man pages section 9: DDI and DKI Kernel Functions

Part No: 816–5180–16
September 2010

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

100617@24378

Contents

Preface

Both novice users and those familar 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 secton 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.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	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.
BUGS	This section describes known bugs and, wherever possible, suggests workarounds.

REFERENCE

Introduction

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 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.

  Note that a mutex acquired in user or kernel context that can also be acquired in interrupt context means that the user or kernel context thread holding that mutex is subject to all the restrictions imposed by interrupt context, for the duration of the ownership of that mutex. Please see the `mutex`(9F) man page for a more complete discussion of proper mutex handling for drivers.
- *High-level interrupt context* is a more restricted form of interrupt context. If a driver interrupt priority returned from `ddi_intr_get_pri`(9F) is greater than the priority returned from `ddi_intr_get_hilevel_pri`(9F) this indicates the interrupt handler will run in high-level interrupt context. These interrupt routines are only allowed to call `ddi_intr_trigger_softint`(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 interrupt priority returned by `ddi_intr_get_pri`(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.

Streams Kernel Function Summary

The following table summarizes the STREAMS functions described in this section.

Routine	Type
`adjmsg`	DDI/DKI
`allocb`	DDI/DKI
`allocb_tmpl`	Solaris DDI
`backq`	DDI/DKI
`bcanput`	DDI/DKI
`bcanputnext`	DDI/DKI
`bufcall`	DDI/DKI
`canput`	DDI/DKI
`canputnext`	DDI/DKI
`clrbuf`	DDI/DKI
`copyb`	DDI/DKI
`copymsg`	DDI/DKI
`DB_BASE`	Solaris DDI
`DB_LIM`	Solaris DDI
`DB_REF`	Solaris DDI
`DB_TYPE`	Solaris DDI
`datamsg`	DDI/DKI
`dupb`	DDI/DKI
`dupmsg`	DDI/DKI
`enableok`	DDI/DKI
`esballoc`	DDI/DKI
`esbbcall`	DDI/DKI
`flushband`	DDI/DKI
`flushq`	DDI/DKI
`freeb`	DDI/DKI

Routine	Type
`freemsg`	DDI/DKI
`freezestr`	DDI/DKI
`getq`	DDI/DKI
`IOC_CONVER_FROM`	Solaris DDI
`insq`	DDI/DKI
`linkb`	DDI/DKI
`MBLKHEAD`	Solaris DDI
`MBLKIN`	Solaris DDI
`MBLKL`	Solaris DDI
`MBLKSIZE`	Solaris DDI
`MBLKTAIL`	Solaris DDI
`mcopyin`	Solaris DDI
`mcopymsg`	Solaris DDI
`mcopyout`	Solaris DDI
`merror`	Solaris DDI
`mexchange`	Solaris DDI
`mioc2ack`	Solaris DDI
`miocack`	Solaris DDI
`mexchange`	Solaris DDI
`miocpullup`	Solaris DDI
`mkiocb`	Solaris DDI
`msgdsize`	DDI/DKI
`msgpullup`	DDI/DKI
`msgsize`	Solaris DDI
`mt-streams`	Solaris DDI
`noenable`	DDI/DKI
`OTHERQ`	DDI/DKI
`pullupmsg`	DDI/DKI

Routine	Type
`put`	DDI/DKI
`putbq`	DDI/DKI
`putctl`	DDI/DKI
`putctl1`	DDI/DKI
`putnext`	DDI/DKI
`putnextctl`	DDI/DKI
`putq`	DDI/DKI
`qassociate`	Solaris DDI
`qbufcall`	Solaris DDI
`qenable`	DDI/DKI
`qprocson`	DDI/DKI
`qprocsoff`	DDI/DKI
`qreply`	DDI/DKI
`qsize`	DDI/DKI
`qtimeout`	Solaris DDI
`qunbufcall`	Solaris DDI
`quntimeout`	Solaris DDI
`qwait`	Solaris DDI
`qwait_sig`	Solaris DDI
`qwriter`	Solaris DDI
`RD`	DDI/DKI
`rmvb`	DDI/DKI
`rmvq`	DDI/DKI
`SAMESTR`	DDI/DKI
`strlog`	DDI/DKI
`strqget`	DDI/DKI
`strqset`	DDI/DKI
`testb`	DDI/DKI

Routine	Type
`unbufcall`	DDI/DKI
`unfreezestr`	DDI/DKI
`unlinkb`	DDI/DKI
`WR`	DDI/DKI

The following table summarizes the functions not specific to STREAMS.

Routine	Type
`ASSERT`	DDI/DKI
`anocancel`	Solaris DDI
`aphysio`	Solaris DDI
`atomic_add`	DDI/DKI
`atomic_and`	DDI/DKI
`atomic_bits`	DDI/DKI
`atomic_cas`	DDI/DKI
`atomic_dec`	DDI/DKI
`atomic_inc`	DDI/DKI
`atomic_ops`	DDI/DKI
`atomic_or`	DDI/DKI
`atomic_swap`	DDI/DKI
`bcmp`	DDI/DKI
`bcopy`	DDI/DKI
`bioclone`	Solaris DDI
`biodone`	DDI/DKI
`biofini`	Solaris DDI
`bioinit`	Solaris DDI
`biomodified`	Solaris DDI
`biosize`	Solaris DDI
`bioerror`	Solaris DDI

Routine	Type
`bioreset`	Solaris DDI
`biowait`	DDI/DKI
`bp_copyin`	DDI/DKI
`bp_copyout`	DDI/DKI
`bp_mapin`	DDI/DKI
`bp_mapout`	DDI/DKI
`btop`	DDI/DKI
`btopr`	DDI/DKI
`bzero`	DDI/DKI
`cmn_err`	DDI/DKI
`condvar`	Solaris DDI
`copyin`	DDI/DKI
`copyout`	DDI/DKI
`csx_AccessConfigurationRegister`	Solaris DDI
`csx_ConvertSize`	Solaris DDI
`csx_ConvertSpeed`	Solaris DDI
`csx_CS_DDI_Info`	Solaris DDI
`csx_DeregisterClient`	Solaris DDI
`csx_DupHandle`	Solaris DDI
`csx_Error2Text`	Solaris DDI
`csx_Event2Text`	Solaris DDI
`csx_FreeHandle`	Solaris DDI
`csx_Get8`	Solaris DDI
`csx_GetFirstClient`	Solaris DDI
`csx_GetFirstTuple`	Solaris DDI
`csx_GetHandleOffset`	Solaris DDI
`csx_GetMappedAddr`	Solaris DDI
`csx_GetStatus`	Solaris DDI

