called before deallocating the storage containing the semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(). However, if the semaphore value is zero, sema_p_sig() will return without decrementing the value if a signal (that is, from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and will not block.

RETURN VALUES

0 sema_tryp() could not decrement the semaphore value because it was zero.
sema_p_sig() was not able to decrement the semaphore value and detected a pending signal.

These functions can be called from user or interrupt context, except for `sema_init()` and `sema_destroy()`, which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, `sema_v()` and `sema_p()` should not be called from any interrupt context.

If `sema_p()` is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the `sema_v()` becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the `sema_v()` may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO `kill(2)`, `condvar(9F)`, `mutex(9F)`

Writing Device Drivers
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp — semaphore functions

#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);

Solaris DDI specific (Solaris DDI).

sp A pointer to a semaphore, type ksema_t.

val Initial value for semaphore.

name Descriptive string. This is obsolete and should be NULL.
(Non-NULL strings are legal, but they are a waste of kernel memory.)

type Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.

arg Type-specific argument; should be NULL.

These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by sema_p() and atomically incremented by sema_v(). The value must always be greater than or equal to zero. If sema_p() is called and the value is zero, the calling thread is blocked until another thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by sema_init(). For this reason, sema_destroy() should be called before deallocating the storage containing the semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(). However, if the semaphore value is zero, sema_p_sig() will return without decrementing the value if a signal (that is, from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and will not block.

0 sema_tryp() could not decrement the semaphore value because it was zero.
sema_tryp(9F)

1    sema_p_sig() was not able to decrement the semaphore value and
detected a pending signal.

CONTEXT

These functions can be called from user or interrupt context, except for sema_init() and sema_destroy(), which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, sema_v() and sema_p() should not be called from any interrupt context.

If sema_p() is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the sema_v() becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the sema_v() may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO

kill(2), condvar(9F), mutex(9F)

Writing Device Drivers
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);

Solaris DDI specific (Solaris DDI).

**NAME**
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

**SYNOPSIS**

```c
#include <sys/ksynch.h>

void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type, void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_tryp(ksema_t *sp);
```

**INTERFACE LEVEL PARAMETERS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sp</code></td>
<td>A pointer to a semaphore, type ksema_t.</td>
</tr>
<tr>
<td><code>val</code></td>
<td>Initial value for semaphore.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)</td>
</tr>
<tr>
<td><code>type</code></td>
<td>Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Type-specific argument; should be NULL.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**
These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by `sema_p()` and atomically incremented by `sema_v()`. The value must always be greater than or equal to zero. If `sema_p()` is called and the value is zero, the calling thread is blocked until another thread performs a `sema_v()` operation on the semaphore.

Semaphores are initialized by calling `sema_init()`. The argument, `val`, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by `sema_init()`. For this reason, `sema_destroy()` should be called before deallocating the storage containing the semaphore.

`sema_p_sig()` decrements the semaphore, as does `sema_p()`. However, if the semaphore value is zero, `sema_p_sig()` will return without decrementing the value if a signal (that is, from `kill(2)`) is pending for the thread.

`sema_tryp()` will decrement the semaphore value only if it is greater than zero, and will not block.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>sema_tryp()</code> could not decrement the semaphore value because it was zero.</td>
</tr>
</tbody>
</table>
sema_p(9F)

1    sema_pSIG() was not able to decrement the semaphore value and
detected a pending signal.

CONTEXT

These functions can be called from user or interrupt context, except for sema_init() and sema_destroy(), which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, sema_v() and sema_p() should not be called from any interrupt context.

If sema_p() is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the sema_v() becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the sema_v() may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO

kill(2), condvar(9F), mutex(9F)

Writing Device Drivers
SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL (structname, handle);
STRUCT_HANDLE (structname, handle);
void STRUCT_INIT (handle, model_t umodel);
void STRUCT_SET_HANDLE (handle, model_t umodel, void *addr);
STRUCT_FGET (handle, field);
STRUCT_FGETP (handle, field);
STRUCT_FSET (handle, field, val);
STRUCT_FSETP (handle, field, val);
<typeof field> *STRUCT_FADDR (handle, field);
struct structname *STRUCT_BUF (handle);
size_t SIZEOF_STRUCT (structname, umodel);
size_t SIZEOF_PTR (umodel);
size_t STRUCT_SIZE (handle);

INTERFACE
LEVEL
PARAMETERS
Solaris DDI specific (Solaris DDI).

The macros take the following parameters:

structname
The structure name (as would appear after the C keyword “struct”) of the native form.

umodel
A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

handle
The variable name used to refer to a particular instance of a structure which is handled by these macros.

field
The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

<table>
<thead>
<tr>
<th>Declaration and Initialization Macros</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCT_DECL(structname, handle)</td>
<td>Declares a &quot;structure handle&quot; for a &quot;struct&quot; and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. handle is a variable name and is declared as a variable by this macro.</td>
</tr>
<tr>
<td>void STRUCT_INIT(handle, model_t umodel)</td>
<td>Initializes handle to point to the instance allocated by STRUCT_DECL(), it also sets data model for handle to umodel, and must be called before any access is made through the macros that operate on these structures. When used in an ioctl(9E) routine umodel is the flag parameter; in devmap(9E) routine umodel is the model parameter and in a mmap(9E) routine, is the return value of ddi_mmap_get_model(9F). This macro is intended for handles created with STRUCT_DECL() only.</td>
</tr>
<tr>
<td>STRUCT_HANDLE(structname, handle)</td>
<td>Declares a &quot;structure handle&quot; handle but unlike STRUCT_DECL() does not allocate an instance of &quot;struct&quot;.</td>
</tr>
<tr>
<td>void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)</td>
<td>Initializes to point to the native form instance at addr, it also sets the data model for handle to umodel. This is intended for handles created with STRUCT_HANDLE(). Fields cannot be referenced via the handle until this macro has been invoked. Typically, addr is the address of the native form structure containing the user-mode programs data. When used in an ioctl(9E) umodel is the flag parameter, in a devmap(9E) routine is the model parameter and in a mmap(9E) routine, umodel is the return value of ddi_mmap_get_model(9F).</td>
</tr>
</tbody>
</table>
Operation Macros

- `size_t STRUCT_SIZE(handle)`
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- `STRUCT_FGET(handle, field)`
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- `STRUCT_FGETP(handle, field)`
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- `STRUCT_FSET(handle, field, val)`
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- `STRUCT_FSETP(handle, field, val)`
  Returns a pointer to the in the structure described by `handle`.

- `struct structname *STRUCT_BUF(handle)`
  Returns a pointer to the native mode instance of the structure described by `handle`.

ProtoNot Using Handles

- `size_t SIZEOF_STRUCT(structname, umodel)`
  Returns size of `structname` based on `umodel`.

- `size_t SIZEOF_PTR(umodel)`
  Returns the size of a pointer based on `umodel`.

EXAMPLES

**Example 1** Copying a Structure

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**Example 2** Defining a Structure

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTL_DATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int llen; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_ioctldata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
}
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

\[
\text{STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,}
\text{(void *)data->b_rptr);}
\text{if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t))}
\text{  \{}
\text{      \_mi_copy_done(q, msgp, EINVAL);}
\text{      return;}
\text{  \} }
\]

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
SIZEOF_STRUCT(9F)

NAME

STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR,
STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP,
STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32–bit
application data access macros

SYNOPSIS

#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);
void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);
STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);
STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);
<typeof field> *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);
size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);

INTERFACE

Solaris DDI specific (Solaris DDI).

LEVEL

PARAMETERS

The macros take the following parameters:

structname

The structure name (as would appear after the C keyword “struct”) of the native
form.

umodel

A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64
model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the
flag parameter; in a devmap(9E), they will be present in the model parameter
mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the
current thread.

handle

The variable name used to refer to a particular instance of a structure which is
handled by these macros.

field

The field name within the structure contain substructures. If the structures contain
substructures, unions, or arrays, then field can be whether complex expression could
occur after the first “.” or “->”.

1482     man pages section 9: DDI and DKI Kernel Functions • Last Revised 23 Feb 1998
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

```c
void STRUCT_INIT(handle, model_t umodel)
```

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 

```c
void STRUCT_DECL(structname, handle)
```

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_HANDLE(structname, handle)
```

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct “. 

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 

```c
void STRUCT_DECL(structname, handle)
```

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_HANDLE(structname, handle)
```

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct “. 

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 

```c
void STRUCT_DECL(structname, handle)
```

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_HANDLE(structname, handle)
```

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct “. 

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 

```c
void STRUCT_DECL(structname, handle)
```

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_HANDLE(structname, handle)
```

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct “. 

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```
SIZEOF_STRUCT(9F)

**Operation Macros**

- **size_t STRUCT_SIZE(handle)**
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- **STRUCT_FGET(handle, field)**
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- **STRUCT_FGETP(handle, field)**
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- **STRUCT_FSET(handle, field, val)**
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- **STRUCT_FSETP(handle, field, val)**
  Returns a pointer to the in the structure described by `handle`.

**Macros Not Using Handles**

- **size_t SIZEOF_STRUCT(structname, umodel)**
  Returns size of `structname` based on `umodel`.

- **size_t SIZEOF_PTR(umodel)**
  Returns the size of a pointer based on `umodel`.

**EXAMPLES**

**EXAMPLE 1 Copying a Structure**

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2 Defining a Structure**

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using \texttt{STRUCT\_DECL()} and \texttt{STRUCT\_INIT()}

Note: This example uses the \texttt{STRUCT\_DECL()} and \texttt{STRUCT\_INIT()} macros to declare and initialize the structure handle.

```c
int xxiocctl(dev_t dev, int cmd, intptr_t arg, int mode,
               cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                    STRUCT\_BUF(op), STRUCT\_SIZE(op)))
        return (EFAULT);
    if (STRUCT\_FGET(op, flag) != FACTIVE ||
               STRUCT\_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT\_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles \texttt{ioctl(9E)} data (M\_IOCDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using \texttt{STRUCT\_HANDLE()} and \texttt{STRUCT\_SET\_HANDLE()}

The next example illustrates the use of the \texttt{STRUCT\_HANDLE()} and \texttt{STRUCT\_SET\_HANDLE()} macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the \texttt{STRUCT\_HANDLE()} and \texttt{STRUCT\_SET\_HANDLE()} as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT\_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi\_copy\_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4  Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()  (Continued)

    STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
        (void *)data->b_rptr);
    if (STRUCT_FGET(sb, maxlen) < (int)sizeof(ipa_t)) {
        mi_copy_done(q, msgp, EINVAL);
        return;
    }
}

SEE ALSO  devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
NAME | sprintf, snprintf – format characters in memory

SYNOPSIS | #include <sys/ddi.h>

    char *sprintf(char *buf, const char *fmt, ...);
    size_t snprintf(char *buf, size_t n, const char *fmt, ...);

INTERFACE | Solaris DDI specific (Solaris DDI).

LEVEL | PARAMETERS

PARAMETERS | buf                Pointer to a character string.
            | fmt                Pointer to a character string.

DESCRIPTION | sprintf() builds a string in buf under the control of the format fmt. The format is a character string with either plain characters, which are simply copied into buf, or conversion specifications, each of which converts zero or more arguments, again copied into buf. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user’s responsibility to ensure that enough storage is available for buf.

The snprintf() function is identical to sprintf() with the addition of the argument n, which specifies the size of the buffer referred to by buf. The buffer is always terminated with the null byte.

Conversion Specifications | Each conversion specification is introduced by the % character, after which the following appear in sequence:

An optional value specifying a minimum field width for numeric conversion. The converted value will be right-justified and, if it has fewer characters than the minimum, is padded with leading spaces unless the field width is an octal value, then it is padded with leading zeroes.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (longlong) integer argument. An l (ll) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

\texttt{d,D,O,O,x,X,u}  
\text{The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X) or unsigned decimal (u), respectively, and copied. The letters abcdef are used for x conversion. The letters ABCDEF are used for X conversion.}

\texttt{c}  
\text{The character value of argument is copied.}

\texttt{b}  
\text{This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form \texttt{<base>[<arg> ...]}. The base supplies the conversion base for the first argument as a binary value; \textbackslash 10 gives}
octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by < and >. Thus, the following function call would generate reg=3<BitTwo,BitOne>
\n in buf.

```c
    sprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```

p

The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void * when using the %p format specifier.

s

The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.

% Copy a %; no argument is converted.

**RETURN VALUES**

`sprintf()` returns its first argument, `buf`.

`snprintf()` returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. If the value of `n` is less than or equal to 0 on a call to `snprintf()`, the function simply returns the number of characters formatted.

**CONTEXT**

`sprintf()` and `snprintf()` can be called from user or interrupt context.

**SEE ALSO**

*Writing Device Drivers*
NAME | sprintf, snprintf – format characters in memory  
SYNOPSIS | #include <sys/ddi.h>  
| char *sprintf(char *buf, const char *fmt, ...);  
| size_t snprintf(char *buf, size_t n, const char *fmt, ...);  
INTERFACE LEVEL | Solaris DDI specific (Solaris DDI).  
PARAMETERS | buf Pointer to a character string.  
| fmt Pointer to a character string.  
DESCRIPTION | sprintf() builds a string in buf under the control of the format fmt. The format is a character string with either plain characters, which are simply copied into buf, or conversion specifications, each of which converts zero or more arguments, again copied into buf. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user’s responsibility to ensure that enough storage is available for buf.

The snprintf() function is identical to sprintf() with the addition of the argument n, which specifies the size of the buffer referred to by buf. The buffer is always terminated with the null byte.

Conversion Specifications | Each conversion specification is introduced by the % character, after which the following appear in sequence:

An optional value specifying a minimum field width for numeric conversion. The converted value will be right-justified and, if it has fewer characters than the minimum, is padded with leading spaces unless the field width is an octal value, then it is padded with leading zeroes.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (ll) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

d,D,O,o,X,x,u  
The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X) or unsigned decimal (u), respectively, and copied. The letters abcdef are used for x conversion. The letters ABCDEF are used for X conversion.

c  
The character value of argument is copied.

b  
This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form <base> [<arg> . . . ]. The base supplies the conversion base for the first argument as a binary value; \10 gives
octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by < and >. Thus, the following function call would generate reg=3<BitTwo,BitOne>

```c
sprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```

p

The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void* when using the %p format specifier.

s

The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.

% Copy a %; no argument is converted.

**RETURN VALUES**

`sprintf()` returns its first argument, `buf`.

`snprintf()` returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. If the value of `n` is less than or equal to 0 on a call to `snprintf()`, the function simply returns the number of characters formatted.

**CONTEXT**

`sprintf()` and `snprintf()` can be called from user or interrupt context.

**SEE ALSO**

`Writing Device Drivers`
### NAME
stoi, numtos – convert between an integer and a decimal string

### SYNOPSIS
```c
#include <sys/ddi.h>

int stoi(char **str);
void numtos(unsigned long num, char *s);
```

### INTERFACE LEVEL PARAMETERS
- **str**: Pointer to a character string to be converted.
- **num**: Decimal number to be converted to a character string.
- **s**: Character buffer to hold converted decimal number.

### return values
- **stoi()**: Returns the integer value of the string `str`. It handles only positive integers; it does not handle leading minus signs.
- **numtos()**: Converts a `long` into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.

### SEE ALSO
- Writing Device Drivers

### NOTES
- **stoi()**: Can be called from user or interrupt context.


<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>stricmp, strcasecmp, strnicmp, strncmp – compare two null-terminated strings.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYNOPSIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
</tbody>
</table>

```c
int strcmp(const char *s1, const char *s2);
int strcasecmp(const char *s1, const char *s2);
int strnicmp(const char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
```

<table>
<thead>
<tr>
<th>INTERFACE LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1, s2</td>
</tr>
</tbody>
</table>

| n           | Count of characters to be compared. |

| strcmp()    | strcmp() returns 0 if the strings are the same, or the integer value of the expression (*s1 - *s2) for the last characters compared if they differ. |

| strcasecmp(), strnicmp() | The strcasecmp() and strnicmp() functions are case-insensitive versions of strcmp() and strncmp(), respectively, described in this section. They assume the ASCII character set and ignore differences in case when comparing lowercase and uppercase characters. |

| strncmp()    | strncmp() returns 0 if the first n characters of s1 and s2 are the same, or (*s1 - *s2) for the last characters compared if they differ. |

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcmp() returns 0 if the strings are the same, or (*s1 - *s2) for the last characters compared if they differ.</td>
</tr>
</tbody>
</table>

| strncmp() and strnicmp() return values in the same fashion as strcmp() and strncmp(), respectively. |

| strnicmp()    | strnicmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for the last characters compared if they differ. |

<table>
<thead>
<tr>
<th>CONTEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>These functions can be called from user or interrupt context.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SEE ALSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing Device Drivers</td>
</tr>
</tbody>
</table>
NAME | strchr, strrchr – find a character in a string

SYNOPSIS | 
```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

char *strchr(const char *str, int chr);
char *strrchr(const char *str, int chr);
```

INTERFACE LEVEL | Solaris DDI specific (Solaris DDI).

PARAMETERS | 
- `str` | Pointer to a string to be searched.
- `chr` | The character to search for.

DESCRIPTION | 
- `strchr()` | The `strchr()` function returns a pointer to the first occurrence of `chr` in the string pointed to by `str`.
- `strrchr()` | The `strrchr()` function returns a pointer to the last occurrence of `chr` in the string pointed to by `str`.

RETURN VALUES | `strchr()` and `strrchr()` return a pointer to a character, or `NULL`, if the search fails.

CONTEXT | These functions can be called from user or interrupt context.

SEE ALSO | `strcmp(9F)`

Writing Device Drivers
strcmp(9F)

NAME
            strcmp, strcasecmp, strncasecmp, strncmp – compare two null-terminated strings.

SYNOPSIS
            #include <sys/ddi.h>
            int strcmp(const char *s1, const char *s2);
            int strcasecmp(const char *s1, const char *s2);
            int strncasecmp(const char *s1, const char *s2, size_t n);
            int strncmp(const char *s1, const char *s2, size_t n);

INTERFACE
            Solaris DDI specific (Solaris DDI).
LEVEL
PARAMETERS
            s1, s2                  Pointers to character strings.
            n                      Count of characters to be compared.

            strcmp() returns 0 if the strings are the same, or the integer value of the expression
            (*s1 - *s2) for the last characters compared if they differ.

            strcasecmp() and strncasecmp() return values in the same fashion as strcmp() and
            strncmp(), respectively. They assume the ASCII character set and ignore differences in case
            when comparing lowercase and uppercase characters.

            strncmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for
            the last characters compared if they differ.

RETURN VALUES
            strcmp() returns 0 if the strings are the same, or (*s1 - *s2) for the last characters
            compared if they differ.

            strcasecmp() and strncasecmp() return values in the same fashion as strcmp() and
            strncmp(), respectively.

            strncmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for
            the last characters compared if they differ.

CONTEXT
            These functions can be called from user or interrupt context.

SEE ALSO
            Writing Device Drivers
NULL

**NAME**
strcpy, strncpy – copy a string from one location to another.

**SYNOPSIS**
```c
#include <sys/ddi.h>

char *strcpy(char *dst, char *srs);
char *strncpy(char *dst, char *srs, size_t n);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
dst, srs
Pointers to character strings.

n
Count of characters to be copied.

**strcpy()**
strcpy() copies characters in the string srs to dst, terminating at the first null character in srs, and returns dst to the caller. No bounds checking is done.

**strncpy()**
strncpy() copies srs to dst, null-padding or truncating at n bytes, and returns dst. No bounds checking is done.