Routine	Type
`csx_GetTupleData`	Solaris DDI
`csx_MakeDeviceNode`	Solaris DDI
`csx_MapLogSocket`	Solaris DDI
`csx_MapMemPage`	Solaris DDI
`csx_ModifyConfiguration`	Solaris DDI
`csx_ModifyWindow`	Solaris DDI
`csx_Parse_CISTPL_BATTERY`	Solaris DDI
`csx_Parse_CISTPL_BYTEORDER`	Solaris DDI
`csx_Parse_CISTPL_CFTABLE_ENTRY`	Solaris DDI
`csx_Parse_CISTPL_CONFIG`	Solaris DDI
`csx_Parse_CISTPL_DATE`	Solaris DDI
`csx_Parse_CISTPL_DEVICE`	Solaris DDI
`csx_Parse_CISTPL_DEVICEGEO`	Solaris DDI
`csx_Parse_CISTPL_DEVICEGEO_A`	Solaris DDI
`csx_Parse_CISTPL_FORMAT`	Solaris DDI
`csx_Parse_CISTPL_FUNCE`	Solaris DDI
`csx_Parse_CISTPL_FUNCID`	Solaris DDI
`csx_Parse_CISTPL_GEOMETRY`	Solaris DDI
`csx_Parse_CISTPL_JEDEC_C`	Solaris DDI
`csx_Parse_CISTPL_LINKTARGET`	Solaris DDI
`csx_Parse_CISTPL_LONGLINK_A`	Solaris DDI
`csx_Parse_CISTPL_LONGLINK_MFC`	Solaris DDI
`csx_Parse_CISTPL_MANFID`	Solaris DDI
`csx_Parse_CISTPL_ORG`	Solaris DDI
`csx_Parse_CISTPL_SPCL`	Solaris DDI
`csx_Parse_CISTPL_SWIL`	Solaris DDI
`csx_Parse_CISTPL_VERS_1`	Solaris DDI
`csx_Parse_CISTPL_VERS_2`	Solaris DDI

Routine	Type
`csx_ParseTuple`	Solaris DDI
`csx_Put8`	Solaris DDI
`csx_RegisterClient`	Solaris DDI
`csx_ReleaseConfiguration`	Solaris DDI
`csx_RepGet8`	Solaris DDI
`csx_RepPut8`	Solaris DDI
`csx_RequestConfiguration`	Solaris DDI
`csx_RequestIO`	Solaris DDI
`csx_RequestIRQ`	Solaris DDI
`csx_RequestSocketMask`	Solaris DDI
`csx_RequestWindow`	Solaris DDI
`csx_ResetFunction`	Solaris DDI
`csx_SetEventMask`	Solaris DDI
`csx_SetHandleOffset`	Solaris DDI
`csx_ValidateCIS`	Solaris DDI
`cv_broadcast`	Solaris DDI
`cv_destroy`	Solaris DDI
`cv_init`	Solaris DDI
`cv_signal`	Solaris DDI
`cv_timedwait`	Solaris DDI
`cv_wait`	Solaris DDI
`cv_wait_sig`	Solaris DDI
`ddi_add_event_handler`	Solaris DDI
`ddi_add_intr`	Solaris DDI
`ddi_add_softintr`	Solaris DDI
`ddi_binding_name`	Solaris DDI
`ddi_btop`	Solaris DDI
`ddi_btopr`	Solaris DDI

Routine	Type
`ddi_can_receive_sig`	Solaris DDI
`ddi_check_acc_handle`	Solaris DDI
`ddi_copyin`	Solaris DDI
`ddi_copyout`	Solaris DDI
`ddi_create_minor_node`	Solaris DDI
`ddi_cred`	Solaris DDI
`ddi_dev_is_sid`	Solaris DDI
`ddi_dev_nintrs`	Solaris DDI
`ddi_dev_nregs`	Solaris DDI
`ddi_dev_regsize`	Solaris DDI
`ddi_device_copy`	Solaris DDI
`ddi_device_zero`	Solaris DDI
`ddi_devmap_segmap`	Solaris DDI
`ddi_dma_addr_bind_handle`	Solaris DDI
`ddi_dma_addr_setup`	Solaris DDI
`ddi_dma_alloc_handle`	Solaris DDI
`ddi_dma_buf_bind_handle`	Solaris DDI
`ddi_dma_buf_setup`	Solaris DDI
`ddi_dma_burstsizes`	Solaris DDI
`ddi_dma_coff`	Solaris SPARC DDI
`ddi_dma_curwin`	Solaris SPARC DDI
`ddi_dma_devalign`	Solaris DDI
`ddi_dma_free`	Solaris DDI
`ddi_dma_free_handle`	Solaris DDI
`ddi_dma_getwin`	Solaris DDI
`ddi_dma_get_attr`	Solaris DDI
`ddi_dma_htoc`	Solaris SPARC DDI
`ddi_dma_mem_alloc`	Solaris DDI

Routine	Type
`ddi_dma_mem_free`	Solaris DDI
`ddi_dma_movwin`	Solaris SPARC DDI
`ddi_dma_nextcookie`	Solaris DDI
`ddi_dma_nextseg`	Solaris DDI
`ddi_dma_nextwin`	Solaris DDI
`ddi_dma_numwin`	Solaris DDI
`ddi_dma_segtocookie`	Solaris DDI
`ddi_dma_set_sbus64`	Solaris DDI
`ddi_dma_setup`	Solaris DDI
`ddi_dma_sync`	Solaris DDI
`ddi_dma_unbind_handle`	Solaris DDI
`ddi_dmae`	Solaris x86 DDI
`ddi_dmae_1stparty`	Solaris x86 DDI
`ddi_dmae_alloc`	Solaris x86 DDI
`ddi_dmae_disable`	Solaris x86 DDI
`ddi_dmae_enable`	Solaris x86 DDI
`ddi_dmae_getattr`	Solaris x86 DDI
`ddi_dmae_getcnt`	Solaris x86 DDI
`ddi_dmae_getlim`	Solaris x86 DDI
`ddi_dmae_prog`	Solaris x86 DDI
`ddi_dmae_release`	Solaris x86 DDI
`ddi_dmae_stop`	Solaris x86 DDI
`ddi_driver_major`	Solaris DDI
`ddi_driver_name`	Solaris DDI
`ddi_enter_critical`	Solaris DDI
`ddi_exit_critical`	Solaris DDI
`ddi_ffs`	Solaris DDI
`ddi_fls`	Solaris DDI

Routine	Type
`ddi_fm_acc_err_clear`	Solaris DDI
`ddi_fm_acc_err_get`	Solaris DDI
`ddi_fm_ereport_post`	Solaris DDI
`ddi_fm_handler_register`	Solaris DDI
`ddi_fm_init`	Solaris DDI
`ddi_fm_service_impact`	Solaris DDI
`ddi_get16`	Solaris DDI
`ddi_get32`	Solaris DDI
`ddi_get64`	Solaris DDI
`ddi_get8`	Solaris DDI
`ddi_get_cred`	Solaris DDI
`ddi_get_devstate`	Solaris DDI
`ddi_get_driver_private`	Solaris DDI
`ddi_get_eventcookie`	Solaris DDI
`ddi_get_iblock_cookie`	Solaris DDI
`ddi_get_iminor`	Solaris DDI
`ddi_get_instance`	Solaris DDI
`ddi_get_kt_did`	Solaris DDI
`ddi_get_lbolt`	Solaris DDI
`ddi_get_name`	Solaris DDI
`ddi_get_parent`	Solaris DDI
`ddi_get_pid`	Solaris DDI
`ddi_get_soft_iblock_cookie`	Solaris DDI
`ddi_get_soft_state`	Solaris DDI
`ddi_getb`	Solaris DDI
`ddi_getl`	Solaris DDI
`ddi_getll`	Solaris DDI
`ddi_getlongprop`	Solaris DDI

Routine	Type
`ddi_getlongprop_buf`	Solaris DDI
`ddi_getprop`	Solaris DDI
`ddi_getproplen`	Solaris DDI
`ddi_getw`	Solaris DDI
`ddi_intr_add_handler`	Solaris DDI
`ddi_intr_add_softint`	Solaris DDI
`ddi_intr_alloc`	Solaris DDI
`ddi_intr_block_disable`	Solaris DDI
`ddi_intr_block_enable`	Solaris DDI
`ddi_intr_clr_mask`	Solaris DDI
`ddi_intr_dup_handler`	Solaris DDI
`ddi_intr_disable`	Solaris DDI
`ddi_intr_enable`	Solaris DDI
`ddi_intr_free`	Solaris DDI
`ddi_intr_get_cap`	Solaris DDI
`ddi_intr_get_hilevel_pri`	Solaris DDI
`ddi_intr_get_navail`	Solaris DDI
`ddi_intr_get_nintrs`	Solaris DDI
`ddi_intr_get_pending`	Solaris DDI
`ddi_intr_get_pri`	Solaris DDI
`ddi_intr_get_softint_pri`	Solaris DDI
`ddi_intr_get_supported_types`	Solaris DDI
`ddi_intr_remove_handler`	Solaris DDI
`ddi_intr_remove_softint`	Solaris DDI
`ddi_intr_set_cap`	Solaris DDI
`ddi_intr_set_mask`	Solaris DDI
`ddi_intr_set_pri`	Solaris DDI
`ddi_intr_set_softint_pri`	Solaris DDI

Routine	Type
`ddi_intr_trigger_softint`	Solaris DDI
`ddi_io_get16`	Solaris DDI
`ddi_io_get32`	Solaris DDI
`ddi_io_get8`	Solaris DDI
`ddi_io_getb`	Solaris DDI
`ddi_io_getl`	Solaris DDI
`ddi_io_getw`	Solaris DDI
`ddi_io_put16`	Solaris DDI
`ddi_io_put32`	Solaris DDI
`ddi_io_put8`	Solaris DDI
`ddi_io_putb`	Solaris DDI
`ddi_io_putl`	Solaris DDI
`ddi_io_putw`	Solaris DDI
`ddi_io_rep_get16`	Solaris DDI
`ddi_io_rep_get32`	Solaris DDI
`ddi_io_rep_get8`	Solaris DDI
`ddi_io_rep_getb`	Solaris DDI
`ddi_io_rep_getl`	Solaris DDI
`ddi_io_rep_getw`	Solaris DDI
`ddi_io_rep_put16`	Solaris DDI
`ddi_io_rep_put32`	Solaris DDI
`ddi_io_rep_put8`	Solaris DDI
`ddi_io_rep_putb`	Solaris DDI
`ddi_io_rep_putl`	Solaris DDI
`ddi_io_rep_putw`	Solaris DDI
`ddi_iomin`	Solaris DDI
`ddi_iopb_alloc`	Solaris DDI
`ddi_iopb_free`	Solaris DDI

Routine	Type
`ddi_log_sysevent`	Solaris DDI
`ddi_map_regs`	Solaris DDI
`ddi_mapdev`	Solaris DDI
`ddi_mapdev_intercept`	Solaris DDI
`ddi_mapdev_nointercept`	Solaris DDI
`ddi_mapdev_set_device_acc_attr`	Solaris DDI
`ddi_mem_alloc`	Solaris DDI
`ddi_mem_free`	Solaris DDI
`ddi_mem_get16`	Solaris DDI
`ddi_mem_get32`	Solaris DDI
`ddi_mem_get64`	Solaris DDI
`ddi_mem_get8`	Solaris DDI
`ddi_mem_getb`	Solaris DDI
`ddi_mem_getl`	Solaris DDI
`ddi_mem_getll`	Solaris DDI
`ddi_mem_getw`	Solaris DDI
`ddi_mem_put16`	Solaris DDI
`ddi_mem_put32`	Solaris DDI
`ddi_mem_put64`	Solaris DDI
`ddi_mem_put8`	Solaris DDI
`ddi_mem_putb`	Solaris DDI
`ddi_mem_putl`	Solaris DDI
`ddi_mem_putll`	Solaris DDI
`ddi_mem_putw`	Solaris DDI
`ddi_mem_rep_get16`	Solaris DDI
`ddi_mem_rep_get32`	Solaris DDI
`ddi_mem_rep_get64`	Solaris DDI
`ddi_mem_rep_get8`	Solaris DDI

Routine	Type
`ddi_mem_rep_getb`	Solaris DDI
`ddi_mem_rep_getl`	Solaris DDI
`ddi_mem_rep_getll`	Solaris DDI
`ddi_mem_rep_getw`	Solaris DDI
`ddi_mem_rep_put16`	Solaris DDI
`ddi_mem_rep_put32`	Solaris DDI
`ddi_mem_rep_put64`	Solaris DDI
`ddi_mem_rep_put8`	Solaris DDI
`ddi_mem_rep_putb`	Solaris DDI
`ddi_mem_rep_putl`	Solaris DDI
`ddi_mem_rep_putll`	Solaris DDI
`ddi_mem_rep_putw`	Solaris DDI
`ddi_mmap_get_model`	Solaris DDI
`ddi_model_convert_from`	Solaris DDI
`ddi_modopen`	Solaris DDI
`ddi_no_info`	Solaris DDI
`ddi_node_name`	Solaris DDI
`ddi_peek16`	Solaris DDI
`ddi_peek32`	Solaris DDI
`ddi_peek64`	Solaris DDI
`ddi_peek8`	Solaris DDI
`ddi_peekc`	Solaris DDI
`ddi_peekd`	Solaris DDI
`ddi_peekl`	Solaris DDI
`ddi_peeks`	Solaris DDI
`ddi_periodic_add`	Solaris DDI
`ddi_periodic_delete`	Solaris DDI
`ddi_poke16`	Solaris DDI

Routine	Type
`ddi_poke32`	Solaris DDI
`ddi_poke64`	Solaris DDI
`ddi_poke8`	Solaris DDI
`ddi_pokec`	Solaris DDI
`ddi_poked`	Solaris DDI
`ddi_pokel`	Solaris DDI
`ddi_pokes`	Solaris DDI
`ddi_prop_create`	Solaris DDI
`ddi_prop_exists`	Solaris DDI
`ddi_prop_free`	Solaris DDI
`ddi_prop_get_int`	Solaris DDI
`ddi_prop_lookup`	Solaris DDI
`ddi_prop_lookup_byte_array`	Solaris DDI
`ddi_prop_lookup_int_array`	Solaris DDI
`ddi_prop_lookup_string`	Solaris DDI
`ddi_prop_lookup_string_array`	Solaris DDI
`ddi_prop_modify`	Solaris DDI
`ddi_prop_op`	Solaris DDI
`ddi_prop_remove`	Solaris DDI
`ddi_prop_remove_all`	Solaris DDI
`ddi_prop_undefine`	Solaris DDI
`ddi_prop_update`	Solaris DDI
`ddi_prop_update_byte_array`	Solaris DDI
`ddi_prop_update_int`	Solaris DDI
`ddi_prop_update_int_array`	Solaris DDI
`ddi_prop_update_string`	Solaris DDI
`ddi_prop_update_string_array`	Solaris DDI
`ddi_ptob`	Solaris DDI