**RETURN VALUES**
strcpy() and strncpy() return dst.

**CONTEXT**
strcpy() can be called from user or interrupt context.

**SEE ALSO**
Writing Device Drivers
strlen(9F)

NAME (strlen) – determine the number of non-null bytes in a string

SYNOPSIS

```c
#include <sys/ddi.h>

size_t strlen(const char *s);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

PARAMETERS

- **s**: Pointer to a character string.

DESCRIPTION

`strlen()` returns the number of non-null bytes in the string argument `s`.

RETURN VALUES

`strlen()` returns the number of non-null bytes in `s`.

CONTEXT

`strlen()` can be called from user or interrupt context.

SEE ALSO

- Writing Device Drivers

---

1496  man pages section 9: DDI and DKI Kernel Functions • Last Revised 11 Apr 1991
NAME
strlog – submit messages to the log driver

SYNOPSIS
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/log.h>

int strlog(short mid, short sid, char level, unsigned short flags, char *
fmt, ...);)

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
mid Identification number of the module or driver submitting the
message (in the case of a module, its mi_idnum value from
module_info(9S)).

sid Identification number for a particular minor device.

level Tracing level for selective screening of low priority messages.
Larger values imply less important information.

flags Valid flag values are:

SL_ERROR Message is for error logger.

SL_TRACE Message is for trace.

SL_NOTIFY Mail copy of message to system administrator.

SL_CONSOLE Log message to console.

SL_FATAL Error is fatal.

SL_WARN Error is a warning.

SL_NOTE Error is a notice.

fmt printf(3C) style format string. %e, %g, and %G formats are not
allowed but %s is supported.

DESCRIPTION
strlog() expands the printf(3C) style format string passed to it, that is, the
conversion specifiers are replaced by the actual argument values in the format string.
The 32–bit representations of the arguments (up to NLORGARGS) follow the string
starting at the next 32–bit boundary following the string. Note that the 64–bit
argument will be truncated to 32–bits here but will be fully represented in the string.

The messages can be retrieved with the getmsg(2) system call. The flags argument
specifies the type of the message and where it is to be sent. strace(1M) receives
messages from the log driver and sends them to the standard output. strerr(1M)
receives error messages from the log driver and appends them to a file called
/var/adm/streams/error.mm-dd, where mm-dd identifies the date of the error
message.

RETURN VALUES
strlog() returns 0 if it fails to submit the message to the log(7D) driver and 1
otherwise.
strlog(9F)

CONTEXT  
strlog() can be called from user or interrupt context.

FILES  
/var/adm/streams/error.mm-dd
   Error messages dated mm-dd appended by strerr(1M) from the log driver

SEE ALSO  
strace(1M), strerr(1M), getmsg(2), log(7D), module_info(9S)

Writing Device Drivers

STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>strcmp, strcasecmp, strncasecmp, strncmp – compare two null-terminated strings.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int strcmp(const char *s1, const char *s2);</td>
</tr>
<tr>
<td></td>
<td>int strcasecmp(const char *s1, const char *s2);</td>
</tr>
<tr>
<td></td>
<td>int strncasecmp(const char *s1, const char *s2, size_t n);</td>
</tr>
<tr>
<td></td>
<td>int strncmp(const char *s1, const char *s2, size_t n);</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>s1, s2 Pointers to character strings.</td>
</tr>
<tr>
<td></td>
<td>n Count of characters to be compared.</td>
</tr>
<tr>
<td>strcmp()</td>
<td>strcmp() returns 0 if the strings are the same, or the integer value of the expression (*s1 - *s2) for the last characters compared if they differ.</td>
</tr>
<tr>
<td>strcasecmp(), strncasecmp()</td>
<td>The strcasecmp() and strncasecmp() functions are case-insensitive versions of strcmp() and strncmp(), respectively, described in this section. They assume the ASCII character set and ignore differences in case when comparing lowercase and uppercase characters.</td>
</tr>
<tr>
<td>strncmp()</td>
<td>strncmp() returns 0 if the first n characters of s1 and s2 are the same, or (*s1 - *s2) for the last characters compared if they differ.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>strcmp() returns 0 if the strings are the same, or (*s1 - *s2) for the last characters compared if they differ.</td>
</tr>
<tr>
<td></td>
<td>strcasecmp() and strncasecmp() return values in the same fashion as strcmp() and strncmp(), respectively.</td>
</tr>
<tr>
<td></td>
<td>strncmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for the last characters compared if they differ.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>These functions can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>Writing Device Drivers</td>
</tr>
</tbody>
</table>
NAME
strcmp, strcasecmp, strncasecmp, strncmp – compare two null-terminated strings.

SYNOPSIS
#include <sys/ddi.h>

int strcmp(const char *s1, const char *s2);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
s1, s2 Points to character strings.
n Count of characters to be compared.

PARAMETERS

strcmp() returns 0 if the strings are the same, or the integer value of the expression (*s1 - *s2) for the last characters compared if they differ.

strcasecmp(),
strncasecmp() The strcasecmp() and strncasecmp() functions are case-insensitive versions of
strcmp() and strncmp(), respectively, described in this section. They assume the
ASCII character set and ignore differences in case when comparing lowercase and
uppercase characters.

strncmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for
the last characters compared if they differ.

RETURN VALUES
strcmp() returns 0 if the strings are the same, or (*s1 - *s2) for the last characters
compared if they differ.

strcasecmp() and strncasecmp() return values in the same fashion as strcmp()
and strncmp(), respectively.

strncmp() returns 0 if the first n characters of strings are the same, or (*s1 - *s2) for
the last characters compared if they differ.

CONTEXT
These functions can be called from user or interrupt context.

SEE ALSO
Writing Device Drivers
**NAME**  
strcpy, strncpy – copy a string from one location to another.

**SYNOPSIS**  
```c
#include <sys/ddi.h>

char *strcpy(char *dst, char *srs);

char *strncpy(char *dst, char *srs, size_t n);
```

**INTERFACE LEVEL**  
Solaris DDI specific (Solaris DDI).

**PARAMETERS**
- `dst`, `srs`  
  Pointers to character strings.
- `n`  
  Count of characters to be copied.

**strcpy()**  
`strcpy()` copies characters in the string `srs` to `dst`, terminating at the first null character in `srs`, and returns `dst` to the caller. No bounds checking is done.

**strncpy()**  
`strncpy()` copies `srs` to `dst`, null-padding or truncating at `n` bytes, and returns `dst`. No bounds checking is done.

**RETURN VALUES**  
`strcpy()` and `strncpy()` return `dst`.

**CONTEXT**  
`strcpy()` can be called from user or interrupt context.

**SEE ALSO**  
*Writing Device Drivers*
**NAME**
strqget – get information about a queue or band of the queue

**SYNOPSIS**
```c
#include <sys/stream.h>

int strqget(queue_t *q, qfields_t what, unsigned char pri, void *valp);
```

**INTERFACE LEVEL PARAMETERS**
Architecture independent level 1 (DDI/DKI).

`q` Pointer to the queue.

`what` Field of the queue structure for (or the specified priority band) to return information about. Valid values are one of:

- `QHIWAT` High water mark.
- `QLOWAT` Low water mark.
- `QMAXPSZ` Largest packet accepted.
- `QMINPSZ` Smallest packet accepted.
- `QCOUNT` Approximate size (in bytes) of data.
- `QFIRST` First message.
- `QLAST` Last message.
- `QFLAG` Status.

`pri` Priority band of interest.

`valp` The address of where to store the value of the requested field.

**DESCRIPTION**
strqget() gives drivers and modules a way to get information about a queue or a particular band of a queue without directly accessing STREAMS data structures, thus insulating them from changes in the implementation of these data structures from release to release.

**RETURN VALUES**
On success, 0 is returned and the value of the requested field is stored in the location pointed to by `valp`. An error number is returned on failure.

**CONTEXT**
strqget() can be called from user or interrupt context.

**SEE ALSO**
strqset(9F), queue(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*
**NAME**
strqset – change information about a queue or band of the queue

**SYNOPSIS**

```c
#include <sys/stream.h>

int strqset(queue_t *q, qfields_t what, unsigned char pri, intptr_t val);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `q`: Pointer to the queue.
- `what`: Field of the queue structure (or the specified priority band) to return information about. Valid values are one of:
  - QHIWAT: High water mark.
  - QLOWAT: Low water mark.
  - QMAXPSZ: Largest packet accepted.
  - QMINPSZ: Smallest packet accepted.
- `pri`: Priority band of interest.
- `val`: The value for the field to be changed.

**DESCRIPTION**
strqset() gives drivers and modules a way to change information about a queue or a particular band of a queue without directly accessing STREAMS data structures.

**RETURN VALUES**
On success, 0 is returned. EINVAL is returned if an undefined attribute is specified.

**CONTEXT**
strqset() can be called from user or interrupt context.

**SEE ALSO**
strqget(9F), queue(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**
When lowering existing values, set QMINPSZ before setting QMAXPSZ; when raising existing values, set QMAXPSZ before setting QMINPSZ.
strchr, strrchr – find a character in a string

NAME  | strchr, strrchr – find a character in a string
SYNOPSIS | include <sys/ddi.h>
 | include <sys/sunddi.h>
 | char *strchr(const char *str, int chr);
 | char *strrchr(const char *str, int chr);

INTERFACE LEVEL
PARAMETERS | Solaris DDI specific (Solaris DDI).
          | str Pointer to a string to be searched.
          | chr The character to search for.
DESCRIPTION | The strchr() function returns a pointer to the first occurrence of chr in the string pointed to by str.
 | strrchr() The strrchr() function returns a pointer to the last occurrence of chr in the string pointed to by str.
RETURN VALUES | strchr() and strrchr() return a pointer to a character, or NULL, if the search fails.
CONTEXT | These functions can be called from user or interrupt context.
SEE ALSO | strcmp(9F)

Writing Device Drivers
The macros take the following parameters:

`structname`

The structure name (as would appear after the C keyword “struct”) of the native form.

`umodel`

A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

`handle`

The variable name used to refer to a particular instance of a structure which is handled by these macros.

`field`

The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then `field` can be whether complex expression could occur after the first “.” or “->”.
**STRUCT_BUF(9F)**

| **DESCRIPTION** | The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver. |
| **Declaration and Initialization Macros** | The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure. |
| | The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctl's within a STREAMS module. |
| | The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program. |
| | The macro `STRUCT_DECL(structname, handle)` declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro. |
| | The macro `void STRUCT_INIT(handle, model_t umodel)` initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only. |
| | The macro `STRUCT_HANDLE(structname, handle)` declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct “. |
| | The macro `void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)` initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. |
**Operation Macros**

- **size_t STRUCT_SIZE(handle)**
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- **STRUCT_FGET(handle, field)**
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- **STRUCT_FGETP(handle, field)**
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast `caddr32_t` to a `(void*)` when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- **STRUCT_FSET(handle, field, val)**
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- **STRUCT_FSETP(handle, field, val)**
  Returns a pointer to the in the structure described by `handle`.

**Macros Not Using Handles**

- **size_t SIZEOF_STRUCT(structname, umodel)**
  Returns size of `structname` based on `umodel`.

- **size_t SIZEOF_PTR(umodel)**
  Returns the size of a pointer based on `umodel`.

**EXAMPLES**

**EXAMPLE 1 Copying a Structure**

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2 Defining a Structure**

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                   STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen;  /* no. of bytes in buffer */
    int len;  /* no. of bytes returned */
    caddr_t buf;  /* pointer to data */
};

static void
wpput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data;  /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```

1508  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 23 Feb 1998
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

(Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
    (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);
void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);
STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);
STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);
< typeof field > *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);
size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);

Solaris DDI specific (Solaris DDI).

The macros take the following parameters:

structname
The structure name (as would appear after the C keyword “struct”) of the native form.

umodel
A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

handle
The variable name used to refer to a particular instance of a structure which is handled by these macros.

field
The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “." or “->".
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

`STRUCT_DECL(structname, handle)`
Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

`void STRUCT_INIT(handle, model_t umodel)`
Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

`STRUCT_HANDLE(structname, handle)`
Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct”.

`void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)`
Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E) umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 
STRUCT_DECL(9F)

**Operation Macros**

- **size_t STRUCT_SIZE(handle)**
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- **STRUCT_FGET(handle, field)**
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- **STRUCT_FGETP(handle, field)**
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast `caddr32_t` to a `(void *)` when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- **STRUCT_FSET(handle, field, val)**
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- **STRUCT_FSETP(handle, field, val)**
  Returns a pointer to the in the structure described by `handle`.

- **struct structname *STRUCT_BUF(handle)**
  Returns a pointer to the native mode instance of the structure described by `handle`.

**Macros Not Using Handles**

- **size_t SIZEOF_STRUCT(structname, umodel)**
  Returns size of `structname` based on `umodel`.

- **size_t SIZEOF_PTR(umodel)**
  Returns the size of a pointer based on `umodel`.

**EXAMPLES**

**EXAMPLE 1** Copying a Structure

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2** Defining a Structure

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```

1512 man pages section 9: DDI and DKI Kernel Functions • Last Revised 23 Feb 1998
**EXAMPLE 3 Using `STRUCT_DECL()` and `STRUCT_INIT()`**

Note: This example uses the `STRUCT_DECL()` and `STRUCT_INIT()` macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int(cmd, intptr_t arg, int mode,
   cred_t *cr, int *rval_p);
{
   STRUCT_DECL(opdata, op);
   if (cmd != OPONE)
      return (ENOTTY);
   STRUCT_INIT(op, mode);
   if (copyin((void *)data,
               STRUCT_BUF(op), STRUCT_SIZE(op)))
      return (EFAULT);
   if (STRUCT_FGET(op, flag) != FACTIVE ||
       STRUCT_FGET(op, size) > sizeof (device_state))
      return (EINVAL);
   xxdowork(device_state, STRUCT_FGET(op, size));
   return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles `ioctl(9E)` data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

**EXAMPLE 4 Using `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()`**

The next example illustrates the use of the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` as follows:

```c
struct strbuf {
   int maxlen; /* no. of bytes in buffer */
   int len; /* no. of bytes returned */
   caddr_t buf; /* pointer to data */
};
static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
   char *data; /* message block descriptor */
   mblk_t *msgp;
   int cmd_len;
   int size;
   caddr_t buf;
   if (mi_copy_state(q, mp, &data) == -1) {
      return;
   }
   STRUCT_DECL(strbuf, sb);
   if (mi_copy_state(q, mp, &data) == -1) {
      return;
   }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag, (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
### STRUCT_FADDR(9F)

#### NAME

| STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP, STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32-bit application data access macros |

#### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);
void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);
STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);
STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);
<typeof field> *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);
size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);
```

#### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

#### PARAMETERS

The macros take the following parameters:

- **structname**
  
  The structure name (as would appear after the C keyword “struct”) of the native form.

- **umodel**
  
  A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

- **handle**
  
  The variable name used to refer to a particular instance of a structure which is handled by these macros.

- **field**
  
  The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first "." or ">>".
### DESCRIPTION

The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

#### STRUCT_DECL(structname, handle)

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_INIT(handle, model_t umodel)
```

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

#### STRUCT_HANDLE(structname, handle)

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct”.

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`.

---

1516  man pages section 9: DDI and DKI Kernel Functions • Last Revised 23 Feb 1998
### Operation Macros

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t STRUCT_SIZE(handle)</td>
<td>Returns size of the structure referred to by handle. It will return the size depending upon the data model associated with handle. If the data model stored by STRUCT_INIT() or STRUCT_SET_HANDLE() was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.</td>
</tr>
<tr>
<td>size_t STRUCT_FGET(handle, field)</td>
<td>Returns the contents of field in the structure described by handle according to the data model associated with handle.</td>
</tr>
<tr>
<td>size_t STRUCT_FGETP(handle, field)</td>
<td>This is the same as STRUCT_FGET() except that the field in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.</td>
</tr>
<tr>
<td>size_t STRUCT_FSET(handle, field, val)</td>
<td>Assigns val to the (non pointer) in the structure described by handle. It should not be used within any other expression, but rather only as a statement.</td>
</tr>
<tr>
<td>size_t STRUCT_FSETP(handle, field, val)</td>
<td>Returns a pointer to the in the structure described by handle.</td>
</tr>
<tr>
<td>struct structname *STRUCT_BUF(handle)</td>
<td>Returns a pointer to the native mode instance of the structure described by handle.</td>
</tr>
<tr>
<td>size_t SIZEOF_STRUCT(structname, umodel)</td>
<td>Returns size of structname based on umodel.</td>
</tr>
<tr>
<td>size_t SIZEOF_PTR(umodel)</td>
<td>Returns the size of a pointer based on umodel.</td>
</tr>
</tbody>
</table>

### Macros Not Using Handles

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE 1 Copying a Structure</td>
<td>The following example uses an ioctl(9E) on a regular character device that copies a data structure that looks like this into the kernel:</td>
</tr>
</tbody>
</table>

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

| EXAMPLE 2 Defining a Structure | This data structure definition describes what the ioctl(9E) would look like in a 32-bit application using fixed width types. |

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
               STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
   (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
   mi_copy_done(q, msgp, EINVAL);
   return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP, STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32–bit application data access macros</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL (structname, handle);

STRUCT_HANDLE (structname, handle);

void STRUCT_INIT (handle, model_t umodel);

void STRUCT_SET_HANDLE (handle, model_t umodel, void *addr);

STRUCT_FGET (handle, field);

STRUCT_FGETP (handle, field);

STRUCT_FSET (handle, field, val);

STRUCT_FSETP (handle, field, val);

<typedef field> *STRUCT_FADDR (handle, field);

struct structname *STRUCT_BUF (handle);

size_t SIZEOF_STRUCT (structname, umodel);

size_t SIZEOF_PTR (umodel);

size_t STRUCT_SIZE (handle);

<table>
<thead>
<tr>
<th>INTERFACE LEVEL</th>
<th>Solaris DDI specific (Solaris DDI).</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETERS</td>
<td>The macros take the following parameters:</td>
</tr>
</tbody>
</table>

**structname**  
The structure name (as would appear after the C keyword “struct”) of the native form.

**umodel**  
A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

**handle**  
The variable name used to refer to a particular instance of a structure which is handled by these macros.

**field**  
The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “.” or “->”.

---

1520  man pages section 9: DDI and DKI Kernel Functions • Last Revised 23 Feb 1998
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros $\text{STRUCT\_DECL()}$ and $\text{STRUCT\_HANDLE()}$ declare structure handles on the stack, whereas the macros $\text{STRUCT\_INIT()}$ and $\text{STRUCT\_SET\_HANDLE()}$ initialize the structure handles to point to an instance of the native form structure.

The macros $\text{STRUCT\_HANDLE()}$ and $\text{STRUCT\_SET\_HANDLE()}$ are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros $\text{STRUCT\_DECL()}$ and $\text{STRUCT\_INIT()}$, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by $\text{STRUCT\_DECL()}$, that is, any standard character or block device driver $\text{ioctl(9E)}$ routine that needs to copy in data from a user-mode program.

$\text{STRUCT\_DECL(\text{structname}, \text{handle})}$
Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. $\text{handle}$ is a variable name and is declared as a variable by this macro.

$\text{void STRUCT\_INIT(\text{handle}, \text{model}_t \text{umodel})}$
Initializes $\text{handle}$ to point to the instance allocated by $\text{STRUCT\_DECL()}$, it also sets data model for $\text{handle}$ to $\text{umodel}$, and must be called before any access is made through the macros that operate on these structures. When used in an $\text{ioctl(9E)}$ routine $\text{umodel}$ is the flag parameter; in $\text{adevmap(9E)}$ routine $\text{umodel}$ is the model parameter and in a $\text{mmap(9E)}$ routine, is the return value of $\text{ddi\_mmap\_get\_model(9F)}$. This macro is intended for handles created with $\text{STRUCT\_DECL()}$ only.

$\text{STRUCT\_HANDLE(\text{structname}, \text{handle})}$
Declares a “structure handle” $\text{handle}$ but unlike $\text{STRUCT\_DECL()}$ does not allocate an instance of “struct”.

$\text{void STRUCT\_SET\_HANDLE(\text{handle}, \text{model}_t \text{umodel}, \text{void}^{*}\text{addr})}$
Initializes to point to the native form instance at $\text{addr}$, it also sets the data model for $\text{handle}$ to $\text{umodel}$. This is intended for handles created with $\text{STRUCT\_HANDLE()}$. Fields cannot be referenced via the $\text{handle}$ until this macro has been invoked. Typically, $\text{addr}$ is the address of the native form structure containing the user-mode programs data. When used in an $\text{ioctl(9E)}$ $\text{umodel}$ is the flag parameter, in a $\text{devmap(9E)}$ routine is the model parameter and in a $\text{mmap(9E)}$ routine, $\text{umodel}$ is the return value of $\text{ddi\_mmap\_get\_model(9F)}$. 

$\text{STRUCT\_FGET(9F)}$
STRUCT_FGET(9F)

Operation Macros

size_t STRUCT_SIZE(handle)
Returns size of the structure referred to by handle. It will return the size depending upon the data model associated with handle. If the data model stored by STRUCT_INIT() or STRUCT_SET_HANDLE() was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

STRUCT_FGET(handle, field)
Returns the contents of field in the structure described by handle according to the data model associated with handle.

STRUCT_FGETP(handle, field)
This is the same as STRUCT_FGET() except that the field in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

STRUCT_FSET(handle, field, val)
Assigns val to the (non pointer) in the structure described by handle. It should not be used within any other expression, but rather only as a statement.

STRUCT_FSETP(handle, field, val)
Returns a pointer to the in the structure described by handle.

struct structname *STRUCT_BUF(handle)
Returns a pointer to the native mode instance of the structure described by handle.

size_t SIZEOF_STRUCT(structname, umodel)
Returns size of structname based on umodel.

size_t SIZEOF_PTR(umodel)
Returns the size of a pointer based on umodel.

Macros Not Using Handles

EXAMPLES

EXAMPLE 1 Copying a Structure
The following example uses an ioctl(9E) on a regular character device that copies a data structure that looks like this into the kernel:

struct opdata {
    size_t size;
    uint_t flag;
};

EXAMPLE 2 Defining a Structure
This data structure definition describes what the ioctl(9E) would look like in a 32-bit application using fixed width types.

#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
EXAMPLE 3 Using `STRUCT_DECL()` and `STRUCT_INIT()`

Note: This example uses the `STRUCT_DECL()` and `STRUCT_INIT()` macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
               STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles `ioctl(9E)` data (M_IOCCDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()`

The next example illustrates the use of the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
               (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
**STRUCT_FGETP(9F)**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP, STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32-bit application data access macros</td>
<td></td>
</tr>
</tbody>
</table>

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);

void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);

STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);

STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);

<typeof field> *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);

size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);
```