Routine	Type
`ddi_put16`	Solaris DDI
`ddi_put32`	Solaris DDI
`ddi_put64`	Solaris DDI
`ddi_put8`	Solaris DDI
`ddi_putb`	Solaris DDI
`ddi_putl`	Solaris DDI
`ddi_putll`	Solaris DDI
`ddi_putw`	Solaris DDI
`ddi_regs_map_free`	Solaris DDI
`ddi_regs_map_setup`	Solaris DDI
`ddi_remove_event_handler`	Solaris DDI
`ddi_remove_intr`	Solaris DDI
`ddi_remove_minor_node`	Solaris DDI
`ddi_remove_softintr`	Solaris DDI
`ddi_removing_power`	Solaris DDI
`ddi_rep_get16`	Solaris DDI
`ddi_rep_get32`	Solaris DDI
`ddi_rep_get64`	Solaris DDI
`ddi_rep_get8`	Solaris DDI
`ddi_rep_getb`	Solaris DDI
`ddi_rep_getl`	Solaris DDI
`ddi_rep_getll`	Solaris DDI
`ddi_rep_getw`	Solaris DDI
`ddi_rep_put16`	Solaris DDI
`ddi_rep_put32`	Solaris DDI
`ddi_rep_put64`	Solaris DDI
`ddi_rep_put8`	Solaris DDI
`ddi_rep_putb`	Solaris DDI

Routine	Type
`ddi_rep_putl`	Solaris DDI
`ddi_rep_putll`	Solaris DDI
`ddi_rep_putw`	Solaris DDI
`ddi_report_dev`	Solaris DDI
`ddi_root_node`	Solaris DDI
`ddi_segmap`	Solaris DDI
`ddi_segmap_setup`	Solaris DDI
`ddi_set_driver_private`	Solaris DDI
`ddi_slaveonly`	Solaris DDI
`ddi_soft_state`	Solaris DDI
`ddi_soft_state_fini`	Solaris DDI
`ddi_soft_state_free`	Solaris DDI
`ddi_soft_state_init`	Solaris DDI
`ddi_soft_state_zalloc`	Solaris DDI
`ddi_strlol`	Solaris DDI
`ddi_strloul`	Solaris DDI
`ddi_trigger_softintr`	Solaris DDI
`ddi_umem_alloc`	Solaris DDI
`ddi_umem_free`	Solaris DDI
`ddi_umem_iosetup`	Solaris DDI
`ddi_umem_lock`	Solaris DDI
`ddi_unmap_regs`	Solaris DDI
`delay`	DDI/DKI
`devmap_default_access`	Solaris DDI
`devmap_devmem_setup`	Solaris DDI
`devmap_do_ctxmgt`	Solaris DDI
`devmap_load`	Solaris DDI
`devmap_set_ctx_timeout`	Solaris DDI

Routine	Type
`devmap_setup`	Solaris DDI
`devmap_umem_setup`	Solaris DDI
`devmap_unload`	Solaris DDI
`disksort`	Solaris DDI
`dlbindack`	Solaris DDI
`drv_getparm`	DDI/DKI
`drv_hztousec`	DDI/DKI
`drv_priv`	DDI/DKI
`drv_usectohz`	DDI/DKI
`drv_usecwait`	DDI/DKI
`free_pktiopb`	Solaris DDI
`freerbuf`	DDI/DKI
`get_pktiopb`	Solaris DDI
`geterror`	DDI/DKI
`gethrtime`	DDI/DKI
`getmajor`	DDI/DKI
`getminor`	DDI/DKI
`getrbuf`	DDI/DKI
`gld`	Solaris DDI
`hat_getkpfnum`	DKI only
`id32_alloc`	Solaris DDI
`inb`	Solaris x86 DDI
`inl`	Solaris x86 DDI
`inw`	Solaris x86 DDI
`kiconv`	Solaris DDI
`kiconv_close`	Solaris DDI
`kiconv_open`	Solaris DDI
`kiconvstr`	Solaris DDI

Routine	Type
`kmem_alloc`	DDI/DKI
`kmem_cache_create`	Solaris DDI
`kmem_free`	DDI/DKI
`kmem_zalloc`	DDI/DKI
`kstat_create`	Solaris DDI
`kstat_delete`	Solaris DDI
`kstat_install`	Solaris DDI
`kstat_named_init`	Solaris DDI
`kstat_queue`	Solaris DDI
`kstat_runq_back_to_waitq`	Solaris DDI
`kstat_runq_enter`	Solaris DDI
`kstat_runq_exit`	Solaris DDI
`kstat_waitq_enter`	Solaris DDI
`kstat_waitq_exit`	Solaris DDI
`kstat_waitq_to_runq`	Solaris DDI
`ldi_add_event_handler`	Solaris DDI
`ldi_aread`	Solaris DDI
`ldi_devmap`	Solaris DDI
`ldi_dump`	Solaris DDI
`ldi_ev_finalize`	Solaris DDI
`ldi_ev_get_cookie`	Solaris DDI
`ldi_ev_get_type`	Solaris DDI
`ldi_ev_notify`	Solaris DDI
`ldi_ev_register_callbacks`	Solaris DDI
`ldi_ev_remove_callbacks`	Solaris DDI
`ldi_get_dev`	Solaris DDI
`ldi_get_eventcookie`	Solaris DDI
`ldi_get_size`	Solaris DDI

Routine	**Type**
`ldi_ident_from_dev`	Solaris DDI
`ldi_ioctl`	Solaris DDI
`ldi_open_by_dev`	Solaris DDI
`ldi_poll`	Solaris DDI
`ldi_prop_exists`	Solaris DDI
`ldi_prop_get_int`	Solaris DDI
`ldi_prop_get_lookup_int_array`	Solaris DDI
`ldi_putmsg`	Solaris DDI
`ldi_read`	Solaris DDI
`ldi_remove_event_handler`	Solaris DDI
`ldi_strategy`	Solaris DDI
`makecom_g0`	Solaris DDI
`makecom_g0_s`	Solaris DDI
`makecom_g1`	Solaris DDI
`makecom_g5`	Solaris DDI
`makedevice`	DDI/DKI
`max`	DDI/DKI
`max`	DDI/DKI
`membar_ops`	Solaris DDI
`memchr`	Solaris DDI
`minphys`	Solaris DDI
`mod_info`	Solaris DDI
`mod_install`	Solaris DDI
`mod_remove`	Solaris DDI
`mutex_destroy`	Solaris DDI
`mutex_enter`	Solaris DDI
`mutex_exit`	Solaris DDI
`mutex_init`	Solaris DDI

Routine	Type
`mutex_owned`	Solaris DDI
`mutex_tryenter`	Solaris DDI
`nochpoll`	Solaris DDI
`nodev`	DDI/DKI
`nulldev`	DDI/DKI
`numtos`	Solaris DDI
`nvlist_add_boolean`	Solaris DDI
`nvlist_alloc`	Solaris DDI
`nvlist_lookup_boolean`	Solaris DDI
`nvlist_lookup_nvpair`	Solaris DDI
`nvlist_next_nvpair`	Solaris DDI
`nvlist_remove`	Solaris DDI
`nvlist_value_byte`	Solaris DDI
`outb`	Solaris x86 DDI
`outl`	Solaris x86 DDI
`outw`	Solaris x86 DDI
`pci_config_get16`	Solaris DDI
`pci_config_get32`	Solaris DDI
`pci_config_get64`	Solaris DDI
`pci_config_get8`	Solaris DDI
`pci_config_getb`	Solaris DDI
`pci_config_getl`	Solaris DDI
`pci_config_getw`	Solaris DDI
`pci_config_put16`	Solaris DDI
`pci_config_put32`	Solaris DDI
`pci_config_put64`	Solaris DDI
`pci_config_put8`	Solaris DDI
`pci_config_putb`	Solaris DDI