**INTERFACE LEVEL PARAMETERS**

Solaris DDI specific (Solaris DDI).

The macros take the following parameters:

- **structname**
  The structure name (as would appear after the C keyword “struct”) of the native form.

- **umodel**
  A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

- **handle**
  The variable name used to refer to a particular instance of a structure which is handled by these macros.

- **field**
  The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “.” or “->”.

Kernel Functions for Drivers  1525
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

`STRUCT_DECL(structname, handle)`
Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

`void STRUCT_INIT(handle, model_t umodel)`
Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

`STRUCT_HANDLE(structname, handle)`
Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct”.

`void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)`
Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked.

Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`.
STRUCT_FGETP(9F)

**Operation Macros**

- `size_t STRUCT_SIZE(handle)`
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- `STRUCT_FGET(handle, field)`
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- `STRUCT_FGETP(handle, field)`
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- `STRUCT_FSET(handle, field, val)`
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- `STRUCT_FSETP(handle, field, val)`
  Returns a pointer to the field in the structure described by `handle`.

- `struct structname *STRUCT_BUF(handle)`
  Returns a pointer to the native mode instance of the structure described by `handle`.

- `size_t SIZEOF_STRUCT(structname, umodel)`
  Returns size of `structname` based on `umodel`.

- `size_t SIZEOF_PTR(umodel)`
  Returns the size of a pointer based on `umodel`.

**Macros Not Using Handles**

- `size_t SIZEOF_STRUCT(structname, umodel)`
  Returns size of `structname` based on `umodel`.

- `size_t SIZEOF_PTR(umodel)`
  Returns the size of a pointer based on `umodel`.

**EXAMPLES**

**EXAMPLE 1 Copying a Structure**

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2 Defining a Structure**

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(__MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
}
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
    (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
## STRUCT_FSET(9F)

### NAME

STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP,
STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32–bit
application data access macros

### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL (structname, handle);
STRUCT_HANDLE (structname, handle);
void STRUCT_INIT (handle, model_t umodel);
void STRUCT_SET_HANDLE (handle, model_t umodel, void *addr);
STRUCT_FGET (handle, field);
STRUCT_FGETP (handle, field);
STRUCT_FSET (handle, field, val);
STRUCT_FSETP (handle, field, val);
<typeof field> *STRUCT_FADDR (handle, field);
struct structname *STRUCT_BUF (handle);
size_t SIZEOF_STRUCT (structname, umodel);
size_t SIZEOF_PTR (umodel);
size_t STRUCT_SIZE (handle);
```

### INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

The macros take the following parameters:

- `structname`: The structure name (as would appear after the C keyword “struct”) of the native form.
- `umodel`: A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.
- `handle`: The variable name used to refer to a particular instance of a structure which is handled by these macros.
- `field`: The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then `field` can be whether complex expression could occur after the first “.” or “->”.
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

**STRUCT_DECL(structname, handle)**

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_INIT(handle, model_t umodel)
```

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `adevmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

**STRUCT_HANDLE(structname, handle)**

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct”.

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 
STRUCT_FSET(9F)

**Operation Macros**

- `size_t STRUCT_SIZE(handle)`
  - Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- `STRUCT_FGET(handle, field)`
  - Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- `STRUCT_FGETP(handle, field)`
  - This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast `caddr32_t` to a `(void *)` when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- `STRUCT_FSET(handle, field, val)`
  - Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- `STRUCT_FSETP(handle, field, val)`
  - Returns a pointer to the in the structure described by `handle`.

**Macros Not Using Handles**

- `size_t SIZEOF_STRUCT(structname, umodel)`
  - Returns size of `structname` based on `umodel`.

- `size_t SIZEOF_PTR(umodel)`
  - Returns the size of a pointer based on `umodel`.

**EXAMPLES**

**EXAMPLE 1** Copying a Structure

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2** Defining a Structure

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using `STRUCT_DECL()` and `STRUCT_INIT()`

Note: This example uses the `STRUCT_DECL()` and `STRUCT_INIT()` macros to declare and initialize the structure handle.

```c
int xioctl(dev_t dev, int cmd, intptr_t arg, int mode,
   cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
      return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
       STRUCT_BUF(op), STRUCT_SIZE(op)))
      return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
      return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles `ioctl(9E)` data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()`

The next example illustrates the use of the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};
static void wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);
    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` (Continued)

```c
    STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag, (void *)data->b_rptr);
    if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
        mi_copy_done(q, msgp, EINVAL);
        return;
    }
```

SEE ALSO `devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)`

Writing Device Drivers

STREAMS Programming Guide
The macros take the following parameters:

**structname**

The structure name (as would appear after the C keyword “struct”) of the native form.

**umodel**

A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

**handle**

The variable name used to refer to a particular instance of a structure which is handled by these macros.

**field**

The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “.” or “->”.

---

**STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP, STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE** – 32-bit application data access macros

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL (structname, handle);
STRUCT_HANDLE (structname, handle);
void STRUCT_INIT (handle, model_t umodel);
void STRUCT_SET_HANDLE (handle, model_t umodel, void *addr);
STRUCT_FGET (handle, field);
STRUCT_FGETP (handle, field);
STRUCT_FSET (handle, field, val);
STRUCT_FSETP (handle, field, val);
<typeof field> *STRUCT_FADDR (handle, field);
struct structname *STRUCT_BUF (handle);
size_t SIZEOF_STRUCT (structname, umodel);
size_t SIZEOF_PTR (umodel);
size_t STRUCT_SIZE (handle);
```

Solaris DDI specific (Solaris DDI).
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

**STRUCT_DECL(structname, handle)**

Declares a "structure handle" for a "struct" and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

```c
void STRUCT_INIT(handle, model_t umodel)
```

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

**STRUCT_HANDLE(structname, handle)**

Declares a "structure handle" `handle` but unlike `STRUCT_DECL()` does not allocate an instance of "struct".

```c
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)
```

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`.
Operation Macros

- **size_t STRUCT_SIZE(handle)**
  Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

- **STRUCT_FGET(handle, field)**
  Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

- **STRUCT_FGETP(handle, field)**
  This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

- **STRUCT_FSET(handle, field, val)**
  Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

- **STRUCT_FSETP(handle, field, val)**
  Returns a pointer to the in the structure described by `handle`.

- **struct structname *STRUCT_BUF(handle)**
  Returns a pointer to the native mode instance of the structure described by `handle`.

- **size_t SIZEOF_STRUCT(structname, umodel)**
  Returns size of `structname` based on `umodel`.

- **size_t SIZEOF_PTR(umodel)**
  Returns the size of a pointer based on `umodel`.

Macros Not Using Handles

- **EXAMPLE 1 Copying a Structure**
  The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

  ```c
  struct opdata {
    size_t size;
    uint_t flag;
  };
  ```

- **EXAMPLE 2 Defining a Structure**
  This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

  ```c
  #if defined(_MULTI_DATAMODEL)
  struct opdata32 {
    size32_t size;
    uint32_t flag;
  };
  #endif
  ```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);

    if (cmd != OPONE)
        return (ENOTTY);

    STRUCT_INIT(op, mode);

    if (copyin((void *)data,
            STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);

    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);

    xxdowork(device_state, STRUCT_FGET(op, size));

    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copy in the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
    (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
STRUCT_HANDLE(9F)

NAME
STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR, STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP, STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32–bit application data access macros

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

structname

void

<typeof field> *

size_t

<typeof field> *

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
PARAMETERS
The macros take the following parameters:

structname
The structure name (as would appear after the C keyword “struct”) of the native form.

umodel
A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

handle
The variable name used to refer to a particular instance of a structure which is handled by these macros.

field
The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “.” or “->”.

1540 man pages section 9: DDI and DKI Kernel Functions • Last Revised 23 Feb 1998
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

```
# macros

# DECLARATION

DECLARE(structname, handle)

Declarer a "structure handle" for a "struct" and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

INITIALIZE

INITIALIZE(handle, model_t umodel)

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

```

```

# Macros

# Declaration and Initialization Macros

# STRUCT_DECL

DECLARE(structname, handle)

Declares a "structure handle" for a "struct" and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

# STRUCT_INIT

INITIALIZE(handle, model_t umodel)

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

# STRUCT_HANDLE

DECLARE(structname, handle)

Declares a "structure handle" `handle` but unlike `STRUCT_DECL()` does not allocate an instance of "struct ".

# STRUCT_SET_HANDLE

INITIALIZE(handle, model_t umodel, void *addr)

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`. 

```
STRUCT_HANDLE(9F)

**Operation Macros**

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t STRUCT_SIZE(handle)</td>
<td>Returns size of the structure referred to by <em>handle</em>. It will return the size depending upon the data model associated with <em>handle</em>. If the data model stored by STRUCT_INIT() or STRUCT_SET_HANDLE() was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.</td>
</tr>
<tr>
<td>STRUCT_FGET(handle, field)</td>
<td>Returns the contents of <em>field</em> in the structure described by <em>handle</em> according to the data model associated with <em>handle</em>.</td>
</tr>
<tr>
<td>STRUCT_FGETP(handle, field)</td>
<td>This is the same as STRUCT_FGET() except that the <em>field</em> in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.</td>
</tr>
<tr>
<td>STRUCT_FSET(handle, field, val)</td>
<td>Assigns <em>val</em> to the (non-pointer) in the structure described by <em>handle</em>. It should not be used within any other expression, but rather only as a statement.</td>
</tr>
<tr>
<td>STRUCT_FSETP(handle, field, val)</td>
<td>Returns a pointer to the in the structure described by <em>handle</em>.</td>
</tr>
</tbody>
</table>

**Macros Not Using Handles**

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t SIZEOF_STRUCT(structname, umodel)</td>
<td>Returns size of <em>structname</em> based on <em>umodel</em>.</td>
</tr>
<tr>
<td>size_t SIZEOF_PTR(umodel)</td>
<td>Returns the size of a pointer based on <em>umodel</em>.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

**EXAMPLE 1** Copying a Structure

The following example uses an ioctl(9E) on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2** Defining a Structure

This data structure definition describes what the ioctl(9E) would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```

1542  man pages section 9: DDI and DKI Kernel Functions  •  Last Revised 23 Feb 1998
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};
static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
Example 4 Using `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` (Continued)

```
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,
               (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof(ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

See Also

devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
#include <sys/ddi.h>
#include <sys/sunddi.h>

**INTERFACE**

_SUN_KERNEL_FUNCTION(SUN_STRUCTFADDR)

**LEVEL**

Solaris DDI specific (Solaris DDI).

**PARAMETERS**

The macros take the following parameters:

- `structname`
  - The structure name (as would appear after the C keyword “struct”) of the native form.

- `umodel`
  - A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

- `handle`
  - The variable name used to refer to a particular instance of a structure which is handled by these macros.

- `field`
  - The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first "." or ">>".
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, iwcts within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

`STRUCT_DECL(structname, handle)`

Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

`void STRUCT_INIT(handle, model_t umodel)`

Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine `umodel` is the flag parameter; in `devmap(9E)` routine `umodel` is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

`STRUCT_HANDLE(structname, handle)`

Declares a “structure handle” `handle` but unlike `STRUCT_DECL()` does not allocate an instance of “struct”.

`void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)`

Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` `umodel` is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, `umodel` is the return value of `ddi_mmap_get_model(9F)`.
Operation Macros

```c
size_t STRUCT_SIZE(handle)
Returns size of the structure referred to by handle. It will return the size depending
upon the data model associated with handle. If the data model stored by
STRUCT_INIT() or STRUCT_SET_HANDLE() was DATAMODEL_ILP32, it will
return the size of the ILP32 form, else it will return the size of the native form.

STRUCT_FGET(handle, field)
Returns the contents of field in the structure described by handle according to the
data model associated with handle.

STRUCT_FGETP(handle, field)
This is the same as STRUCT_FGET() except that the field in question is a pointer of
some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure
to use this macro for a pointer will lead to compiler warnings or failures.

STRUCT_FSET(handle, field, val)
Assigns val to the (non pointer) in the structure described by handle. It should not
be used within any other expression, but rather only as a statement.

STRUCT_FSETP(handle, field, val)
Returns a pointer to the in the structure described by handle.
```

```c
struct structname *STRUCT_BUF(handle)
Returns a pointer to the native mode instance of the structure described by handle.
```

```c
size_t SIZEOF_STRUCT(structname, umodel)
Returns size of structname based on umodel.
```

```c
size_t SIZEOF_PTR(umodel)
Returns the size of a pointer based on umodel.
```

Macros Not Using Handles

```c
EXAMPLES
EXAMPLE 1 Copying a Structure
The following example uses an ioctl(9E) on a regular character device that copies a
data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

EXAMPLE 2 Defining a Structure
This data structure definition describes what the ioctl(9E) would look like in a 32-bit
application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE
        || STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wpput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using \texttt{STRUCT\_HANDLE()} and \texttt{STRUCT\_SET\_HANDLE()} (Continued)

\begin{verbatim}
STRUCT\_SET\_HANDLE(sb,((struct iocblk *)msgp->b_rptr)->ioc_flag, 
(void *)data->b_rptr);
if (STRUCT\_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
  mi_copy_done(q, msgp, EINVAL);
  return;
}
\end{verbatim}

SEE ALSO \texttt{devmap(9E), ioctl(9E), mmap(9E), ddi\_mmap\_get\_model(9F)}

\textit{Writing Device Drivers}

\textit{STREAMS Programming Guide}
STRUCT_SET_HANDLE(9F)

NAME
STRUCT_DECL, SIZEOF_PTR, SIZEOF_STRUCT, STRUCT_BUF, STRUCT_FADDR,
STRUCT_FGET, STRUCT_FGETP, STRUCT_FSET, STRUCT_FSETP,
STRUCT_HANDLE, STRUCT_INIT, STRUCT_SIZE, STRUCT_SET_HANDLE – 32–bit
application data access macros

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);
void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);
STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);
STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);
<typeof field> *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);
size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

PARAMETERS
The macros take the following parameters:

structname
The structure name (as would appear after the C keyword “struct”) of the native
form.

umodel
A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64
model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the
flag parameter; in a devmap(9E), they will be present in the model parameter
mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the
current thread.

handle
The variable name used to refer to a particular instance of a structure which is
handled by these macros.

field
The field name within the structure contain substructures. If the structures contain
substructures, unions, or arrays, then field can be whether complex expression could
occur after the first “.” or “->”.
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctls within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl` routine that needs to copy in data from a user-mode program.

`STRUCT_DECL(structname, handle)`
Declares a "structure handle" for a "struct" and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. `handle` is a variable name and is declared as a variable by this macro.

`void STRUCT_INIT(handle, model_t umodel)`
Initializes `handle` to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for `handle` to `umodel`, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl` routine `umodel` is the flag parameter; in `devmap` routine `umodel` is the model parameter and in a `mmap` routine, is the return value of `ddi_mmap_get_model`. This macro is intended for handles created with `STRUCT_DECL()` only.

`STRUCT_HANDLE(structname, handle)`
Declares a "structure handle" `handle` but unlike `STRUCT_DECL()` does not allocate an instance of "struct ".

`void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)`
Initializes to point to the native form instance at `addr`, it also sets the data model for `handle` to `umodel`. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the `handle` until this macro has been invoked. Typically, `addr` is the address of the native form structure containing the user-mode programs data. When used in an `ioctl` routine `umodel` is the flag parameter, in a `devmap` routine is the model parameter and in a `mmap` routine, `umodel` is the return value of `ddi_mmap_get_model`.

DESCRIPTION

Declaration and Initialization Macros

Kernel Functions for Drivers  1551
STRUCT_SET_HANDLE(9F)

Operation Macros

size_t STRUCT_SIZE(handle)
Returns size of the structure referred to by *handle*. It will return the size depending upon the data model associated with *handle*. If the data model stored by STRUCT_INIT() or STRUCT_SET_HANDLE() was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

STRUCT_FGET(handle, field)
Returns the contents of *field* in the structure described by *handle* according to the data model associated with *handle*.

STRUCT_FGETP(handle, field)
This is the same as STRUCT_FGET() except that the *field* in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

STRUCT_FSET(handle, field, val)
Assigns *val* to the (non pointer) in the structure described by *handle*. It should not be used within any other expression, but rather only as a statement.

STRUCT_FSETP(handle, field, val)
Returns a pointer to the in the structure described by *handle*.

struct structname *STRUCT_BUF(handle)
Returns a pointer to the native mode instance of the structure described by *handle*.

size_t SIZEOF_STRUCT(structname, umodel)
Returns size of *structname* based on *umodel*.

size_t SIZEOF_PTR(umodel)
Returns the size of a pointer based on *umodel*.