Routine	Type
`pci_config_putl`	Solaris DDI
`pci_config_putw`	Solaris DDI
`pci_config_setup`	Solaris DDI
`pci_config_teardown`	Solaris DDI
`pci_ereport_setup`	Solaris DDI
`pci_report_pmcap`	Solaris DDI
`pci_save_config_regs`	Solaris DDI
`physio`	Solaris DDI
`pm_busy_component`	Solaris DDI
`pm_power_has_changed`	Solaris DDI
`pm_raise_power`	Solaris DDI
`pm_trans_check`	Solaris DDI
`pollwakeup`	DDI/DKI
`pci_config_teardown`	Solaris DDI
`pci_config_teardown`	Solaris DDI
`priv_getbyname`	Solaris DDI
`priv_policy`	Solaris DDI
`proc_signal`	Solaris DDI
`proc_unref`	Solaris DDI
`ptob`	DDI/DKI
`repinsb`	Solaris x86 DDI
`repinsd`	Solaris x86 DDI
`repinsw`	Solaris x86 DDI
`repoutsb`	Solaris x86 DDI
`repoutsd`	Solaris x86 DDI
`repoutsw`	Solaris x86 DDI
`rmalloc`	DDI/DKI
`rmalloc_wait`	DDI/DKI

Routine	Type
`rmallocmap`	DDI/DKI
`rmallocmap_wait`	DDI/DKI
`rmfree`	DDI/DKI
`rmfreemap`	DDI/DKI
`rw_destroy`	Solaris DDI
`rw_downgrade`	Solaris DDI
`rw_enter`	Solaris DDI
`rw_exit`	Solaris DDI
`rw_init`	Solaris DDI
`rw_read_locked`	Solaris DDI
`rw_tryenter`	Solaris DDI
`rw_tryupgrade`	Solaris DDI
`scsi_abort`	Solaris DDI
`scsi_alloc_consistent_buf`	Solaris DDI
`scsi_cname`	Solaris DDI
`scsi_destroy_pkt`	Solaris DDI
`scsi_dmafree`	Solaris DDI
`scsi_dmaget`	Solaris DDI
`scsi_dname`	Solaris DDI
`scsi_errmsg`	Solaris DDI
`scsi_ext_sense_fields`	Solaris DDI
`scsi_find_sense_descr`	Solaris DDI
`scsi_free_consistent_buf`	Solaris DDI
`scsi_get_device_type_scsi_options`	Solaris DDI
`scsi_get_device_type_string`	Solaris DDI
`scsi_hba_attach`	Solaris DDI
`scsi_hba_attach_setup`	Solaris DDI
`scsi_hba_detach`	Solaris DDI

Routine	Type
`scsi_hba_fini`	Solaris DDI
`scsi_hba_init`	Solaris DDI
`scsi_hba_lookup_capstr`	Solaris DDI
`scsi_hba_pkt_alloc`	Solaris DDI
`scsi_hba_pkt_free`	Solaris DDI
`scsi_hba_probe`	Solaris DDI
`scsi_hba_tran_alloc`	Solaris DDI
`scsi_hba_tran_free`	Solaris DDI
`scsi_ifgetcap`	Solaris DDI
`scsi_ifsetcap`	Solaris DDI
`scsi_init_pkt`	Solaris DDI
`scsi_log`	Solaris DDI
`scsi_mname`	Solaris DDI
`scsi_pktalloc`	Solaris DDI
`scsi_pktfree`	Solaris DDI
`scsi_poll`	Solaris DDI
`scsi_probe`	Solaris DDI
`scsi_resalloc`	Solaris DDI
`scsi_reset`	Solaris DDI
`scsi_reset_notify`	Solaris DDI
`scsi_resfree`	Solaris DDI
`scsi_rname`	Solaris DDI
`scsi_sense_key`	Solaris DDI
`scsi_setup_cdb`	Solaris DDI
`scsi_slave`	Solaris DDI
`scsi_sname`	Solaris DDI
`scsi_sync_pkt`	Solaris DDI
`scsi_transport`	Solaris DDI

Routine	Type
`scsi_unprobe`	Solaris DDI
`scsi_unslave`	Solaris DDI
`scsi_validate_sense`	Solaris DDI
`scsi_vu_errmsg`	Solaris DDI
`sema_destroy`	Solaris DDI
`sema_init`	Solaris DDI
`sema_p`	Solaris DDI
`sema_p_sig`	Solaris DDI
`sema_tryp`	Solaris DDI
`sema_v`	Solaris DDI
`sprintf`	Solaris DDI
`stoi`	Solaris DDI
`strchr`	Solaris DDI
`strcmp`	Solaris DDI
`strcpy`	Solaris DDI
`strlen`	Solaris DDI
`strncmp`	Solaris DDI
`strncpy`	Solaris DDI
`STRUCT_DECL`	Solaris DDI
`swab`	DDI/DKI
`taskq`	Solaris DDI
`timeout`	DDI/DKI
`u8_strcmp`	Solaris DDI
`u8_textprep_str`	Solaris DDI
`u8_validate`	Solaris DDI
`uconv_u16tou32`	Solaris DDI
`uiomove`	DDI/DKI
`untimeout`	DDI/DKI

Routine	Type
`ureadc`	DDI/DKI
`usb_alloc_request`	Solaris DDI
`usb_client_attach`	Solaris DDI
`usb_clr_feature`	Solaris DDI
`usb_create_pm_components`	Solaris DDI
`usb_get_addr`	Solaris DDI
`usb_get_alt_if`	Solaris DDI
`usb_get_cfg`	Solaris DDI
`usb_get_current_frame_number`	Solaris DDI
`usb_get_dev_data`	Solaris DDI
`usb_get_max_pkts_per_ioc_request`	Solaris DDI
`usb_get_status`	Solaris DDI
`usb_get_string_desc`	Solaris DDI
`usb_handle_remote_wakeup`	Solaris DDI
`usb_lookup_ep_data`	Solaris DDI
`usb_parse_data`	Solaris DDI
`usb_pipe_bulk_xfer`	Solaris DDI
`usb_pipe_close`	Solaris DDI
`usb_pipe_ctrl_xfer`	Solaris DDI
`usb_pipe_drain_reqs`	Solaris DDI
`usb_pipe_get_max_bulk_transfer_size`	Solaris DDI
`usb_pipe_get_state`	Solaris DDI
`usb_pipe_intr_xfer`	Solaris DDI
`usb_pipe_isoc_xfer`	Solaris DDI
`usb_pipe_open`	Solaris DDI
`usb_pipe_reset`	Solaris DDI
`usb_pipe_set_private`	Solaris DDI
`usb_register_hotplug_cbs`	Solaris DDI

Routine	Type
`usb_reset_device`	Solaris DDI
`uwritec`	DDI/DKI
`va_arg`	Solaris DDI
`va_end`	Solaris DDI
`va_start`	Solaris DDI
`vcmn_err`	DDI/DKI
`vsprintf`	Solaris DDI

See Also

`Intro`(9E), `mutex`(9F)

REFERENCE

Kernel Functions for Drivers

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, interrupt, or kernel context.

See Also *STREAMS Programming Guide*

Name

allocb – allocate a message block

Synopsis

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



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

Interface Level

Architecture independent level 1 (DDI/DKI).

Description

The `allocb()` function 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.

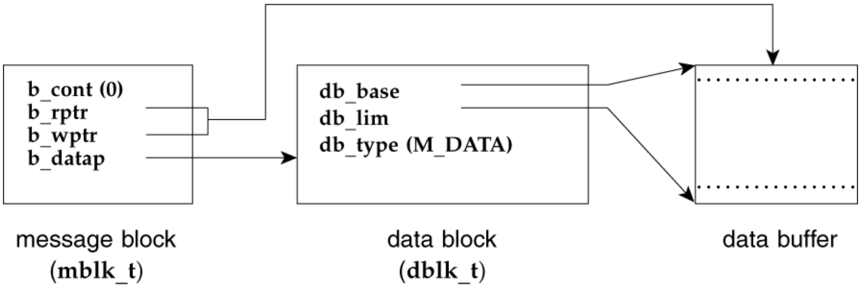
The fields in the `mblk_t` structure are initialized as follows:

`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.



Parameters *size* The number of bytes in the message block.

pri Priority of the request (no longer used).

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 The `allocb()` function can be called from user, interrupt, or kernel context.

Examples **EXAMPLE 1** `allocb()` Code Sample

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.

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->b_wptr`) by the size (one byte) of the data in the message.

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->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.

```
1  send_error(q,err)
2    queue_t *q;
3    unsigned char err;
4  {
5    mblk_t *bp;
6
7    if ((bp = allocb(1, BPRI_HI)) == NULL) /* allocate msg. block */
8         return(0);
9
10    bp->b_datap->db_type = M_ERROR;    /* set msg type to M_ERROR */
11    *bp->b_wptr++ = err;               /* increment write pointer */
12
```

EXAMPLE 1 `allocb()` Code Sample *(Continued)*

```
  if (!(q->q_flag & QREADR))         /* if not read queue     */
       q = RD(q);                    /*    get read queue     */
  putnext(q,bp);                     /* send message upstream */
  return(1);
}
```

See Also `RD`(9F), `bufcall`(9F), `esballoc`(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.

Name allocb_tmpl – allocate a message block using a template

Synopsis

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

mblk_t *allocb_tmpl(size_t size, const mblk_t *tmpl);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

size	The number of bytes in the message block.
tmpl	The template message block.

Description The `allocb_tmpl()` function tries to allocate a STREAMS message block using `allocb`(9F). If the allocation is successful, the `db_type` field in the data block structure (`dblk_t`, see `datab`(9S)), as well as some implementation-private data, are copied from the `dblk_t` associated with *tmpl*.

The `allocb_tmpl()` function should be used when a new STREAMS message block is allocated. This block is then used to contain data derived from another STREAMS message block. The original message is used as the *tmpl* argument.

Return Values Upon success, `allocb_tmpl()` returns a pointer to the allocated message block of the same type as *tmpl*. On failure, `allocb_tmpl()` returns a NULL pointer.

Context The `allocb_tmpl()` function can be called from user, interrupt, or kernel context.

See Also `allocb`(9F), `datab`(9S), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

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

Name aphysio, aminphys – 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_reqp);
```

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_reqp	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_reqp*→*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, `physio()` returns `ENOTSUP`.

- Call `aminphys(9F)` to ensure that the driver does not circumvent additional system limits. If `aminphys(9F)` does not return 0, return `ENOTSUP`.

Return Values `aphysio()` returns:

`0`	Upon success.
`non-zero`	Upon failure.

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.

Name ASSERT, assert – expression verification

Synopsis

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



void ASSERT(EX);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters		
	EX	boolean expression.

Description The `ASSERT()` macro 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.

Context The `ASSERT()` macro can be used from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

Name

atomic_add, atomic_add_8, atomic_add_char, atomic_add_16, atomic_add_short, atomic_add_32, atomic_add_int, atomic_add_long, atomic_add_64, atomic_add_ptr, atomic_add_8_nv, atomic_add_char_nv, atomic_add_16_nv, atomic_add_short_nv, atomic_add_32_nv, atomic_add_int_nv, atomic_add_long_nv, atomic_add_64_nv, atomic_add_ptr_nv – atomic add operations

Synopsis

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

void atomic_add_8(volatile uint8_t *target, int8_t delta);

void atomic_add_char(volatile uchar_t *target, signed char delta);

void atomic_add_16(volatile uint16_t *target, int16_t delta);

void atomic_add_short(volatile ushort_t *target, short delta);

void atomic_add_32(volatile uint32_t *target, int32_t delta);

void atomic_add_int(volatile uint_t *target, int delta);

void atomic_add_long(volatile ulong_t *target, long delta);

void atomic_add_64(volatile uint64_t *target, int64_t delta);

void atomic_add_ptr(volatile void *target, ssize_t delta);

uint8_t atomic_add_8_nv(volatile uint8_t *target, int8_t delta);

uchar_t atomic_add_char_nv(volatile uchar_t *target, signed char delta);

uint16_t atomic_add_16_nv(volatile uint16_t *target, int16_t delta);

ushort_t atomic_add_short_nv(volatile ushort_t *target, shortdelta);

uint32_t atomic_add_32_nv(volatile uint32_t *target, int32_t delta);

uint_t atomic_add_int_nv(volatile uint_t *target, int delta);

ulong_t atomic_add_long_nv(volatile ulong_t *target, long delta);

uint64_t atomic_add_64_nv(volatile uint64_t *target, int64_t delta);

void *atomic_add_ptr_nv(volatile void *target, ssize_t delta);
```

Description

These functions enable the addition of *delta* to the value stored in *target* to occur in an atomic manner.

Return Values

The `*_nv()` variants of these functions return the new value of *target*.

Errors

No errors are defined.

Context

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

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_and`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_dec`(9F), `atomic_inc`(9F), `atomic_or`(9F), `atomic_swap`(9F), `membar_ops`(9F), `attributes`(5), `atomic_ops`(3C)

Notes The `*_nv()` variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically* (for example, when decrementing a reference count and checking whether it went to zero).