Macros Not Using Handles

EXAMPLES

EXAMPLE 1 Copying a Structure

The following example uses an ioctl(9E) on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

EXAMPLE 2 Defining a Structure

This data structure definition describes what the ioctl(9E) would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
            cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);
    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE() (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag, 
    (void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO
devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
### SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

// Structural data access macros

structname *STRUCT_DECL(structname, handle);

void STRUCT_HANDLE(structname, handle);

void STRUCT_INIT(handle, model_t umodel);

void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);

field *STRUCT_FGET(handle, field);

field *STRUCT_FGETP(handle, field);

field *STRUCT_FSET(handle, field, val);

field *STRUCT_FSETP(handle, field, val);

<typeof field> *STRUCT_FADDR(handle, field);

struct structname *STRUCT_BUF(handle);

size_t SIZEOF_STRUCT(structname, umodel);

size_t SIZEOF_PTR(umodel);

size_t STRUCT_SIZE(handle);
```

### INTERFACE LEVEL PARAMETERS

Solaris DDI specific (Solaris DDI).

The macros take the following parameters:

- **structname**
  - The structure name (as would appear after the C keyword “struct”) of the native form.

- **umodel**
  - A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an ioctl(9E), these bits will be present in the flag parameter; in a devmap(9E), they will be present in the model parameter mmap(9E) and can call ddi_mmap_get_model(9F) to get the data model of the current thread.

- **handle**
  - The variable name used to refer to a particular instance of a structure which is handled by these macros.

- **field**
  - The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then field can be whether complex expression could occur after the first “.” or “->”.

---

**Kernel Functions for Drivers 1555**
The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, ioctl within a STREAMS module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

**Declaration and Initialization Macros**

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>STRUCT_DECL()</code></td>
<td>Declares a “structure handle” for a “struct” and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. handle is a variable name and is declared as a variable by this macro.</td>
</tr>
<tr>
<td><code>STRUCT_INIT()</code></td>
<td>Initializes handle to point to the instance allocated by <code>STRUCT_DECL()</code>, it also sets data model for handle to umodel, and must be called before any access is made through the macros that operate on these structures. When used in an <code>ioctl(9E)</code> routine umodel is the flag parameter; in <code>devmap(9E)</code> routine umodel is the model parameter and in a <code>mmap(9E)</code> routine, is the return value of <code>ddi_mmap_get_model(9F)</code>. This macro is intended for handles created with <code>STRUCT_DECL()</code> only.</td>
</tr>
<tr>
<td><code>STRUCT_HANDLE()</code></td>
<td>Declares a “structure handle” handle but unlike <code>STRUCT_DECL()</code> does not allocate an instance of “struct”.</td>
</tr>
<tr>
<td><code>STRUCT_SET_HANDLE()</code></td>
<td>Initializes to point to the native form instance at addr, it also sets the data model for handle to umodel. This is intended for handles created with <code>STRUCT_HANDLE()</code>. Fields cannot be referenced via the handle until this macro has been invoked. Typically, addr is the address of the native form structure containing the user-mode programs data. When used in an <code>ioctl(9E)</code> umodel is the flag parameter, in a <code>devmap(9E)</code> routine is the model parameter and in a <code>mmap(9E)</code> routine, umodel is the return value of <code>ddi_mmap_get_model(9F)</code>.</td>
</tr>
</tbody>
</table>
Operation Macros

**size_t STRUCT_SIZE(handle)**
Returns size of the structure referred to by `handle`. It will return the size depending upon the data model associated with `handle`. If the data model stored by `STRUCT_INIT()` or `STRUCT_SET_HANDLE()` was DATAMODEL_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

**STRUCT_FGET(handle, field)**
Returns the contents of `field` in the structure described by `handle` according to the data model associated with `handle`.

**STRUCT_FGETP(handle, field)**
This is the same as `STRUCT_FGET()` except that the `field` in question is a pointer of some kind. This macro will cast caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

**STRUCT_FSET(handle, field, val)**
Assigns `val` to the (non pointer) in the structure described by `handle`. It should not be used within any other expression, but rather only as a statement.

**STRUCT_FSETP(handle, field, val)**
Returns a pointer to the in the structure described by `handle`.

**struct structname *STRUCT_BUF(handle)**
Returns a pointer to the native mode instance of the structure described by `handle`.

Macros Not Using Handles

**size_t SIZEOF_STRUCT(structname, umodel)**
Returns size of `structname` based on `umodel`.

**size_t SIZEOF_PTR(umodel)**
Returns the size of a pointer based on `umodel`.

EXAMPLES

**EXAMPLE 1 Copying a Structure**

The following example uses an `ioctl(9E)` on a regular character device that copies a data structure that looks like this into the kernel:

```c
struct opdata {
    size_t size;
    uint_t flag;
};
```

**EXAMPLE 2 Defining a Structure**

This data structure definition describes what the `ioctl(9E)` would look like in a 32-bit application using fixed width types.

```c
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t size;
    uint32_t flag;
};
#endif
```
EXAMPLE 3 Using STRUCT_DECL() and STRUCT_INIT()

Note: This example uses the STRUCT_DECL() and STRUCT_INIT() macros to declare and initialize the structure handle.

```c
int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);
    if (cmd != OPONE)
        return (ENOTTY);
    STRUCT_INIT(op, mode);
    if (copyin((void *)data,
                STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);
    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()

The next example illustrates the use of the STRUCT_HANDLE() and STRUCT_SET_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT_HANDLE() and STRUCT_SET_HANDLE() as follows:

```c
struct strbuf {
    int maxlen; /* no. of bytes in buffer */
    int len; /* no. of bytes returned */
    caddr_t buf; /* pointer to data */
};

static void
wput_ioctldata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
```
EXAMPLE 4 Using STRUCT_HANDLE() and STRUCT_SET_HANDLE()  (Continued)

```c
STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag, 
(void *)data->b_rptr);
if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
    mi_copy_done(q, msgp, EINVAL);
    return;
}
```

SEE ALSO  devmap(9E), ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers

STREAMS Programming Guide
swab(9F)

NAME  swab – swap bytes in 16-bit halfwords

SYNOPSIS  
#include <sys/sunddi.h>

void swab(void *src, void *dst, size_t nbytes);

INTERFACE LEVEL PARAMETERS  Architecture independent level 1 (DDI/DKI).

src  A pointer to the buffer containing the bytes to be swapped.

dst  A pointer to the destination buffer where the swapped bytes will be written. If dst is the same as src the buffer will be swapped in place.

nbytes  Number of bytes to be swapped, rounded down to the nearest half-word.

DESCRIPTION  swab() copies the bytes in the buffer pointed to by src to the buffer pointer to by dst, swapping the order of adjacent bytes in half-word pairs as the copy proceeds. A total of nbytes bytes are copied, rounded down to the nearest half-word.

CONTEXT  swab() can be called from user or interrupt context.

SEE ALSO  Writing Device Drivers

NOTES  Since swab() operates byte-by-byte, it can be used on non-aligned buffers.
testb(9F)

NAME
testb – check for an available buffer

SYNOPSIS
#include <sys/stream.h>

int testb(size_t size, uint_t pri);

INTERFACE LEVEL
PARAMETERS
Architecture independent level 1 (DDI/DKI).

DESCRIPTION
testb() checks to see if an allocb(9F) call is likely to succeed if a buffer of size
bytes at priority pri is requested. Even if testb() returns successfully, the call to
allocb(9F) can fail. The pri argument is no longer used, but is retained for
compatibility.

RETURN VALUES
Returns 1 if a buffer of the requested size is available, and 0 if one is not.

CONTEXT
testb() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 testb() example

In a service routine, if copymsg(9F) fails (line 6), the message is put back on the queue
(line 7) and a routine, tryagain, is scheduled to be run in one tenth of a second. Then
the service routine returns.

When the timeout(9F) function runs, if there is no message on the front of the queue,
it just returns. Otherwise, for each message block in the first message, check to see if
an allocation would succeed. If the number of message blocks equals the number we
can allocate, then enable the service procedure. Otherwise, reschedule tryagain to
run again in another tenth of a second. Note that tryagain is merely an
approximation. Its accounting may be faulty. Consider the case of a message
comprised of two 1024-byte message blocks. If there is only one free 1024-byte
message block and no free 2048-byte message blocks, then testb() will still succeed
twice. If no message blocks are freed of these sizes before the service procedure runs
again, then the copymsg(9F) will still fail. The reason testb() is used here is because
it is significantly faster than calling copymsg. We must minimize the amount of time
spent in a timeout() routine.

1  xxsrv(q)
2     queue_t *q;
3 {
4     mblk_t *mp;
5     mblk_t *nmp;
6     ...
7     if (!(nmp = copymsg(mp)) == NULL) {
8         putbq(q, mp);
9         timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
10        return;
11     }
12     ...

Kernel Functions for Drivers  1561
EXAMPLE 1 testb() example  (Continued)

    12  tryagain(q)
    13  queue_t *q;
    14  {
    15    register int can_alloc = 0;
    16    register int num_blks = 0;
    17    register mblk_t *mp;
    18    if (!q->q_first)
    19      return;
    20    for (mp = q->q_first; mp; mp = mp->b_cont) {
    21      num_blks++;
    22      can_alloc += testb((mp->b_datap->db_lim -
    23                           mp->b_datap->db_base), BPRI_MED);
    24    }
    25    if (num_blks == can_alloc)
    26      qenable(q);
    27    else
    28      timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
    29  }

SEE ALSO allocb(9F), bufcall(9F), copymsg(9F), timeout(9F)

Writing Device Drivers

STREAMS Programming Guide

NOTES The pri argument is provided for compatibility only. Its value is ignored.
timeout(9F)

NAME
timeout – execute a function after a specified length of time

SYNOPSIS
```
#include <sys/types.h>
#include <sys/conf.h>

timeout_id_t timeout(void (*func)(void *), void *arg, clock_t ticks);
```

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

PARAMETERS
- **func**: Kernel function to invoke when the time increment expires.
- **arg**: Argument to the function.
- **ticks**: Number of clock ticks to wait before the function is called. Use `drv_usectohz(9F)` to convert microseconds to clock ticks.

DESCRIPTION
The `timeout()` function schedules the specified function to be called after a specified time interval. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The function called by `timeout()` must adhere to the same restrictions as a driver soft interrupt handler.

The function called by `timeout()` is run in interrupt context and must not sleep or call other functions that might sleep.

The `delay(9F)` function calls `timeout()`. Because `timeout()` is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use `cv_timedwait(9F)` rather than `delay()`.

RETURN VALUES
`timeout()` returns an opaque non-zero timeout identifier that can be passed to `untimout(9F)` to cancel the request.

CONTEXT
`timeout()` can be called from user or interrupt context.

EXAMPLES
**EXAMPLE 1 Using timeout()**

In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

```c
static void
xxtimeout_handler(void *arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->lock);
    cv_signal(&xsp->cv);
    xsp->flags |= TIMED_OUT;
    mutex_exit(&xsp->lock);
    xsp->timeout_id = 0;
}
```

```c
static uint_t
xxintr(caddr_t arg)
{
```
EXAMPLE 1 Using timeout() (Continued)

```c
struct xxstate *xsp = (struct xxstate *)arg;
    .
    .
mutex_enter(&xsp->lock);
/* Service interrupt */
cv_signal(&xsp->cv);
mutex_exit(&xsp->lock);
if (xsp->timeout_id != 0) {
    (void) untimeout(xsp->timeout_id);
    xsp->timeout_id = 0;
}
return(DDI_INTRCLAIMED);
}
static void
xxcheckcond(struct xxstate *xsp)
{
    .
    .
xsp->timeout_id = timeout(xxtimeout_handler,
                            xsp, (5 * drv_usectohz(1000000)));
mutex_enter(&xsp->lock);
while (/* Waiting for interrupt or timeout*/)
    cv_wait(&xsp->cv, &xsp->lock);
if (xsp->flags & TIMED_OUT)
    cmn_err(CE_WARN, "Device not responding");
    .
    .
mutex_exit(&xsp->lock);
    .
}
```

SEE ALSO
bufcall(9F), cv_timedwait(9F), ddi_in_panic(9F), delay(9F),
drv_usectohz(9F), untimeout(9F)

Writing Device Drivers
uiomove(9F)

NAME
uiomove – copy kernel data using uio structure

SYNOPSIS
#include <sys/types.h>
#include <sys/uio.h>

int uiomove(caddr_t address, size_t nbytes, enum uio_rw rflag, uio_t *uio_p);

INTERFACE LEVEL PARAMETERS
Architecture independent level 1 (DDI/DKI).

PARAMETERS
address Source/destination kernel address of the copy.
nbytes Number of bytes to copy.
rflag Flag indicating read or write operation. Possible values are UIO_READ and UIO_WRITE.
uio_p Pointer to the uio structure for the copy.

DESCRIPTION
The uiomove() function copies nbytes of data to or from the space defined by the uio structure (described in uio(9S)) and the driver.

The uio_segflg member of the uio(9S) structure determines the type of space to or from which the transfer is being made. If it is set to UIO_SYSSPACE, the data transfer is between addresses in the kernel. If it is set to UIO_USERSPACE, the transfer is between a user program and kernel space.

rflag indicates the direction of the transfer. If UIO_READ is set, the data will be transferred from address to the buffer(s) described by uio_p. If UIO_WRITE is set, the data will be transferred from the buffer(s) described by uio_p to address.

In addition to moving the data, uiomove() adds the number of bytes moved to the iov_base member of the iovec(9S) structure, decreases the iov_len member, increases the uio_offset member of the uio(9S) structure, and decreases the uio_resid member.

This function automatically handles page faults. nbytes does not have to be word-aligned.

RETURN VALUES
uiomove() returns 0 upon success or EFAULT on failure.

CONTEXT
User context only, if uio_segflg is set to UIO_USERSPACE. User or interrupt context, if uio_segflg is set to UIO_SYSSPACE.

SEE ALSO
ureadc(9F), uwritec(9F), iovec(9S), uio(9S)

Writing Device Drivers

WARNINGS
If uio_segflg is set to UIO_SYSSPACE and address is selected from user space, the system may panic.
unbufcall(9F)

NAME  unbufcall – cancel a pending bufcall request

SYNOPSIS  #include <sys/stream.h>

   void unbufcall(bufcall_id_t id);

INTERFACE LEVEL  Architecture independent level 1 (DDI/DKI).

PARAMETERS

   id        Identifier returned from bufcall(9F) or esbbcall(9F).

DESCRIPTION  unbufcall cancels a pending bufcall() or esbbcall() request. The argument id
             is a non-zero identifier for the request to be cancelled. id is returned from the
             bufcall() or esbbcall() function used to issue the request. unbufcall() will
             not return until the pending callback is cancelled or has run. Because of this, locks
             acquired by the callback routine should not be held across the call to unbufcall() or
             deadlock may result.

RETURN VALUES  None.

CONTEXT  unbufcall() can be called from user or interrupt context.

SEE ALSO  bufcall(9F), esbbcall(9F)

Writing Device Drivers

STREAMS Programming Guide
**NAME**
freezestr, unfreezestr – freeze, thaw the state of a stream

**SYNOPSIS**
```
#include <sys/stream.h>
#include <sys/ddi.h>

void freezestr(queue_t *q);
void unfreezestr(queue_t *q);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `q` Pointer to the message queue to freeze/unfreeze.

**DESCRIPTION**
`freezestr()` freezes the state of the entire stream containing the queue pair `q`. A frozen stream blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the stream, and blocks any thread currently within the stream if it attempts to put messages onto or take messages off of any queue within the stream (with the sole exception of the caller). Threads blocked by this mechanism remain so until the stream is thawed by a call to `unfreezestr()`.

Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as `getq(9F)`, `putq(9F)`, `putbq(9F)`, etc.)

**CONTEXT**
These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.

**SEE ALSO**
- *Writing Device Drivers*
- *STREAMS Programming Guide*

**NOTES**
The `freezestr()` and `unfreezestr()` functions can have a serious impact on system performance. Their use should be very limited. In most cases, there is no need to use `freezestr()` and there are usually better ways to accomplish what you need to do than by freezing the stream.

Calling `freezestr()` to freeze a stream that is already frozen by the caller will result in a single-party deadlock.

The caller of `unfreezestr()` must be the thread who called `freezestr()`.

STREAMS utility functions such as `getq(9F)`, `putq(9F)`, `putbq(9F)`, and so forth, should not be called by the caller of `freezestr()` while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.
unlinkb(9F)

NAME unlinkb – remove a message block from the head of a message

SYNOPSIS

```c
#include <sys/stream.h>

mblk_t *unlinkb(mblk_t *mp);
```

INTERFACE LEVEL

PARAMETERS

mp Pointer to the message.

DESCRIPTION unlinkb() removes the first message block from the message pointed to by mp. A new message, minus the removed message block, is returned.

RETURN VALUES If successful, unlinkb() returns a pointer to the message with the first message block removed. If there is only one message block in the message, NULL is returned.

CONTEXT unlinkb() can be called from user or interrupt context.

EXAMPLES

**EXAMPLE 1** unlinkb() example

The routine expects to get passed an M_PROTO T_DATA_IND message. It will remove and free the M_PROTO header and return the remaining M_DATA portion of the message.