Name atomic_and, atomic_and_8, atomic_and_uchar, atomic_and_16, atomic_and_ushort, atomic_and_32, atomic_and_uint, atomic_and_ulong, atomic_and_64, atomic_and_8_nv, atomic_and_uchar_nv, atomic_and_16_nv, atomic_and_ushort_nv, atomic_and_32_nv, atomic_and_uint_nv, atomic_and_ulong_nv, atomic_and_64_nv – atomic AND operations

Synopsis

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

void atomic_and_8(volatile uint8_t *target, uint8_t bits);

void atomic_and_uchar(volatile uchar_t *target, uchar_t bits);

void atomic_and_16(volatile uint16_t *target, uint16_t bits);

void atomic_and_ushort(volatile ushort_t *target, ushort_t bits);

void atomic_and_32(volatile uint32_t *target, uint32_t bits);

void atomic_and_uint(volatile uint_t *target, uint_t bits);

void atomic_and_ulong(volatile ulong_t *target, ulong_t bits);

void atomic_and_64(volatile uint64_t *target, uint64_t bits);

uint8_t atomic_and_8_nv(volatile uint8_t *target, uint8_t bits);

uchar_t atomic_and_uchar_nv(volatile uchar_t *target, uchar_t bits);

uint16_t atomic_and_16_nv(volatile uint16_t *target, uint16_t bits);

ushort_t atomic_and_ushort_nv(volatile ushort_t *target, ushort_t bits);

uint32_t atomic_and_32_nv(volatile uint32_t *target, uint32_t bits);

uint_t atomic_and_uint_nv(volatile uint_t *target, uint_t bits);

ulong_t atomic_and_ulong_nv(volatile ulong_t *target, ulong_t bits);

uint64_t atomic_and_64_nv(volatile uint64_t *target, uint64_t bits);
```

Description These functions enable the bitwise AND of *bits* to the value stored in *target* to occur in an atomic manner.

Return Values The `*_nv()` variants of these functions return the new value of *target*.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `atomic_add`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_dec`(9F), `atomic_inc`(9F), `atomic_or`(9F), `atomic_swap`(9F), `membar_ops`(9F), `attributes`(5), `atomic_ops`(3C)

Notes The `*_nv()` variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name atomic_bits, atomic_set_long_excl, atomic_clear_long_excl – atomic set and clear bit operations

Synopsis

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

int atomic_set_long_excl(volatile ulong_t *target, uint_t bit);

int atomic_clear_long_excl(volatile ulong_t *target, uint_t bit);
```

Description The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions perform an exclusive atomic bit set or clear operation on *target*. The value of *bit* specifies the number of the bit to be modified within target. Bits are numbered from zero to one less than the maximum number of bits in a `long`. If the value of bit falls outside of this range, the result of the operation is undefined.

Return Values The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions return 0 if *bit* was successfully set or cleared. They return -1 if *bit* was already set or cleared.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add(9F)`, `atomic_and(9F)`, `atomic_cas(9F)`, `atomic_dec(9F)`, `atomic_inc(9F)`, `atomic_or(9F)`, `atomic_swap(9F)`, `membar_ops(9F)`, `attributes(5)`, `atomic_ops(3C)`

Name atomic_cas, atomic_cas_8, atomic_cas_uchar, atomic_cas_16, atomic_cas_ushort, atomic_cas_32, atomic_cas_uint, atomic_cas_ulong, atomic_cas_64, atomic_cas_ptr – atomic compare and swap operations

Synopsis

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

uint8_t atomic_cas_8(volatile uint8_t *target, uint8_t cmp, uint8_t newval);

uchar_t atomic_cas_uchar(volatile uchar_t *target, uchar_t cmp, uchar_t newval);

uint16_t atomic_cas_16(volatile uint16_t *target, uint16_t cmp, uint16_t newval);

ushort_t atomic_cas_ushort(volatile ushort_t *target, ushort_t cmp,
     ushort_t newval);

uint32_t atomic_cas_32(volatile uint32_t *target, uint32_t cmp, uint32_t newval);

uint_t atomic_cas_uint(volatile uint_t *target, uint_t cmp, uint_t newval);

ulong_t atomic_cas_ulong(volatile ulong_t *target, ulong_t cmp, ulong_t newval);

uint64_t atomic_cas_64(volatile uint64_t *target, uint64_t cmp, uint64_t newval);

void *atomic_cas_ptr(volatile void *target, void *cmp, void *newval);
```

Description These functions enable a compare and swap operation to occur atomically. The value stored in *target* is compared with *cmp*. If these values are equal, the value stored in *target* is replaced with *newval*. The old value stored in *target* is returned by the function whether or not the replacement occurred.

Return Values These functions return the old of **target*.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add(9F)`, `atomic_and(9F)`, `atomic_bits(9F)`, `atomic_dec(9F)`, `atomic_inc(9F)`, `atomic_or(9F)`, `atomic_swap(9F)`, `membar_ops(9F)`, `attributes(5)`, `atomic_ops(3C)`

Name

atomic_dec, atomic_dec_8, atomic_dec_uchar, atomic_dec_16, atomic_dec_ushort, atomic_dec_32, atomic_dec_uint, atomic_dec_ulong, atomic_dec_64, atomic_dec_ptr, atomic_dec_8_nv, atomic_dec_uchar_nv, atomic_dec_16_nv, atomic_dec_ushort_nv, atomic_dec_32_nv, atomic_dec_uint_nv, atomic_dec_ulong_nv, atomic_dec_64_nv, atomic_dec_ptr_nv – atomic decrement operations

Synopsis

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

void atomic_dec_8(volatile uint8_t *target);

void atomic_dec_uchar(volatile uchar_t *target);

void atomic_dec_16(volatile uint16_t *target);

void atomic_dec_ushort(volatile ushort_t *target);

void atomic_dec_32(volatile uint32_t *target);

void atomic_dec_uint(volatile uint_t *target);

void atomic_dec_ulong(volatile ulong_t *target);

void atomic_dec_64(volatile uint64_t *target);

void atomic_dec_ptr(volatile void *target);

uint8_t atomic_dec_8_nv(volatile uint8_t *target);

uchar_t atomic_dec_uchar_nv(volatile uchar_t *target);

uint16_t atomic_dec_16_nv(volatile uint16_t *target);

ushort_t atomic_dec_ushort_nv(volatile ushort_t *target);

uint32_t atomic_dec_32_nv(volatile uint32_t *target);

uint_t atomic_dec_uint_nv(volatile uint_t *target);

ulong_t atomic_dec_ulong_nv(volatile ulong_t *target);

uint64_t atomic_dec_64_nv(volatile uint64_t *target);

void *atomic_dec_ptr_nv(volatile void *target);
```

Description

These functions enable the decrementing (by one) of the value stored in *target* to occur in an atomic manner.

Return Values

The `*_nv()` variants of these functions return the new value of *target*.

Errors

No errors are defined.

Context

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

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add`(9F), `atomic_and`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_inc`(9F), `atomic_or`(9F), `atomic_swap`(9F), `membar_ops`(9F), `attributes`(5), `atomic_ops`(3C)

Notes The `*_nv()` variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value atomically (for example, when decrementing a reference count and checking whether it went to zero).

Name

atomic_inc, atomic_inc_8, atomic_inc_uchar, atomic_inc_16, atomic_inc_ushort, atomic_inc_32, atomic_inc_uint, atomic_inc_ulong, atomic_inc_64, atomic_inc_ptr, atomic_inc_8_nv, atomic_inc_uchar_nv, atomic_inc_16_nv, atomic_inc_ushort_nv, atomic_inc_32_nv, atomic_inc_uint_nv, atomic_inc_ulong_nv, atomic_inc_64_nv, atomic_inc_ptr_nv – atomic inrement operations