```c
1  mblk_t *
2  makedata(mp)
3   mblk_t *mp;
4 {
5   mblk_t *mp;
6   7   mp = unlinkb(mp);
8   freeb(mp);
9   return(mp);
10  }
```

SEE ALSO

linkb(9F)

Writing Device Drivers

STREAMS Programming Guide
untimoutout(9F)

NAME
untimoutout – cancel previous timeout function call

SYNOPSIS
#include <sys/types.h>
#include <sys/conf.h>

clock_t untimoutout(timeout_id_t id);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
id Opaque timeout ID from a previous timeout(9F) call.

DESCRIPTION
untimoutout() cancels a pending timeout(9F) request. untimoutout() will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to untimoutout() or a deadlock may result.

Since no mutex should be held across the call to untimoutout(), there is a race condition between the occurrence of an expected event and the execution of the timeout handler. In particular, it should be noted that no problems will result from calling untimoutout() for a timeout which is either running on another CPU, or has already completed. Drivers should be structured with the understanding that the arrival of both an interrupt and a timeout for that interrupt can occasionally occur, in either order.

RETURN VALUES
untimoutout() returns -1 if the id is not found. Otherwise, it returns an integer value greater than or equal to 0.

CONTEXT
untimoutout() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1

In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

    static void xxtimeout_handler(void *arg)
    {
        struct xxstate *xsp = (struct xxstate *)arg;
        mutex_enter(&xsp->lock);
        cv_signal(&xsp->cv);
        xsp->flags |= TIMED_OUT;
        mutex_exit(&xsp->lock);
        xsp->timeout_id = 0;
    }

    static uint_t xxintr(caddr_t arg)
    {
        struct xxstate *xsp = (struct xxstate *)arg;
        ...
        ...
        mutex_enter(&xsp->lock);
        /* Service interrupt */
EXAMPLE 1  (Continued)

    cv_signal(&xsp->cv);
    mutex_exit(&xsp->lock);
    if (xsp->timeout_id != 0) {
        (void) untimeout(xsp->timeout_id);
        xsp->timeout_id = 0;
    }
    return(DDI_INTR_CLAIMED);
}

static void
xxcheckcond(struct xxstate *xsp)
{
    .
    .
    xsp->timeout_id = timeout(xxtimeout_handler,
                xsp, (5 * drv_usectohz(1000000)));
    mutex_enter(&xsp->lock);
    while (/* Waiting for interrupt or timeout*/
            cv_wait(&xsp->cv, &xsp->lock);
    if (xsp->flags & TIMED_OUT)
        cmn_err(CE_WARN, "Device not responding");
    .
    .
    mutex_exit(&xsp->lock);
    .
}

SEE ALSO  open(9E), cv_signal(9F), cv_wait_sig(9F), delay(9F), timeout(9F)

Writing Device Drivers
<table>
<thead>
<tr>
<th>NAME</th>
<th>ureadc – add character to a uio structure</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/uio.h>  
#include <sys/types.h>  
int ureadc(int c, uio_t *uio_p); |
| INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI). |
| PARAMETERS |  
| c | The character added to the uio(9S) structure.  
uio_p | Pointer to the uio(9S) structure. |
| DESCRIPTION | ureadc() transfers the character c into the address space of the uio(9S) structure pointed to by uio_p, and updates the uio structure as for uiomove(9F). |
| RETURN VALUES | 0 is returned on success andEFAULT on failure. |
| CONTEXT | ureadc() can be called from user or interrupt context. |
| SEE ALSO | uiomove(9F), uwritec(9F), iovec(9S), uio(9S)  
Writing Device Drivers |
uwritec(9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>uwritec – remove a character from a uio structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/uio.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int uwritec(uio_t *uio_p);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI)</td>
</tr>
<tr>
<td>LEVEL</td>
<td>uio_p Pointer to the uio(9S) structure</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>DESCRIPTION uwritec() returns a character from</td>
</tr>
<tr>
<td></td>
<td>the uio structure pointed to by uio_p and</td>
</tr>
<tr>
<td></td>
<td>updates the uio structure. See uiomove(9F).</td>
</tr>
<tr>
<td></td>
<td>RETURN VALUES The next character for processing</td>
</tr>
<tr>
<td></td>
<td>is returned on success, and -1 is returned if</td>
</tr>
<tr>
<td></td>
<td>uio is empty or if there is an error.</td>
</tr>
<tr>
<td></td>
<td>CONTEXT uwritec() can be called from user or</td>
</tr>
<tr>
<td></td>
<td>interrupt context.</td>
</tr>
<tr>
<td></td>
<td>SEE ALSO uiomove(9F), ureadc(9F), iovec(9S),</td>
</tr>
<tr>
<td></td>
<td>uio(9S)</td>
</tr>
</tbody>
</table>

Writing Device Drivers
va_start(), va_arg(), va_copy(), va_end() – handle variable argument list

#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)
va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);

Solaris DDI specific (Solaris DDI).

va_start() is called to initialize pvar to the beginning of the variable argument list.
va_start() must be invoked before any access to the unnamed arguments. The
parameter name is the identifier of the rightmost parameter in the variable parameter
list in the function definition (the one just before the “,...”). If this parameter is
declared with the register storage class or with a function or array type, or with a
type that is not compatible with the type that results after application of the default
argument promotions, the behavior is undefined.

va_arg() expands to an expression that has the type and value of the next argument
in the call. The parameter pvar must be initialized by va_start(). Each invocation of
va_arg() modifies pvar so that the values of successive arguments are returned in
turn. The parameter type is the type name of the next argument to be returned. The
type name must be specified in such a way that the type of pointer to an object that
The `va_copy()` macro saves the state represented by the `va_list src` in the
`va_list dest`. The `va_list` passed as `dest` should not be initialized by a previous call
to `va_start()` before being reused as a parameter to `va_start()` or as the `dest` parameter of a subsequent call to
`va_copy()`. The behavior is undefined if any of these restrictions are not met.

The `va_end()` macro is used to clean up. It invalidates `pvar` for use (unless
`va_start()` is invoked again).

Multiple traversals, each bracketed by a call to `va_start()` and `va_end()`, are
possible.

**EXAMPLES**

**EXAMPLE 1 Creating a Variable Length Command**

The following example uses these routines to create a variable-length command. This
might be useful for a device that provides for a variable-length command set.
`ncmdbytes` is the number of bytes in the command. The new command is written to`cmdp`.

```c
static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
     * Write variable-length command to destination
     */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}
```

**SEE ALSO**

`vcmn_err(9F), vsprintf(9F)`

**NOTES**

It is up to the calling routine to specify in some manner how many arguments there
are, since it is not always possible to determine the number of arguments from the
stack frame.

Specifying a second argument of `char` or `short` to `va_arg` makes your code
non-portable, because arguments seen by the called function are not `char` or `short`.
C converts `char` and `short` arguments to `int` before passing them to a function.
va_arg, va_start, va_copy, va_end – handle variable argument list

```
#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)

va_arg(va_list pvar, type);

void va_copy(va_list dest, va_list src);

void va_end(va_list pvar);
```

**NAME**
va_arg, va_start, va_copy, va_end – handle variable argument list

**SYNOPSIS**
```
#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)

va_arg(va_list pvar, type);

void va_copy(va_list dest, va_list src);

void va_end(va_list pvar);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

va_start()  
`pvar`  
Pointer to variable argument list.

name  
Identifier of rightmost parameter in the function definition.

va_arg()  
`pvar`  
Pointer to variable argument list.

type  
Type name of the next argument to be returned.

va_copy()  
dest  
Destination variable argument list.

src  
Source variable argument list.

va_end()  
`pvar`  
Pointer to variable argument list.

**DESCRIPTION**
This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists but do not use the varargs() macros are inherently non-portable, as different machines use different argument-passing conventions. Routines that accept a variable argument list can use these macros to traverse the list.

va_list is the type defined for the variable used to traverse the list of arguments.

va_start() is called to initialize `pvar` to the beginning of the variable argument list. va_start() must be invoked before any access to the unnamed arguments. The parameter name is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the “,, . . . ”). If this parameter is declared with the register storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

va_arg() expands to an expression that has the type and value of the next argument in the call. The parameter `pvar` must be initialized by va_start(). Each invocation of va_arg() modifies `pvar` so that the values of successive arguments are returned in turn. The parameter type is the type name of the next argument to be returned. The type name must be specified in such a way that the type of pointer to an object that
has the specified type can be obtained by postfixing a * to type. If there is no actual
next argument, or if type is not compatible with the type of the actual next argument
(as promoted according to the default argument promotions), the behavior is
undefined.

The va_copy() macro saves the state represented by the va_list src in the
va_list dest. The va_list passed as dest should not be initialized by a previous call
to va_start() It then must be passed to va_end() before being reused as a
parameter to va_start() or as the dest parameter of a subsequent call to
va_copy(). The behavior is undefined if any of these restrictions are not met.

The va_end() macro is used to clean up. It invalidates pvar for use (unless
va_start() is invoked again).

Multiple traversals, each bracketed by a call to va_start() and va_end(), are
possible.

EXAMPLES

EXAMPLE 1 Creating a Variable Length Command

The following example uses these routines to create a variable length command. This
might be useful for a device that provides for a variable-length command set.
ncmdbytes is the number of bytes in the command. The new command is written to
cmdp.

static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
      * Write variable-length command to destination
      */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}

SEE ALSO

vcmn_err(9F), vsprintf(9F)

NOTES

It is up to the calling routine to specify in some manner how many arguments there
are, since it is not always possible to determine the number of arguments from the
stack frame.

Specifying a second argument of char or short to va_arg makes your code
non-portable, because arguments seen by the called function are not char or short.
C converts char and short arguments to int before passing them to a function.
va_arg, va_start, va_copy, va_end – handle variable argument list

#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)
va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);

NAME
va_arg, va_start, va_copy, va_end – handle variable argument list

SYNOPSIS
#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)
va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);

INTERFACE
Solaris DDI specific (Solaris DDI).
LEVEL
va_start()  pvar Pointer to variable argument list.
name Identifier of rightmost parameter in the function definition.
va_arg()  pvar Pointer to variable argument list.
type Type name of the next argument to be returned.
va_copy()  dest Destination variable argument list.
src Source variable argument list.
va_end()  pvar Pointer to variable argument list.

DESCRIPTION
This set of macros allows portable procedures that accept variable argument lists to be
written. Routines that have variable argument lists but do not use the varargs() macros are inherently non-portable, as different machines use different argument-passing conventions. Routines that accept a variable argument list can use these macros to traverse the list.

va_list is the type defined for the variable used to traverse the list of arguments.

va_start() is called to initialize pvar to the beginning of the variable argument list. va_start() must be invoked before any access to the unnamed arguments. The parameter name is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the “,...”). If this parameter is declared with the register storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

va_arg() expands to an expression that has the type and value of the next argument in the call. The parameter pvar must be initialized by va_start(). Each invocation of va_arg() modifies pvar so that the values of successive arguments are returned in turn. The parameter type is the type name of the next argument to be returned. The type name must be specified in such a way that the type of pointer to an object that...
va_end(9F)

has the specified type can be obtained by postfixing a * to type. If there is no actual
next argument, or if type is not compatible with the type of the actual next argument
(as promoted according to the default argument promotions), the behavior is
undefined.

The va_copy() macro saves the state represented by the va_list src in the
va_list dest. The va_list passed as dest should not be initialized by a previous call
to va_start(). It then must be passed to va_end() before being reused as a
parameter to va_start() or as the dest parameter of a subsequent call to
va_copy(). The behavior is undefined if any of these restrictions are not met.

The va_end() macro is used to clean up. It invalidates pvar for use (unless
va_start() is invoked again).

Multiple traversals, each bracketed by a call to va_start() and va_end(), are
possible.

EXAMPLES

EXAMPLE 1 Creating a Variable Length Command

The following example uses these routines to create a variable length command. This
might be useful for a device that provides for a variable-length command set.
ncmdbytes is the number of bytes in the command. The new command is written to
cmdp.

```c
static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
     * Write variable-length command to destination
     */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}
```

SEE ALSO

vcmn_err(9F), vsprintf(9F)

NOTES

It is up to the calling routine to specify in some manner how many arguments there
are, since it is not always possible to determine the number of arguments from the
stack frame.

Specifying a second argument of char or short to va_arg makes your code
non-portable, because arguments seen by the called function are not char or short.
C converts char and short arguments to int before passing them to a function.
va_start(9F)

NAME
va_arg, va_start, va_copy, va_end – handle variable argument list

SYNOPSIS
#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *)

va_arg(va_list pvar, type);

void va_copy(va_list dest, va_list src);

void va_end(va_list pvar);

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

va_start() pvar Pointer to variable argument list.
name Identifier of rightmost parameter in the function definition.

va_arg() pvar Pointer to variable argument list.

va_copy() dest Destination variable argument list.

va_end() pvar Pointer to variable argument list.

src Source variable argument list.

DESCRIPTION
This set of macros allows portable procedures that accept variable argument lists to be
written. Routines that have variable argument lists but do not use the varargs() macros are inherently non-portable, as different machines use different
argument-passing conventions. Routines that accept a variable argument list can use
these macros to traverse the list.

va_list is the type defined for the variable used to traverse the list of arguments.

va_start() is called to initialize pvar to the beginning of the variable argument list.
va_start() must be invoked before any access to the unnamed arguments. The
parameter name is the identifier of the rightmost parameter in the variable parameter
list in the function definition (the one just before the “,...”). If this parameter is
declared with the register storage class or with a function or array type, or with a
type that is not compatible with the type that results after application of the default
argument promotions, the behavior is undefined.

va_arg() expands to an expression that has the type and value of the next argument
in the call. The parameter pvar must be initialized by va_start(). Each invocation of
va_arg() modifies pvar so that the values of successive arguments are returned in
turn. The parameter type is the type name of the next argument to be returned. The
type name must be specified in such a way that the type of pointer to an object that
va_start(9F)

has the specified type can be obtained by postfixing a * to type. If there is no actual
next argument, or if type is not compatible with the type of the actual next argument
(as promoted according to the default argument promotions), the behavior is
undefined.

The va_copy() macro saves the state represented by the va_list src in the
va_list dest. The va_list passed as dest should not be initialized by a previous call
to va_start() It then must be passed to va_end() before being reused as a
parameter to va_start() or as the dest parameter of a subsequent call to
va_copy(). The behavior is undefined if any of these restrictions are not met.

The va_end() macro is used to clean up. It invalidates pvar for use (unless
va_start() is invoked again).

Multiple traversals, each bracketed by a call to va_start() and va_end(), are
possible.

EXAMPLES

EXAMPLE 1 Creating a Variable Length Command

The following example uses these routines to create a variable length command. This
might be useful for a device that provides for a variable-length command set.
ncmdbytes is the number of bytes in the command. The new command is written to
cmdp.

```
static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
    * Write variable-length command to destination
    */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}
```

SEE ALSO
vcmn_err(9F), vsprintf(9F)

NOTES

It is up to the calling routine to specify in some manner how many arguments there
are, since it is not always possible to determine the number of arguments from the
stack frame.

Specifying a second argument of char or short to va_arg makes your code
non-portable, because arguments seen by the called function are not char or short.
C converts char and short arguments to int before passing them to a function.
### NAME

cmn_err, vcmn_err – display an error message or panic the system

### SYNOPSIS

```c
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

void cmn_err(int level, char *format...);

#include <sys/varargs.h>

void vcmn_err(int level, char *format, va_list ap);
```

### INTERFACE LEVEL

cmn_err() - Architecture independent level 1 (DDI/DKI).

- **level**: A constant indicating the severity of the error condition.
- **format**: The message to be displayed.

vcmn_err() - Takes **level** and **format** as described for cmn_err(), but its third argument is different:

- **ap**: The variable argument list passed to the function.

cmn_err() - cmn_err() displays a specified message on the console. cmn_err() can also panic the system. When the system panics, it attempts to save recent changes to data, display a “panic message” on the console, attempt to write a core file, and halt system processing. See the CE_PANIC level below.

- **level**: Is a constant indicating the severity of the error condition. The four severity levels are:

  - **CE_CONT**: Used to continue another message or to display an informative message not associated with an error. Note that multiple CE_CONT messages without a newline may or may not appear on the system console or in the system log as a single line message. A single line message may be produced by constructing the message with `sprintf(9F)` or `vprintf(9F)` before calling cmn_err().

  - **CE_NOTE**: Used to display a message preceded with NOTICE. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a message saying that a sector on a disk needs to be accessed repeatedly before it can be accessed correctly might be noteworthy.

  - **CE_WARN**: Used to display a message preceded with WARNING. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.

  - **CE_PANIC**: Used to display a message preceded with "panic", and to panic the system. Drivers should specify this level only under the most severe conditions or when debugging a driver. A valid use of this.
level is when the system cannot continue to function. If the error is recoverable, or not essential to continued system operation, do not panic the system.

format is the message to be displayed. It is a character string which may contain plain characters and conversion specifications. By default, the message is sent both to the system console and to the system log.

Each conversion specification in format is introduced by the % character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional l (l) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (l) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

c The character value of the argument is displayed.

b The %b conversion specification allows bit values to be displayed meaningfully. Each %b takes an integer value and a format string from the argument list. The first character of the format string should be the output base encoded as a control character. This base is used to display the integer argument. The remaining groups of characters in the format string consist of a bit number (between 1 and 32, also encoded as a control character) and the next characters (up to the next control character or '\0') give the name of the bit field. The string corresponding to the bit fields set in the integer argument is displayed after the numerical value. See EXAMPLE section.

p The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void * when using the %p format specifier.

s The argument is taken to be a string (character pointer), and characters from the string are displayed until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.

% Copy a %; no argument is converted.
The first character in format affects where the message will be written:

! The message goes only to the system log.

^ The message goes only to the console.

? If level is also CE_CONT, the message is always sent to the system log, but is only written to the console when the system has been booted in verbose mode. See kernel(1M). If neither condition is met, the ‘?’ character has no effect and is simply ignored.

Refer to syslogd(1M) to determine where the system log is written.

cmn_err() appends a \n to each format, except when level is CE_CONT.

vcmn_err() is identical to cmn_err() except that its last argument, ap, is a pointer to a variable list of arguments. ap contains the list of arguments used by the conversion specifications in format. ap must be initialized by calling va_start(9F). va_end(9F) is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by va_start(9F) and va_end(9F), are possible.

RETURN VALUES None. However, if an unknown level is passed to cmn_err(), the following panic error message is displayed:

panic: unknown level in cmn_err (level=level, msg=format)

CONTEXT cmn_err() can be called from user, kernel, interrupt, or high-level interrupt context.

EXAMPLES EXAMPLE 1 Using cmn_err()

This first example shows how cmn_err() can record tracing and debugging information only in the system log (lines 17); display problems with a device only on the system console (line 23); or display problems with the device on both the system console and in the system log (line 28).

```c
1 struct reg {  
2   uchar_t data;  
3   uchar_t csr;  
4 };  
5  
6 struct xxstate {  
7   ...  
8   dev_info_t *dip;  
9   struct reg *regp;  
10   ...  
11 };  
12  
13 dev_t dev;  
14 struct xxstate *xsp;  
15  
16 #ifdef DEBUG    /* in debugging mode, log function call */  
17   cmn_err(CE_CONT, "%!s: xxopen function called.",  
18       ddi_binding_name(xsp->dip), getminor(dev));  
```
EXAMPLE 1 Using cmn_err() (Continued)

```c
19 #endif /* end DEBUG */
20 ...
21 /* display device power failure on system console */
22 if ((xsp->regp->csr & POWER) == OFF)
23    cmn_err(CE_NOTE, "OFF.",
24            ddi_binding_name(xsp->dip), getminor(dev));
25 ...
26 /* display warning if device has bad VTOC */
27 if (xsp->regp->csr & BADVTOC)
28    cmn_err(CE_WARN, "%s%d: xxopen: Bad VTOC.",
29            ddi_binding_name(xsp->dip), getminor(dev));
```

EXAMPLE 2 Using the %b conversion specification

This example shows how to use the %b conversion specification. Because of the leading ‘?’ character in the format string, this message will always be logged, but it will only be displayed when the kernel is booted in verbose mode.

```c
cmn_err(CE_CONT, "?reg=0x%b\n", regval, ",\020\3Intr\2Err\1Enable");
```

EXAMPLE 3 Using regval

When `regval` is set to (decimal) 13, the following message would be displayed:

```c
reg=0xd<Intr,,Enable>
```

EXAMPLE 4 Error Routine

The third example is an error reporting routine which accepts a variable number of arguments and displays a single line error message both in the system log and on the system console. Note the use of `vsprintf()` to construct the error message before calling `cmn_err()`.

```c
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#define MAX_MSG 256;

void xxerror(dev_info_t *dip, int level, const char *fmt,...)
{
    va_list ap;
    int instance;
    char buf[MAX_MSG], *name;

    instance = ddi_get_instance(dip);
    name = ddi_binding_name(dip);

    /* format buf using fmt and arguments contained in ap */
    va_start(ap, fmt);
    vcmn_err(9F)
    va_start(ap, fmt);
    ```
EXAMPLE 4 Error Routine  (Continued)

```c
vsprintf(buf, fmt, ap);
va_end(ap);
/* pass formatted string to cmn_err(9F) */

/* pass formatted string to cmn_err(9F) */

/* pass formatted string to cmn_err(9F) */

} /* cmn_err() */

SEE ALSO dmesg(1M), kernel(1M), printf(3C), ddi_binding_name(9F), sprintf(9F), va_arg(9F), va_end(9F), va_start(9F), vsprintf(9F)

Writing Device Drivers

WARNINGS cmn_err() with the CE_CONT argument can be used by driver developers as a driver code debugging tool. However, using cmn_err() in this capacity can change system timing characteristics.

NOTES Messages of arbitrary length can be generated using cmn_err(), but if the call to cmn_err() is made from high-level interrupt context and insufficient memory is available to create a buffer of the specified size, the message will be truncated to LOG_MSGSIZE bytes (see <sys/log.h>). For this reason, callers of cmn_err() that require complete and accurate message generation should post down from high-level interrupt context before calling cmn_err().

```
vsprintf(9F)

NAME     vsprintf – format characters in memory

SYNOPSIS  
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

char *vsprintf(char *buf, const char *fmt, va_list ap);

INTERFACE LEVEL
PARAMETERS  Solaris DDI specific (Solaris DDI).

buf        Pointer to a character string.
fmt        Pointer to a character string.
ap        Pointer to a variable argument list.

DESCRIPTION vsprintf() builds a string in buf under the control of the format fmt. The format is a
character string with either plain characters, which are simply copied into buf, or
conversion specifications, each of which converts zero or more arguments, again
copied into buf. The results are unpredictable if there are insufficient arguments for the
format; excess arguments are simply ignored. It is the user’s responsibility to ensure
that enough storage is available for buf.

ap contains the list of arguments used by the conversion specifications in fmt. ap is a
variable argument list and must be initialized by calling va_start(9F). va_end(9F) is
used to clean up and must be called after each traversal of the list. Multiple traversals
of the argument list, each bracketed by va_start(9F) and va_end(9F), are possible.

Each conversion specification is introduced by the % character, after which the
following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion.
The converted value will be right-justified and padded with leading zeroes if it has
fewer characters than the minimum.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion
character applies to a long (long long) integer argument. An l (ll) before any
other conversion character is ignored.

A character indicating the type of conversion to be applied:

d, D, o, O, x, X, u
The integer argument is converted to signed decimal (d, D), unsigned octal (o, O),
unsigned hexadecimal (x, X) or unsigned decimal (u), respectively, and copied. The
letters abcdef are used for x conversion. The letters ABCDEF are used for X
conversion.

c
The character value of the argument is copied.

b
This conversion uses two additional arguments. The first is an integer, and is
converted according to the base specified in the second argument. The second
argument is a character string in the form <base>[<arg> . . . ]. The base supplies the conversion base for the first argument as a binary value; \10 gives octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by < and >. Thus, the following function call would generate reg=3<BitTwo,BitOne>
 in buf.

vsprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")

The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL on SPARC, the string <nullstring> is used in its place; on IA, it is undefined.

% Copy a %; no argument is converted.

**RETURN VALUES**

vsprintf() returns its first parameter, buf.

**CONTEXT**

vsprintf() can be called from user, kernel, or interrupt context.

**EXAMPLES**

**EXAMPLE 1 Using vsprintf()**

In this example, xxerror() accepts a pointer to a dev_info_t structure dip, an error level level, a format fmt, and a variable number of arguments. The routine uses vsprintf() to format the error message in buf. Note that va_start(9F) and va_end(9F) bracket the call to vsprintf(). instance, level, name, and buf are then passed to cmn_err(9F).

```c
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#define MAX_MSG 256

void xxerror(dev_info_t *dip, int level, const char *fmt, ...)
{
    va_list ap;
    int instance;
    char *buf[MAX_MSG], *name;

    instance = ddi_get_instance(dip);
    name = ddi_binding_name(dip);

    /* format buf using fmt and arguments contained in ap */
    va_start(ap, fmt);
    vsprintf(buf, fmt, ap);
    va_end(ap);
```
EXAMPLE 1 Using vsprintf()  (Continued)

    /* pass formatted string to cmn_err(9F) */
    cmn_err(level, "%s%d: %s", name, instance, buf);
}

SEE ALSO cmn_err(9F), ddi_binding_name(9F), ddi_get_instance(9F), va_arg(9F)

Writing Device Drivers
### WR(9F)

**NAME**
WR, wr – get pointer to the write queue for this module or driver

**SYNOPSIS**
```c
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *WR(queue_t *q);
```

**INTERFACE LEVEL**
Architecture independent level 1 (DDI/DKI).

**PARAMETERS**
- `q` Pointer to the read queue whose write queue is to be returned.

**DESCRIPTION**
The `WR()` function accepts a read queue pointer as an argument and returns a pointer to the write queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a read queue. `WR()` will not check for queue type, and a system panic could result if the pointer is not to a read queue.

**RETURN VALUES**
The pointer to the write queue.

**CONTEXT**
`WR()` can be called from user or interrupt context.

**EXAMPLES**

**Example 1 Using WR()**

In a STREAMS `close(9E)` routine, the driver or module is passed a pointer to the read queue. These usually are set to the address of the module-specific data structure for the minor device.

```c
xxxclose(q, flag)
```

```c
queue_t *q;
int flag;
{
    q->q_ptr = NULL;
    WR(q)->q_ptr = NULL;
    ...
}
```

**SEE ALSO**
close(9E), OTHERQ(9F), RD(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*
NAME
WR, wr – get pointer to the write queue for this module or driver

SYNOPSIS
```
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *WR(queue_t *q);
```

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
PARAMETERS
q Pointer to the read queue whose write queue is to be returned.

DESCRIPTION
The WR() function accepts a read queue pointer as an argument and returns a pointer to the write queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a read queue. WR() will not check for queue type, and a system panic could result if the pointer is not to a read queue.

RETURN VALUES
The pointer to the write queue.

CONTEXT
WR() can be called from user or interrupt context.

EXAMPLES
EXAMPLE 1 Using WR()

In a STREAMS close(9E) routine, the driver or module is passed a pointer to the read queue. These usually are set to the address of the module-specific data structure for the minor device.

```
1 xxxclose(q, flag)
2 queue_t *q;
3 int flag;
4 {
5   q->q_ptr = NULL;
6   WR(q)->q_ptr = NULL;
7 }
```

SEE ALSO
close(9E), OTHERQ(9F), RD(9F)

Writing Device Drivers
STREAMS Programming Guide
Index

**Numbers and Symbols**
32-bit driver ID management routines —
  id32_alloc, 996
32-bit driver ID management routines —
  id32_free, 996
32-bit driver ID management routines —
  id32_lookup, 996

A
activate a new DMA window —
  ddi_dma_getwin, 444
add a fully initialized kstat to the system —
  kstat_install, 1037
add a soft interrupt, — ddi_add_softcintr, 303
add an interrupt handler, — ddi_add_intr, 299
address, return mapped virtual address —
  csx_GetMappedAddr, 127
adjmsg — trim bytes from a message, 50
Device power cycle advisory check —
  pm_trans_check, 1292
allocate and free a scsi_pkt structure —
  scsi_hba_pkt_alloc, 1413
allocate and free transport structures —
  scsi_hba_tran_alloc, 1418
allocate and free a scsi_pkt structure —
  scsi_hba_pkt_alloc, scsi_hba_pkt_free, 1413
allocate and free transport structures —
  scsi_hba_tran_alloc, scsi_hba_tran_free, 1418
allocate a message block — allocb, 51
allocate and free non-sequentially accessed memory (Continued)
  — ddi_iopb_alloc, 547
  — ddi_iopb_free, 547
allocate DMA handle —
  ddi_dma_alloc_handle, 385
allocate kernel memory
  — ddi_umem_alloc, 907
  — ddi_umem_free, 907
  — ddi_umem_zalloc, 907
  — kmem_alloc, 1008
  — kmem_free, 1008
  — kmem_zalloc, 1008
allocate memory for DMA transfer —
  ddi_dma_mem_alloc, 448
allocate space — rmalloc, 1342
allocate space from a resource map —
  rmalloc_wait, 1347
allocb — allocate a message block, 51
allow 64 bit transfers on SBus —
  ddi_dma_set_sbuss64, 462
ano_cancel — prevent cancellation of asynchronous I/O request, 54
aphysio — perform asynchronous physical I/O, 55
assert — expression verification, 57
associate STREAMS queue with bottom driver in queue — qassociate, 1314
asynchronous physical I/O — aphysio, 55
asynchronous STREAMS perimeter upgrade —
  qwriter, 1330
bcopy — copy data between address locations in kernel, 63
binds a system buffer to a DMA handle —
 ddi_dma.buf.bind.handle, 388
binds an address to a DMA handle —
 ddi_dma_addr.bind.handle, 380
bioclone — clone another buffer, 65
bioerror — indicate error in buffer header, 70
biofini — uninitialized a buffer structure, 71
bioinit — initialize a buffer structure, 72
biomodi — check if a buffer is modified, 73
biureset — reuse a private buffer header after
 I/O is complete, 74
biosize — returns size of a buffer structure, 75
bufcall — call a function when a buffer becomes
 available, 81, 1566
bufcall, call a function when a buffer becomes
 available, 81
buffer header
 indicate error — bioerror, 70
 reuse a private buffer header after I/O is
 complete — biureset, 74
busy-wait for specified interval —
 ddrv_usecwait, 945
byte streams, compare two — bcmp, 62
bytes, size
 convert size in pages — ptob, 1301
 convert to size in memory pages (round
down) — btop, 79
 convert to size in memory pages (round up)
 — btopr, 80
character strings
 compare two null terminated strings —
 strcmp, strncmp, 1494
 convert between an integer and a decimal
 string — stoi, numtos, 1491
 copy a string from one location to another —
 strcpy, strncpy, 1495
determine the number of non-null bytes in a
string — strlen, 1496
find a character in a string — strchr, 1493
format in memory — sprintf, 1489
check data access and DMA handles, 312
check device state, 485
check for an available buffer — testb, 1561
check for the existence of a property —
 ddi_prop_exists, 707
check if a buffer is modified — biomodi, 73
CIS tuple
 first tuple — csx_GetFirstTuple, 124
 next tuple — csx_GetNextTuple, 124
clear client event mask —
 csx_ReleaseSocketMask, 262
client, register client — csx_RegisterClient, 212
client event mask
 return client event mask —
 csx_GetEventMask, 270
 set client event mask —
 csx_SetEventMask, 270
client return
 — csx_GetFirstClient, 122
 — csx_GetNextClient, 122
cclone another buffer — bioclone, 65
concatenate two message blocks — linkb, 1054
count two message blocks, 62
control driver notification of user accesses —
 ddi_mapdev_intercept, 593
ddi_mapdev_noInterceptor, 593

man pages section 9: DDI and DKI Kernel Functions • April 2003
control device components’ availability for Power Management
— pm_busy_component, 1274
— pm_idle_component, 1274
control the validation of memory address translations
— devmap_load, 937
— devmap_unload, 937
convert a DMA segment to a DMA address cookie — ddi_dma_segtocookie, 460
convert clock ticks to microseconds —
drv_hztousec, 942
convert device sizes — csx_ConvertSize, 106
convert device speeds —
csx_ConvertSpeed, 107
convert error return codes to text strings —
csx_Error2Text, 113
convert events to text strings —
csx_Event2Text, 114
convert microseconds to clock ticks —
drv_usectohz, 944
copy data from one device register to another device register — ddi_device_copy, 325
create minor nodes for client —
csx_MakeDeviceNode, 138
create a minor node for this device —
ddi_create_minor_node, 323
create and initialize a new kstat —
ksstat_create, 1035
create driver-controlled mapping of device —
ddi_mapdev, 590
csx_AccessConfigurationRegister — read or write a PC Card Configuration Register, 104
csx_ConvertSize — convert device sizes, 106
csx_ConvertSpeed — convert device speeds, 107
csx_CS_DDI_Info — obtain DDI information, 108
csx_DeRegisterClient — remove client from Card Services list, 110
csx_DupHandle — duplicate access handle, 111
csx_Error2Text — convert error return codes to text strings, 113
csx_Event2Text — convert events to text strings, 114
csx_FreeHandle — free access handle, 115
csx_Get16 — read from device register, 119
csx_Get32 — read from device register, 119
csx_Get64 — read from device register, 119
csx_Get8 — read from device register, 119
csx_GetEventMask — return client event mask, 270
csx_GetFirstClient — return first client, 122
csx_GetFirstTuple — return first CIS tuple, 124
csx_GetHandleOffset — return current access handle offset, 126
csx_GetMappedAddr — return mapped virtual address, 127
csx_GetNextClient — return next client, 122
csx_GetNextTuple — return next CIS tuple, 124
csx_GetStatus — return status of PC Card and socket, 132
csx_GetTupleData — return data portion of tuple, 136
csx_MakeDeviceNode — create minor nodes for client, 138
csx_MapLogSocket — return physical socket number, 140
csx_MapMemPage — map memory area on PC Card, 141
csx_ModifyConfiguration — modify PC Card configuration, 142
csx_ModifyWindow — modify window attributes, 144
csx_Parse_CISTPL_BATTERY — parse Battery Replacement Date tuple, 146
csx_Parse_CISTPL_BYTEORDER — parse Byte Order tuple, 147
csx_Parse_CISTPL_CFTABLE_ENTRY — parse Card Configuration Table tuple, 149
csx_Parse_CISTPL_CONFIG — parse Configuration tuple, 155
csx_Parse_CISTPL_DATE — parse Card Initialization Date tuple, 158
csx_Parse_CISTPL_DEVICE — parse Device Information tuple for Common Memory, 159
csx_Parse_CISTPL_DEVICE_A — parse Device Information tuple for Attribute Memory, 159
csx_Parse_CISTPL_DEVICE_OA — parse Other Condition Device Information tuple for Attribute Memory, 159
csx_Parse_CISTPL_DEVICE_OC — parse Other Condition Device Information tuple for Common Memory, 159
csx_Parse_CISTPL_DEVICEGEO — parse Device Geo tuple, 165
ddi_create_minor_node — create a minor node for this device, 323

ddi_dev_is_needed — inform the system that a device’s component is required, 368
ddi_dev_report_fault, 376

DDI device access, slave access only —

ddi_slaveonly, 874

ddi_device_copy — copy data from one device register to another device register, 325

DDI device critical region of control

enter — ddi_enter_critical, 470
exit — ddi_exit_critical, 470

DDI device information structure

find parent — ddi_get_parent, 507
get the root of the dev_info tree —

ddi_root_node, 866
remove a minor node for this devinfo —

ddi_remove_minor_node, 823

DDI device instance number, get —

ddi_get_instance, 491

DDI device mapping

ddi_mapdev — create driver-controlled mapping of device, 590

ddi_mapdev_intercept — control driver notification of user accesses, 593

ddi_mapdev_nointercept — control driver notification of user accesses, 593

devmap_default_access — device mapping access entry point, 921

DDI device registers

map — ddi_map_regs, 599
return the number of register sets —

ddi_dev_nregs, 374
return the size — ddi_dev_regsize, 375
unmap — ddi_unmap_regs, 599

DDI device’s private data area

get the address —

ddi_get_driver_private, 486
set the address —

ddi_set_driver_private, 486

DDI device virtual address

read 16 bit — ddi_peek16, 675
read 32 bit — ddi_peek32, 675
read 64 bit — ddi_peek64, 675
read 8 bit — ddi_peek8, 675
read a value — ddi.peek, 693
write 16 bit — ddi_poke16, 693
write 32 bit — ddi_poke32, 693

DDI device virtual address (Continued)

write 64 bit — ddi_poke64, 693
write 8 bit — ddi_poke8, 693
write a value — ddi_poke, 693

ddi_device_zero — zero fill the device register, 327

ddi_devid_compare — Kernel interfaces for device ids, 329

ddi_devid_free — Kernel interfaces for device ids, 329

ddi_devid_init — Kernel interfaces for device ids, 329

ddi_devid_register — Kernel interfaces for device ids, 329

ddi_devid_sizeof — Kernel interfaces for device ids, 329

ddi_devid_unregister — Kernel interfaces for device ids, 329

ddi_devid_valid — Kernel interfaces for device ids, 329

DDI devinfo node name

return — ddi_binding_name, 309
return — ddi_get_name, 309
return — ddi_node_name, 665

DDI direct memory access, convert DMA handle to DMA addressing cookie —

ddi_dma_htoc, 446

DDI direct memory access services

allocate consistent memory—

ddi_iopb_alloc, 601
convert a DMA cookie — ddi_dma_coff, 394
easier DMA setup —

ddi_dma_addr_setup, 383
easier DMA setup —

ddi_dma_buf_setup, 391
find minimum alignment and transfer size for device — ddi_iomin, 546
find post DMA mapping alignment and minimum effect properties —

ddi_dma_dealign, 396
free consistent memory —

ddi_iopb_free, 601
report current DMA window offset and size — ddi_dma_curwin, 395
setup DMA mapping —

ddi_dma_setup, 455, 457, 460
setup DMA resources —

ddi_dma_setup, 463
DDI direct memory access services (Continued)

shift current DMA window —
  ddi_dma_movwin, 451

tear down DMA mapping —
  ddi_dma_free, 441

ddi_dma_addr_bind_handle — binds an
  address to a DMA handle, 380

ddi_dma_alloc_handle — allocate DMA
  handle, 385

ddi_dma_buf_bind_handle — binds a system
  buffer to a DMA handle, 388

ddi_dma_burstsizes — find out the allowed
  burst sizes for a DMA mapping, 393

ddi_dma_free_handle — free DMA handle, 442

ddi_dma_get_attr, 443

ddi_dma_getwin — activate a new DMA
  window, 444

ddi_dma_mem_alloc — allocate memory for
  DMA transfer, 448

ddi_dma_mem_free — free previously allocated
  memory, 450

ddi_dma_nextcookie — retrieve subsequent
  DMA cookie, 453

ddi_dma_nextseg — get next DMA
  segment, 455

ddi_dma_nextwin — get next DMA
  window, 457

ddi_dma_numwin — retrieve number of DMA
  windows, 459

ddi_dma_segtocookie — convert a DMA
  segment to a DMA address cookie, 460

ddi_dma_set_sbus64 — allow 64 bit transfers on
  SBus, 462

ddi_dma_sync — synchronize CPU and I/O
  views of memory, 465

ddi_dma_unbind_handle — unbinds the
  address in a DMA handle, 467

ddi_dmae — system DMA engine
  functions, 402

ddi_dmae_1stparty — system DMA engine
  functions, 402

ddi_dmae_alloc — system DMA engine
  functions, 402

ddi_dmae_disable — system DMA engine
  functions, 402

ddi_dmae_enable — system DMA engine
  functions, 402

ddi_dmae_getattr — system DMA engine
  functions, 402

ddi_dmae_getcnt — system DMA engine
  functions, 402

ddi_dmae_getlim — system DMA engine
  functions, 402

ddi_dmae_prog — system DMA engine
  functions, 402

ddi_dmae_release — system DMA engine
  functions, 402

ddi_dmae_stop — system DMA engine
  functions, 402

ddi_dmae_major, 468

ddi_dmae_name — return normalized driver
  name, 469

ddi_ffs — find first (last) bit set in a long
  integer, 472

ddi_fls — find first (last) bit set in a long
  integer, 472

ddi_get_iblock_cookie — get interrupt block
  cookie, 299

ddi_get_kt_did, 492

ddi_get_lbolt, returns the value of lbolt, 495

ddi_get_name — return driver binding
  name, 309

ddi_get_pid, returns the process ID, 508

ddi_get_soft_iblock_cookie — get soft interrupt
  block cookie, 303

ddi_get_time, returns the current time in
  seconds, 529

ddi_get16 — read data from the device, 480

ddi_get32 — read data from the device, 480

ddi_get64 — read data from the device, 480

ddi_get8 — read data from the device, 480

ddi_getin xor, display a SCSI request sense
  message, 490

ddi_in panic — determine if system is in panic
  state, 532

DDI information — csx_CS_DDI_Info, 108

DDI interrupt handling
  add an interrupt — ddi_add_intr, 299
  get interrupt block cookie —
    ddi_get_iblock_cookie, 299
  indicate interrupt handler type —
    ddi_intr_hilevel, 533
  remove an interrupt — ddi_remove_intr, 299
  return the number of interrupt specifications
    — ddi_dev_nintrs, 373
ddi_io_get16 — read data from the mapped device register in I/O space, 538
ddi_io_get32 — read data from the mapped device register in I/O space, 538
ddi_io_get8 — read data from the mapped device register in I/O space, 538
ddi_io_getb — read data from the mapped device register in I/O space, 538
ddi_io_getl — read data from the mapped device register in I/O space, 538
ddi_io_getw — read data from the mapped device register in I/O space, 538

ddi_io_put16 — write data to the mapped device register in I/O space, 555
ddi_io_put32 — write data to the mapped device register in I/O space, 555
ddi_io_put8 — write data to the mapped device register in I/O space, 555
ddi_io_putb — write data to the mapped device register in I/O space, 555
ddi_io_putl — write data to the mapped device register in I/O space, 555
ddi_io_putw — write data to the mapped device register in I/O space, 555

ddi_io_rep_get16 — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_get32 — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_get8 — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_getb — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_getl — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_getw — read multiple data from the mapped device register in I/O space, 567
ddi_io_rep_put16 — write multiple data to the mapped device register in I/O space, 579
ddi_io_rep_put32 — write multiple data to the mapped device register in I/O space, 579
ddi_io_rep_put8 — write multiple data to the mapped device register in I/O space, 579
ddi_io_rep_putb — write multiple data to the mapped device register in I/O space, 579
ddi_io_rep_putl — write multiple data to the mapped device register in I/O space, 579
ddi_io_rep_putw — write multiple data to the mapped device register in I/O space, 579

ddi_iopb_alloc — allocate and free non-sequentially accessed memory, 547
ddi_iopb_free — allocate and free non-sequentially accessed memory, 547
ddi_log_sysevent — 587

ddi_mapdev — create driver-controlled mapping of device, 590
ddi_mapdev_intercept — control driver notification of user accesses, 593
ddi_mapdev_set_device_acc_attr — Set the device attributes for the mapping, 597
ddi_mem_get16 — read data from mapped device in the memory space or allocated DMA memory, 608
ddi_mem_get32 — read data from mapped device in the memory space or allocated DMA memory, 608
ddi_mem_get64 — read data from mapped device in the memory space or allocated DMA memory, 608
ddi_mem_put16 — write data to mapped device in the memory space or allocated DMA memory, 619
ddi_mem_put32 — write data to mapped device in the memory space or allocated DMA memory, 619
ddi_mem_put64 — write data to mapped device in the memory space or allocated DMA memory, 619
ddi_mem_rep_get16 — read data from mapped device in the memory space or allocated DMA memory, 635
ddi_mem_rep_get32 — read data from mapped device in the memory space or allocated DMA memory, 635
ddi_mem_rep_get64 — read data from mapped device in the memory space or allocated DMA memory, 635
ddi_mem_rep_get8 — read data from mapped device in the memory space or allocated DMA memory, 635
ddi_mem_rep_put16 — write data to mapped device in the memory space or allocated DMA memory, 651
ddi_mem_rep_put32 — write data to mapped device in the memory space or allocated DMA memory, 651
ddi_mem_rep_put64 — write data to mapped device in the memory space or allocated DMA memory, 651

ddi_mem_rep_put8 — write data to mapped device in the memory space or allocated DMA memory, 651

DDI memory mapping

map a segment — ddi_memrepmap, 868
map a segment — devmap_setup, 933

ddi_mmap_get_model — return data model type of current thread, 661

ddi_model_convert_from — determine data model type mismatch, 663

ddi_model_convert_from — Determine if there is a need to translate shared data structure contents, 663

ddi_no_info — returns DDI_FAILURE, a convenience for drivers implementing DLPI style 2 services, 666

ddi_node_name — return the devinfo node name, 665

DDI page size conversions

— ddi_btop, 310
— ddi_btopr, 310
— ddi_ptob, 310

ddi_prop_exists — check for the existence of a property, 707

ddi_prop_get_int — look up integer property, 716

ddi_prop_lookup — lookup property information, 719

ddi_prop_lookup_byte_array — lookup property information, 719

ddi_prop_lookup_int_array — lookup property information, 719

ddi_prop_lookup_string — lookup property information, 719

ddi_prop_lookup_string_array — lookup property information, 719

ddi_prop_update — update property information, 769

ddi_prop_update_byte_array — update property information, 769

ddi_prop_update_int — update property information, 769

ddi_prop_update_int_array — update property information, 769

ddi_prop_update_string — update property information, 769

DDI property management

create properties for leaf device drivers —

— ddi_prop_create, 704
— ddi_getlongprop, 735
— ddi_getlongprop_buf, 735
— ddi_getprop, 735
— ddi_getproplen, 735
— ddi_prop_op, 735

modify properties for leaf device drivers —

— ddi_prop_modify, 704

remove all properties for leaf device drivers —

— ddi_prop_remove_all, 704

remove properties for leaf device drivers —

— ddi_prop_remove, 704

— ddi_prop_undefined, 704

ddi_put16 — write data to the device, 807

ddi_put32 — write data to the device, 807

ddi_put64 — write data to the device, 807

ddi_put8 — write data to the device, 807

ddi_regs_map_free — free mapped register address space, 817

ddi_regs_map_setup — set up a mapping for a register address space, 818

ddi_remove_intr — remove an interrupt handler, 299

ddi_remove_softintr — remove a soft interrupt, 303

ddi_removing_power, 831

ddi_rep_get16 — read data from the mapped memory address, device register or allocated DMA memory address, 839

ddi_rep_get32 — read data from the mapped memory address, device register or allocated DMA memory address, 839

ddi_rep_get64 — read data from the mapped memory address, device register or allocated DMA memory address, 839

ddi_rep_get8 — read data from the mapped memory address, device register or allocated DMA memory address, 839

ddi_rep_getb — read data from the mapped memory address, device register or allocated DMA memory address, 839

1598 man pages section 9: DDI and DKI Kernel Functions • April 2003
ddi_rep_getl — read data from the mapped memory address, device register or allocated DMA memory address, 839

DDI self identifying devices, tell whether a device is self-identifying —

ddi_dev_is_sid, 370

DDI soft state utility routines (Continued)

generate pointer to soft state —

ddi_get_soft_state, 875

initialize state — ddi_soft_state_init, 875

remove all state info —

    ddi_soft_state_fini, 875

DDI_SUSPEND, 831

ddi_trigger_softintr — trigger a soft interrupt, 303

ddi_umem_alloc — allocate kernel memory, 907

ddi_umem_free — allocate kernel memory, 907

ddi_umem_lock — Locks and unlocks memory pages, 913

ddi_umem_zalloc — allocate kernel memory, 907

default SCSI HBA probe function —

    scsi_hba_probe, 1417

delay — delay process execution for a specified number of clock ticks, 919

deregister client from Card Services list —

    csx_DeregisterClient, 110

determine data model type mismatch —

    ddi_model_convert_from, 663

Device Driver Interface

See DDI

device mapping access entry point —

    devmap_default_access, 921

device switch tables, return function for insignificant entries — nulldev, 1104

devices

    get major device number — getmajor, 965

    get minor device number — getminor, 966

    make device number from major and minor numbers — makedevice, 1065

devices, non-pollable, error return function —

    nochpoll, 1101

devmap_default_access — device mapping access entry point, 921

devmap_devmem_setup — Set driver memory mapping parameters, 924

    devmap_devmem_setup(), 923

    devmap_umem_setup(), 924

devmap_do_ctxmgt — perform device context switching on a mapping, 926

devmap_load — control the validation of memory address translations, 937
devmap_set_ctx_timeout — set context management timeout value, 931
devmap_umem_setup — Set driver memory mapping parameters, 924
devmap_unload — control the validation of memory address translations, 937
disksort — single direction elevator seek sort for buffers, 939
display a SCSI request sense message, scsi_vu_errmsg, 1460
DMA attribute structure, 443
DMA mapping, the allowed burst sizes for — ddi_dma_burstsizes, 393
driver buffers
   copy data — ddi_copyin, 316
   copy data from driver — ddi_copyout, 319
   copy data from driver to user program — copyout, 102
   copy data from user program — copyin, 98
driver error messages, display an error message or panic the system — cmn_err, 88
driver privilege — drv_priv, 943
drv_getparm — retrieve kernel state information, 940
   drv_hztousec — convert clock ticks to microseconds, 942
   drv_priv — determine driver privilege, 943
   drv_usectohz — convert microseconds to clock ticks, 944
   drv_usecwait — busy-wait for specified interval, 945
dupb — duplicate a message block descriptor, 946
duplicate a message — dupmsg, 949
duplicate a message block descriptor — dupb, 946
duplicate access handle — csx_DupHandle, 111
dupmsg — duplicate a message, 949
derror return function for illegal entries — nodev, 1102
event mask
   return client event mask —
      csx_GetEventMask, 270
   set client event mask —
      csx_SetEventMask, 270
   events converted to text strings —
      csx_Event2Text, 114
expression verification, — assert, 57
F
find first (last) bit set in a long integer —
   ddi_ffs, 472
ddi_ffs, 472
first CIS tuple — csx_GetFirstTuple, 124
flushband — flush messages for specified priority band, 954
free access handle — csx_FreeHandle, 115
free DMA handle, —
   ddi_dma_free_handle, 442
free mapped register address space —
   ddi_regs_map_free, 817
free previously allocated memory —
   ddi_dma_mem_free, 450
free space — rmfree, 1348
freerbuf — free a raw buffer header, 961
freeze, thaw the state of a stream —
   freezestr, 962
unfreezestr, 962
freezestr — freeze, thaw the state of a stream, 962
G
generic tuple parser — csx_ParseTuple, 206
get interrupt block cookie, —
   ddi_get_iblock_cookie, 299
get kernel internal minor number from an external dev_t, scsi_vu_errmsg, 490
get next DMA segment —
   ddi_dma_nextseg, 455
get next DMA window —
   ddi_dma_nextwin, 457
E
enable/disable accesses to the PCI Local Bus Configuration space.
   — pci_config_setup, 1264
   — pci_config_teardown, 1264
error return codes converted to text strings —
   csx_Error2Text, 113
get soft interrupt block cookie, —
  ddi_get_soft_iblock_cookie, 303
gethrtime, 964
getmajor — get major device number, 965
getminor — get minor device number, 966
getbuf — get a raw buffer header, 970
gld_mac_alloc — allocate a GLD mac_info
  structure, 971
gld_mac_free — free a GLD mac_info
  structure, 971
gld_recv — pass the inbound packet
  upstream, 972
gld_register — link the driver with the GLD
  framework, 971
gld_sched — reschedule stalled outbound
  packets, 972
gld_unregister — unlink the driver from the
  GLD framework, 972

H
handle variable argument list
  — va_arg, 1573
  — va_copy, 1573
  — va_end, 1573
  — va_start, 1573
high resolution time, 964

I
I/O, block, suspend processes pending
  completion — biowait, 76
I/O, buffer, release buffer and notify processes
  — biodone, 68
I/O, paged request
  allocate virtual address space —
    bp_mapin, 77
deallocate virtual address space —
    bp_mapout, 78
I/O, physical
  — minphys, 1272
  — physio, 1272
I/O error, return — geterror, 963
I/O resources
  release I/O resources — csx_ReleaseIO, 254
  request I/O resources — csx_RequestIO, 254
id32_alloc — 32-bit driver ID management
  routines, 996
id32_free — 32-bit driver ID management
  routines, 996
id32_lookup — 32-bit driver ID management
  routines, 996
inb — read from an I/O port, 999
inform the system that a device’s component is
  required. — ddi_dev_is_needed, 368
initialize a named kstat —
    kstat_named_init, 1038
initialize a buffer structure — bioinit, 72
inl — read from an I/O port, 999
interrupt handling
  add an interrupt — ddi_add_intr, 299
  get interrupt block cookie —
    ddi_get_iblock_cookie, 299
  remove an interrupt — ddi_remove_intr, 299
inv — read from an I/O port, 999
IOC_CONVERT_FROM — Determine if there is
  a need to translate M_IOCTL contents, 1007
IRQ resource
  release IRQ resource — csx_ReleaseIRQ, 259
  request IRQ resource —
    csx_RequestIRQ, 259

K
kernel memory cache allocator operations —
  kmem_cache_alloc, 1015
kernel memory cache allocator operations —
  kmem_cache_create, 1015
kernel memory cache allocator operations —
  kmem_cache_destroy, 1015
kernel memory cache allocator operations —
  kmem_cache_free, 1015
kernel address locations, between locations —
  bcopy, 63
kernel addresses, get page frame number —
  hat_getkpfnum, 995
Kernel interfaces for device ids
  — ddi_devid_compare, 329
  — ddi_devid_free, 329
  — ddi_devid_init, 329
  — ddi_devid_register, 329
  — ddi_devid_sizeof, 329
  — ddi_devid_unregister, 329
Kernel interfaces for device ids (Continued)
— ddi_devid_valid, 329
kernel modules, dynamic loading
  add loadable module — mod_install, 1074
  query loadable module — mod_info, 1074
  remove loadable module —
    mod_remove, 1074
kernel state information — drv_getparm, 940
kmem_alloc — allocate kernel memory, 1008
kmem_cache_alloc — kernel memory cache
  allocator operations, 1015
kmem_cache_create — kernel memory cache
  allocator operations, 1015
kmem_cache_destroy — kernel memory cache
  allocator operations, 1015
kmem_cache_free — kernel memory cache
  allocator operations, 1015
kmem_free — allocate kernel memory, 1008
kmem_zalloc — allocate kernel memory, 1008
kstat_create — create and initialize a new
  kstat, 1035
kstat_delete — remove a kstat from the
  system, 1036
kstat_install — add a fully initialized kstat to
  the system, 1037
kstat_named_init — initialize a named
  kstat, 1038
kstat_queue — update I/O kstat statistics, 1040
kstat_runq_back_to_waitq — update I/O kstat
  statistics, 1040
kstat_runq_enter — update I/O kstat
  statistics, 1040
kstat_runq_exit — update I/O kstat
  statistics, 1040
kstat_waitq_enter — update I/O kstat
  statistics, 1040
kstat_waitq_exit — update I/O kstat
  statistics, 1040
kstat_waitq_to_runq — update I/O kstat
  statistics, 1040

look up integer property —
  ddi_prop_get_int, 716
look up per-device-type scsi-options property
  — scsi_get_device_type_scsi_options, 1398
lookup property information
  — ddi_prop_lookup, 719
  — ddi_prop_lookup_byte_array, 719
  — ddi_prop_lookup_int_array, 719
  — ddi_prop_lookup_string, 719
  — ddi_prop_lookup_string_array, 719

major device number, 468
makedevice — make device number from major
  and minor numbers, 1065
map memory area on PC Card —
  csx_MapMemPage, 141
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_boolean, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_byte, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_byte_array, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int16, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int16_array, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int32, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int32_array, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int64, 1146
match name and type indicated by the interface
  name and retrieve data value —
  nvlist_lookup_int64_array, 1146

linkb — concatenate two message blocks, 1054
Locks and unlocks memory pages —
  ddi_umem_lock, 913
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_string, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_string_array, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint16, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint16_array, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint32, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint32_array, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint64, 1146
match name and type indicated by the interface
name and retrieve data value —
   nvlist_lookup_uint64_array, 1146
max — return the larger of two integers, 1066
memory, clear for a given number of bytes —
bzero, 84
min — return the lesser of two integers, 1067
minor node for device, create —
   ddi_create_minor_node, 323
modify PC Card configuration —
   csx_ModifyConfiguration, 142
modify window attributes —
   csx_ModifyWindow, 144
mt-streams — STREAMS multithreading, 1078
mutex routines
   — mutex, 1080
   — mutex_destroy, 1080
   — mutex_enter, 1080
   — mutex_exit, 1080
   — mutex_init, 1080
   — mutex_owned, 1080
   — mutex_tryenter, 1080

muninclusion lock
   See mutex

N
next CIS tuple — csx_GetNextTuple, 124
nodes, create minor nodes for client —
   csx_MakeDeviceNode, 138
Notify pm framework of autonomous power
level change —
   pm_power_has_changed, 1285
notify target driver of bus resets —
   scsi_reset_notify, 1446
nvlist_lookup_boolean — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_byte — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_byte_array — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int16 — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int16_array — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int32 — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int32_array — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int64 — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_int64_array — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_string — match name and type
indicated by the interface name and retrieve
data value, 1146
nvlist_lookup_string_array — match name and type
indicated by the interface name and retrieve
data value, 1146

Index 1603
nvlist_lookup_uint16 — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_lookup_uint16_array — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_lookup_uint32 — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_lookup_uint32_array — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_lookup_uint64 — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_lookup_uint64_array — match name and type indicated by the interface name and retrieve data value, 1146
nvlist_remove — remove name-value pairs, 1184
nvlist_remove_all — remove name-value pairs, 1184

O
obtain DDI information —
   csx_CS_DDI_Info, 108
OTHERQ — get pointer to queue’s partner queue, 1226
outb — write to an I/O port, 1228
outl — write to an I/O port, 1228
outw — write to an I/O port, 1228

P
panic state — ddi_in_panic, 532
parse Battery Replacement Date tuple —
   csx_Parse_CISTPL_BATTERY, 146
parse Byte Order tuple —
   csx_Parse_CISTPL_BYTEORDER, 147
parse Card Configuration Table tuple —
   csx_Parse_CISTPL_CFTABLE_ENTRY, 149
parse Card Initialization Date tuple —
   csx_Parse_CISTPL_DATE, 158
parse Configuration tuple —
   csx_Parse_CISTPL_CONFIG, 155
parser, for tuples (generic) —
csx_ParseTuple, 206
pci_config_get16 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_get32 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_get64 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_get8 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_getb — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_getl — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_getll — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_get16 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_get32 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_get64 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_get8 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_getb — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_getl — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_getll — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_confi_getw — read or write single datum of various sizes to the PCI Local Bus Configuration space, 1238
pci_config_setup — enable/disable accesses to the PCI Local Bus Configuration space, 1264
pci_config_teardown — enable/disable accesses to the PCI Local Bus Configuration space, 1264
pci_report_pmcap — Report power management capability of a PCI device, 1266
perform device context switching on a mapping — devmap_do_ctxmgt, 926
pm_busy_component — control device components’ availability for Power Management, 1274
pm_idle_component — control device components’ availability for Power Management, 1274
pm_power_has_changed — Notify pm framework of autonomous power level change, 1285
pm_raise_power — Raise or lower power of components, 1287
pm_trans_check — advisory check for device power cycles, 1292
pollwakeup — inform a process that an event has occurred, 1294
proc_ref — send a signal to a process, 1297
proc_signal — send a signal to a process, 1297
proc_unref — send a signal to a process, 1297
put — call a STREAMS put procedure, 1304
Q
qassociate — associate STREAMS queue with bottom driver in queue, 1314
qbufcall — call a function when a buffer becomes available, 1316
qtimeout — execute a function after a specified length of time, 1323
qunbufcall — cancel a pending qbufcall request, 1324
quntimeout — cancel previous timeout function call, 1325
qwait — STREAMS wait routines, 1326
qwait_sig — STREAMS wait routines, 1326
qwriter — asynchronous STREAMS perimeter upgrade, 1330

R
Raise or lower power of components –
   pm_raise_power, 1287
raw buffer
   free a raw buffer header — freerbuf, 961
   get a raw buffer header — getbuf, 970
RD — get pointer to the read queue, 1331
read from an I/O port — inb, 999
read from an I/O port — inl, inl, 999
read from an I/O port — repinsb, repinsb, 999
read from an I/O port — repinsd, repinsd, 999
read from an I/O port — repinsw, repinsw, 999
read data from mapped device in the memory space or allocated DMA memory
   — ddi_mem_get16, 608
   — ddi_mem_get32, 608
   — ddi_mem_get64, 608
   — ddi_mem_get8, 608
   — ddi_mem_rep_get16, 635
   — ddi_mem_rep_get32, 635
   — ddi_mem_rep_get64, 635
   — ddi_mem_rep_get8, 635
read data from the device
   — ddi_get16, 480
   — ddi_get32, 480
   — ddi_get64, 480
   — ddi_get8, 480
read data from the mapped device register in I/O space
   — ddi_io_get16, 538
   — ddi_io_get32, 538
   — ddi_io_get8, 538
   — ddi_io_getb, 538
   — ddi_io_getl, 538
   — ddi_io_getw, 538
read data from the mapped memory address, device register or allocated DMA memory address (Continued)
   — ddi_rep_getb, 839
   — ddi_rep_getl, 839
   — ddi_rep_getll, 839
   — ddi_rep_getw, 839
read from device register
   — csx_Get16, 119
   — csx_Get32, 119
   — csx_Get64, 119
   — csx_Get8, 119
read multiple data from the mapped device register in I/O space
   — ddi_io_rep_get16, 567
   — ddi_io_rep_get32, 567
   — ddi_io_rep_get8, 567
   — ddi_io_rep_getb, 567
   — ddi_io_rep_getl, 567
   — ddi_io_rep_getw, 567
read or write a PC Card Configuration Register
   — csx_AccessConfigurationRegister, 104
read or write single datum of various sizes to the PCI Local Bus Configuration space
   — pci_config_get16, 1238
   — pci_config_get32, 1238
   — pci_config_get64, 1238
   — pci_config_get8, 1238
   — pci_config_getb, 1238
   — pci_config_getl, 1238
   — pci_config_getll, 1238
   — pci_config_getw, 1238
   — pci_config_put16, 1238
   — pci_config_put32, 1238
   — pci_config_put64, 1238
   — pci_config_put8, 1238
   — pci_config_putb, 1238
   — pci_config_putl, 1238
   — pci_config_putll, 1238
   — pci_config_putw, 1238
read repetitively from device register
   — csx_RepGet16, 240
   — csx_RepGet32, 240
   — csx_RepGet64, 240
   — csx_RepGet8, 240
readers/writer lock functions
   — rw_destroy, 1368
   — rw_downgrade, 1368
readers/writer lock functions (Continued)
  — rw_enter, 1368
  — rw_exit, 1368
  — rw_init, 1368
  — rw_read_locked, 1368
  — rw_tryenter, 1368
  — rw_tryupgrade, 1368
  — rwlock, 1368
register client — csx_RegisterClient, 212
release client event mask —
  csx_ReleaseSocketMask, 262
release I/O resources — csx_ReleaseIO, 254
release IRQ resource — csx_ReleaseIRQ, 259
release window resources —
  csx_ReleaseWindow, 264
release configuration on PC Card —
  csx_ReleaseConfiguration, 215
remove name-value pairs —
  nvlist_remove, 1184
remove name-value pairs —
  nvlist_remove_all, 1184
remove a kstat from the system —
  kstat_delete, 1036
remove a soft interrupt, —
  ddi_remove_softintr, 303
remove an interrupt handler, —
  ddi_remove_intr, 299
remove client from Card Services list —
  csx_DeregisterClient, 110
repinsb — read from an I/O port, 999
repinsd — read from an I/O port, 999
repinsw — read from an I/O port, 999
Report a hardware failure, 376
Report power management capability of a PCI
  device-pci_report_pmcap, 1266
repoutsb — write to an I/O port, 1228
repoutsd — write to an I/O port, 1228
repoutsw — write to an I/O port, 1228
request client event mask —
  csx_RequestSocketMask, 262
request I/O resources — csx_RequestIO, 254
request IRQ resource — csx_RequestIRQ, 259
request window resources —
  csx_RequestWindow, 264
reset a function on a PC card —
  csx_ResetFunction, 269
resource map — rmallocmap, 1345
resource map (Continued)
  free resource maps — rmallocmap, 1345
retrieve number of DMA windows —
  ddi_dma_numwin, 459
retrieve subsequent DMA cookie —
  ddi_dma_nextcookie, 453
return client event mask —
  csx_GetEventMask, 270
return client —
  — csx_GetFirstClient, 122
  — csx_GetNextClient, 122
return current access handle offset —
  csx_GetHandleOffset, 126
return data model type of current thread —
  ddi_mmap_get_model, 661
return data portion of tuple —
  csx_GetTupleData, 136
return driver binding name —
  — ddi_binding_name, 309
  — ddi_get_name, 309
return index matching capability string —
  scsi_hba_lookup_capstr, 1411
return normalized driver name —
  ddi_driver_name, 469
return physical socket number —
  ddi_MapLogSocket, 140
return status of PC Card and socket —
  csx_GetStatus, 132
return the devinfo node name —
  ddi_node_name, 665
return the larger of two integers — max, 1066
return the lesser of two integers — min, 1067
return tuple —
  first CIS tuple — csx_GetFirstTuple, 124
  next CIS tuple — csx_GetNextTuple, 124
returns DDI_FAILURE, a convenience for
  drivers implementing DLPI style 2 services — ddi_no_info, 666
returns size of a buffer structure — biosize, 75
returns the current time in seconds, ddi_get_time, 529
returns the process ID, ddi_get_pid, 508
returns the value of lbolt, returns the value of
  lbolt, 495
rmalloc — allocate space from a resource
  map, 1342
rmalloc_wait — allocate space from a resource
  map, 1347
rmfree — free space back into a resource map, 1348

S
SAMESTR — test if next queue is in the same stream, 1380
SCSI Host Bus Adapter system initialization and completion routines
   — scsi_hba_init, 1410
scsi_abort — abort a SCSI command, 1382
scsi_alloc_consistent_buf — scsi dma utility for allocating an I/O buffer for SCSI DMA, 1383
scsi_cname — decode SCSI commands, 1385
SCSI commands, make packet
   — makecom, 1055
   — makecom_g0, 1055
   — makecom_g0_s, 1055
   — makecom_g1, 1055
   — makecom_g5, 1055
scsi_destroy_pkt — free an allocated SCSI packet and its DMA resource, 1387
SCSI dma utility routines
   — scsi_dmafree, 1390
   — scsi_dmaget, 1390
scsi_dname — decode SCSI peripheral device type, 1385
scsi_errmsg — display a SCSI request sense message, 1394
scsi_free_consistent_buf — free a previously allocated SCSI DMA I/O buffer, 1397
scsi_get_device_type_scsi_options — look up per-device-type scsi-options property, 1398
scsi_hba_attach — SCSI HBA attach and detach routines, 1403
SCSI HBA attach and detach routines
   — scsi_hba_attach, 1403
   — scsi_hba_attach_setup, 1403
   — scsi_hba_detach, 1403
scsi_hba_attach_setup — SCSI HBA attach and detach routines, 1403
scsi_hba_detach — SCSI HBA attach and detach routines, 1403
scsi_hba_fini — SCSI Host Bus Adapter system completion routines, 1410
scsi_hba_init — SCSI Host Bus Adapter system initialization routines, 1410
scsi_hba_lookup_capstr — return index matching capability string, 1411
scsi_hba_pkt_alloc — allocate and free a scsi_pkt structure, 1413
scsi_hba_pkt_free — allocate and free a scsi_pkt structure, 1413
scsi_hba_probe — default SCSI HBA probe function, 1417
scsi_hba_tran_alloc — allocate and free transport structures, 1418
scsi_hba_tran_free — allocate and free transport structures, 1418
scsi_ifgetcap — get SCSI transport capability, 1420
scsi_ifsetcap — set SCSI transport capability, 1420
scsi_init_pkt — prepare a complete SCSI packet, 1428
scsi_log — display a SCSI-device-related message, 1432
scsi_mname — decode SCSI messages, 1385
SCSI packet
   allocate a SCSI packet in iopb map —
      get_pktiopb, 967
   free a packet in iopb map —
      free_pktiopb, 967
   free an allocated SCSI packet and its DMA resource —
      scsi_destroy_pkt, 1387
SCSI packet utility routines
   — scsi_pktalloc, 1436
   — scsi_pktfree, 1436
   — scsi_resalloc, 1436
   — scsi_resfree, 1436
scsi_poll — run a polled SCSI command on behalf of a target driver, 1440
scsi_probe — utility for probing a scsi device, 1441
scsi_reset — reset a SCSI bus or target, 1445
scsi_reset_notify — notify target driver of bus resets, 1446
scsi_rname — decode SCSI packet completion reasons, 1385
scsi_setup_cdb — setup SCSI command descriptor block (CDB), 1451
scsi_slave — utility for SCSI target drivers to establish the presence of a target, 1452
scsi_sname — decode SCSI sense keys, 1385
scsi_sync_pkt — synchronize CPU and I/O views of memory, 1456
scsi_transport — request by a target driver to start a SCSI command, 1457
scsi_unprobe — free resources allocated during initial probing, 1458
scsi_unslave — free resources allocated during initial probing, 1458
scsi_vu_errmsg, display a SCSI request sense message, 1460
semaphore functions
— sema_destro, 1469
— sema_init, 1469
— sema_p, 1469
— sema_p_sig, 1469
— sema_tryp, 1469
— sema_v, 1469
— semaphore, 1469
send a signal to a process
— proc_ref, 1297
— proc_signal, 1297
— proc_unref, 1297
set client event mask
— csx_RequestSocketMask, 262
set client event mask — csx_SetEventMask, 270
set current access handle offset —
— csx_SetHandleOffset, 272
Set driver memory mapping parameters
— devmap_devmem_setup, 924
— devmap_umem_setup, 924
Set the device attributes for the mapping —
— ddi_mapdev_set_device_acc_attr, 597
set up a mapping for a register address space —
— ddi_regs_map_setup, 818
setup SCSI command descriptor block (CDB) —
— scsi_setup_cdb, 1451
single direction elevator seek sort for buffers —
— disksort, 939
size in bytes
— convert size in pages — ptob, 1301
— convert to size in memory pages (round down) — btop, 79
— convert to size in memory pages (round up) — btopr, 80
socket number, return physical socket number —
— csx_MapLogSocket, 140
soft interrupt handling
— add a soft interrupt — ddi_add_softintrl, 303
— get soft interrupt block cookie —
— ddi_get_soft_iblock_cookie, 303
— remove a soft interrupt —
— ddi_remove_softer, 303
— trigger a soft interrupt —
— ddi_trigger_softer, 303
sprintf — format characters in memory, 1489
status of PC Card and socket —
— csx_GetStatus, 132
STREAMS wait routines — qwait, qwait_sig, 1326
STREAMS ioctl blocks, allocate —
— mkiocb, 1070
STREAMS message blocks
— attach a user-supplied data buffer in place —
— esballoc, 951
— call a function when a buffer becomes available — bufcall, 81, 1566
— call a function when a buffer becomes available — qbufcall, 1316, 1324
— call function when buffer is available —
— esbcall, 953
— concatenate bytes in a message —
— msgpullup, 1077
— concatenate bytes in a message —
— pullupmsg, 1302
— concatenate two — linkb, 1054
— copy — copyb, 96
— erase the contents of a buffer — clrbuf, 87
— free all message blocks in a message —
— freemsg, 958
— free one — freeb, 957
— remove from head of message —
— unlinkb, 1568
— remove one form a message — rmvb, 1350
STREAMS message queue, insert a message into a queue —
— insq, 1003
STREAMS message queues, 59
STREAMS Message queues, get next message —
— getq, 969
STREAMS message queues
— reschedule a queue for service —
— enableok, 950
— test for room — canputnext, 86
— test for room — canput, 85
STREAMS messages
— copy a message — copymsg, 100
STREAMS messages (Continued)
flush for specified priority band — flushband, 954
remove form queue — flushq, 955
remove form queue — rmvq, 1351
return the number of bytes in a message — msgdsize, 1076
submit messages to the log driver — strlog, 1497
test whether a message is a data message — datamsg, 298
trim bytes — adjmsg, 50

STREAMS multithreading
— mt-streams, 1078
qbufcall — call a function when a buffer becomes available, 1316
qtimeout — execute a function after a specified length of time, 1323
qunbufcall — cancel a pending qbufcall request, 1324
quntimeout — cancel previous timeout function call, 1326
qwait, qwait_sig — STREAMS wait routines, 1326
qwriter — asynchronous STREAMS perimeter upgrade, 1330

STREAMS put and service procedures
disable — qprocsoff, 1319
enable — qprocson, 1319

STREAMS queues
change information about a queue or band of the queue — strqset, 1503
enable a queue — qenable, 1317
got pointer to queue’s partner queue — OTHERQ, 1226
got pointer to the read queue — RD, 1331
got information about a queue or band of the queue — strqget, 1502
number of messages on a queue — qsize, 1322
place a message at the head of a queue — putbq, 1305
prevent a queue from being scheduled — noenable, 1103
put a message on a queue — putq, 1313
send a control message to a queue — putctrl, 1307

STREAMS queues (Continued)
send a control message to a queue — putnextctl, 1311
send a control message with a one-byte parameter to a queue — putctl1, 1306
send a control message with a one-byte parameter to a queue — putnextctl1, 1310
send a message on a stream in the reverse direction — qreply, 1320
send a message to the next queue — putnext, 1309
test if next queue is in the same stream — SAMESTR, 1380
test for flow control in specified priority band — bcanput, 60

STREAMS write queues, get pointer for this module or driver — WR, 1589
STRUCTDECL, 32-bit application data access macros, 1510
swab — swap bytes in 16-bit halfwords, 1560
synchronize CPU and I/O views of memory — ddi_dma_sync, 465
synchronize CPU and I/O views of memory — scsi_sync_pkt, 1456

system DMA engine functions
— ddi_dmae, 402
— ddi_dmae_1stparty, 402
— ddi_dmae.alloc, 402
— ddi_dmae.disable, 402
— ddi_dmae.enable, 402
— ddi_dmae.getattr, 402
— ddi_dmae.getcnt, 402
— ddi_dmae.getlim, 402
— ddi_dmae.prog, 402
— ddi_dmae.release, 402
— ddi_dmae_stop, 402

system event, logging of — 587

T
testb — check for an available buffer, 1561
timeout — execute a function after a specified length of time, 1563
timeout, cancel previous timeout function call — untimout, 1569
trigger a soft interrupt — ddi_trigger_softintr, 303
tuple
first CIS tuple — csx_GetFirstTuple, 124
next CIS tuple — csx_GetNextTuple, 124
return data portion of tuple —
  csx_GetTupleData, 136
tuple entry
generic tuple parser — csx_ParseTuple, 206
parse Device Information tuple for Attribute Memory —
  csx_Parse_CISTPL_DEVICE_A, 159
parse Device Information tuple for Common Memory —
  csx_Parse_CISTPLDEVICE, 159
parse JEDEC Identifier tuple for Attribute Memory —
  csx_Parse_CISTPL_JEDEC_A, 190
parse JEDEC Identifier tuple for Common Memory —
  csx_Parse_CISTPL_JEDEC_C, 190
parse Long Link A tuple —
  csx_Parse_CISTPL_LONGLINK_A, 193
parse Long Link C tuple —
  csx_Parse_CISTPL_LONGLINK_C, 193
parse Other Condition Device Information tuple for Attribute Memory —
  csx_Parse_CISTPLDEVICE_OA, 159
parse Other Condition Device Information tuple for Common Memory —
  csx_Parse_CISTPLDEVICE_OC, 159
parse Battery Replacement Date tuple —
  csx_Parse_CISTPL_BATTERY, 146
parse Byte Order tuple —
  csx_Parse_CISTPL_BYTEORDER, 147
parse Card Configuration Table tuple —
  csx_Parse_CISTPL_CFTABLE_ENTRY, 149
parse Card Initialization Date tuple —
  csx_Parse_CISTPL_DATE, 158
parse Configuration tuple —
  csx_Parse_CISTPLCONFIG, 155
parse Data Organization tuple —
  csx_Parse_CISTPLORG, 200
parse Data Recording Format tuple —
  csx_Parse_CISTPL_FORMAT, 175
parse Device Geo A tuple —
  csx_Parse_CISTPLDEVICE_A, 167
parse Device Geo tuple —
  csx_Parse_CISTPLDEVICEGEO, 165
tuple entry (Continued)
parse Function Extension tuple —
  csx_Parse_CISTPL_FUNCE, 177
parse Function Identification tuple —
  csx_Parse_CISTPL_FUNCID, 185
parse Geometry tuple —
  csx_Parse_CISTPL_GEOMETRY, 187
parse Level-1 Version/Product Information tuple —
  csx_Parse_CISTPL_VERS_1, 204
parse Level-2 Version and Information tuple —
  csx_Parse_CISTPL VERS_2, 205
parse Link Target tuple —
  csx_Parse_CISTPL LINKTARGET, 192
parse Manufacturer Identification tuple —
  csx_Parse_CISTPL_MANFID, 199
parse Multi-Function tuple —
  csx_Parse_CISTPL_LONGLINK_MFC, 197
parse Software Interleaving tuple —
  csx_Parse_CISTPL_SWIL, 203
parse Special Purpose tuple —
  csx_Parse_CISTPL_SPCL, 201

U
ui0 structure
  add character — ureadc, 1571
  remove a character — uwritec, 1572
uiomove — copy kernel data using ui0 structure, 1565
unbinds the address in a DMA handle —
  ddi_dma_unbind_handle, 467
unfreezestr — freeze, thaw the state of a stream, 962
uninitialize a buffer structure — biofini, 71
update I/O kstat statistics
  — kstat_queue, 1040
  — kstat_runq_back_to_waitq, 1040
  — kstat_runq_enter, 1040
  — kstat_runq_exit, 1040
  — kstat_waitq_enter, 1040
  — kstat_waitq_exit, 1040
  — kstat_waitq_to_runq, 1040
update property information.
  — ddi_prop_update, 769
  — ddi_prop_update_byte_array, 769
  — ddi_prop_update_int, 769
  — ddi_prop_update_int_array, 769
update property information. (Continued)
— ddi_prop_update_string, 769
— ddi_prop_update_string_array, 769

V
va_arg — handle variable argument list, 1573
va_copy — handle variable argument list, 1573
va_end — handle variable argument list, 1573
validate Card Information Structure (CIS) —
    csx.ValidateCIS, 273
virtual address, return mapped virtual address
    — csx.GetMappedAddr, 127
vsprintf — format characters in memory, 1586

W
window resources
    release window resources —
        csx_ReleaseWindow, 264
    request window resources —
        csx_RequestWindow, 264
write data to mapped device in the memory
    space or allocated DMA memory
    — ddi_mem_put16, 619
    — ddi_mem_put32, 619
    — ddi_mem_put64, 619
    — ddi_mem_put8, 619
    — ddi_mem_rep_put16, 651
    — ddi_mem_rep_put32, 651
    — ddi_mem_rep_put64, 651
    — ddi_mem_rep_put8, 651
write data to the device
    — ddi_put16, 807
    — ddi_put32, 807
    — ddi_put64, 807
    — ddi_put8, 807
write data to the mapped device register in I/O
    space
    — ddi_io_put16, 555
    — ddi_io_put32, 555
    — ddi_io_put8, 555
    — ddi_io_putb, 555
    — ddi_io_putl, 555
    — ddi_io_putw, 555
write data to the mapped memory address,
device register or allocated DMA memory
    address
    — ddi_rep_put16, 856
    — ddi_rep_put32, 856
    — ddi_rep_put64, 856
    — ddi_rep_put8, 856
    — ddi_rep_putb, 856
    — ddi_rep_putl, 856
    — ddi_rep_putw, 856
write multiple data to the mapped device
    register in I/O space
    — ddi_io_rep_put16, 579
    — ddi_io_rep_put32, 579
    — ddi_io_rep_put8, 579
    — ddi_io_rep_putb, 579
    — ddi_io_rep_putl, 579
    — ddi_io_rep_putw, 579
write or read a PC Card Configuration Register
    — csx_AccessConfigurationRegister, 104
write repetitively to device register
    — csx_RepPut16, 248
    — csx_RepPut32, 248
    — csx_RepPut64, 248
    — csx_RepPut8, 248
write to an I/O port
    — outb, 1228
    — outl, 1228
    — outw, 1228
    — repoutsb, 1228
    — repoutsd, 1228
    — repoutsw, 1228
write to device register
    — csx_Put16, 211
    — csx_Put32, 211
    — csx_Put64, 211
    — csx_Put8, 211
zero fill the device register —
    ddi_device_zero, 327