Synopsis

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

void atomic_inc_8(volatile uint8_t *target);

void atomic_inc_uchar(volatile uchar_t *target);

void atomic_inc_16(volatile uint16_t *target);

void atomic_inc_ushort(volatile ushort_t *target);

void atomic_inc_32(volatile uint32_t *target);

void atomic_inc_uint(volatile uint_t *target);

void atomic_inc_ulong(volatile ulong_t *target);

void atomic_inc_64(volatile uint64_t *target);

void atomic_inc_ptr(volatile void *target);

uint8_t atomic_inc_8_nv(volatile uint8_t *target);

uchar_t atomic_inc_uchar_nv(volatile uchar_t *target);

uint16_t atomic_inc_16_nv(volatile uint16_t *target);

ushort_t atomic_inc_ushort_nv(volatile ushort_t *target);

uint32_t atomic_inc_32_nv(volatile uint32_t *target);

uint_t atomic_inc_uint_nv(volatile uint_t *target);

ulong_t atomic_inc_ulong_nv(volatile ulong_t *target);

uint64_t atomic_inc_64_nv(volatile uint64_t *target);

void *atomic_inc_ptr_nv(volatile void *target);
```

Description

These functions enable the inrementing (by one) of the value stored in *target* to occur in an atomic manner.

Return Values

The `*_nv()` variants of these functions return the new value of *target*.

Errors

No errors are defined.

Context

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

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add`(9F), `atomic_and`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_dec`(9F), `atomic_or`(9F), `atomic_swap`(9F), `membar_ops`(9F), `attributes`(5), `atomic_ops`(3C)

Notes The `*_nv()` variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name

atomic_ops – atomic operations

Synopsis

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

Description

This collection of functions provides atomic memory operations. There are 8 different classes of atomic operations:

`atomic_add(9F)`	These functions provide an atomic addition of a signed value to a variable.
`atomic_and(9F)`	These functions provide an atomic logical 'and' of a value to a variable.
`atomic_bits(9F)`	These functions provide atomic bit setting and clearing within a variable.
`atomic_cas(9F)`	These functions provide an atomic comparison of a value with a variable. If the comparison is equal, then swap in a new value for the variable, returning the old value of the variable in either case.
`atomic_dec(9F)`	These functions provide an atomic decrement on a variable.
`atomic_inc(9F)`	These functions provide an atomic increment on a variable.
`atomic_or(9F)`	These functions provide an atomic logical 'or' of a value to a variable.
`atomic_swap(9F)`	These functions provide an atomic swap of a value with a variable, returning the old value of the variable.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

`atomic_add(9F)`, `atomic_and(9F)`, `atomic_bits(9F)`, `atomic_cas(9F)`, `atomic_dec(9F)`, `atomic_inc(9F)`, `atomic_or(9F)`, `atomic_swap(9F)`, `membar_ops(9F)`, `attributes(5)`

Notes

Atomic instructions ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used. See `membar_ops(9F)`.

Atomic instructions can be expensive. since they require synchronization to occur at a hardware level. This means they should be used with care to ensure that forcing hardware level synchronization occurs a minimum number of times. For example, if you have several variables that need to be incremented as a group, and each needs to be done atomically, then do so with a mutex lock protecting all of them being incremented rather than using the `atomic_inc(9F)` operation on each of them.

Name atomic_or, atomic_or_8, atomic_or_uchar, atomic_or_16, atomic_or_ushort, atomic_or_32, atomic_or_uint, atomic_or_ulong, atomic_or_64, atomic_or_8_nv, atomic_or_uchar_nv, atomic_or_16_nv, atomic_or_ushort_nv, atomic_or_32_nv, atomic_or_uint_nv, atomic_or_ulong_nv, atomic_or_64_nv – atomic OR operations

Synopsis

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

void atomic_or_8(volatile uint8_t *target, uint8_t bits);

void atomic_or_uchar(volatile uchar_t *target, uchar_t bits);

void atomic_or_16(volatile uint16_t *target, uint16_t bits);

void atomic_or_ushort(volatile ushort_t *target, ushort_t bits);

void atomic_or_32(volatile uint32_t *target, uint32_t bits);

void atomic_or_uint(volatile uint_t *target, uint_t bits);

void atomic_or_ulong(volatile ulong_t *target, ulong_t bits);

void atomic_or_64(volatile uint64_t *target, uint64_t bits);

uint8_t atomic_or_8_nv(volatile uint8_t *target, uint8_t bits);

uchar_t atomic_or_uchar_nv(volatile uchar_t *target, uchar_t bits);

uint16_t atomic_or_16_nv(volatile uint16_t *target, uint16_t bits);

ushort_t atomic_or_ushort_nv(volatile ushort_t *target, ushort_t bits);

uint32_t atomic_or_32_nv(volatile uint32_t *target, uint32_t bits);

uint_t atomic_or_uint_nv(volatile uint_t *target, uint_t bits);

ulong_t atomic_or_ulong_nv(volatile ulong_t *target, ulong_t bits);

uint64_t atomic_or_64_nv(volatile uint64_t *target, uint64_t bits);
```

Description These functions enable the bitwise OR of *bits* to the value stored in *target* to occur in an atomic manner.

Return Values The `*_nv()` variants of these functions return the new value of *target*.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `atomic_add`(9F), `atomic_and`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_dec`(9F), `atomic_inc`(9F), `atomic_swap`(9F), `membar_ops`(9F), `attributes`(5), `atomic_ops`(3C)

Notes The `*_nv()` variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

Name atomic_swap, atomic_swap_8, atomic_swap_uchar, atomic_swap_16, atomic_swap_ushort, atomic_swap_32, atomic_swap_uint, atomic_swap_ulong, atomic_swap_64, atomic_swap_ptr – atomic swap operations

Synopsis

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

uint8_t atomic_swap_8(volatile uint8_t *target, uint8_t newval);

uchar_t atomic_swap_uchar(volatile uchar_t *target, uchar_t newval);

uint16_t atomic_swap_16(volatile uint16_t *target, uint16_t newval);

ushort_t atomic_swap_ushort(volatile ushort_t *target, ushort_t newval);

uint32_t atomic_swap_32(volatile uint32_t *target, uint32_t newval);

uint_t atomic_swap_uint(volatile uint_t *target, uint_t newval);

ulong_t atomic_swap_ulong(volatile ulong_t *target, ulong_t newval);

uint64_t atomic_swap_64(volatile uint64_t *target, uint64_t newval);

void *atomic_swap_ptr(volatile void *target, void *newval);
```

Description These functions enable a swap operation to occur atomically. The value stored in *target* is replaced with *newval*. The old value is returned by the function.

Return Values These functions return the old of **target*.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add(9F)`, `atomic_and(9F)`, `atomic_bits(9F)`, `atomic_dec(9F)`, `atomic_inc(9F)`, `atomic_or(9F)`, `atomic_cas(9F)`, `membar_ops(9F)`, `attributes(5)`, `atomic_ops(3C)`

Name backq – get pointer to the queue behind the current queue

Synopsis

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



queue_t *backq(queue_t *cq);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

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

Description The `backq()` function 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 The `backq()` function can be called from user, interrupt, or kernel context.

See Also queue(9S)

Writing Device Drivers

STREAMS Programming Guide

Name

bcanput – test for flow control in specified priority band

Synopsis

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



int bcanput(queue_t *q, unsigned char pri);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Pointer to the message queue.
pri	Message priority.

Description

The `bcanput()` function 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.

If *pri* is `0`, `bcanput()` is equivalent to a call with `canput(9F)`.

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

Return Values

`1`	If a message of priority *pri* can be placed on the queue.
`0`	If the priority band is full.

Context

The `bcanput()` function can be called from user, interrupt, or kernel context.

See Also

`bcanputnext(9F)`, `canput(9F)`, `canputnext(9F)`, `putbq(9F)`, `putnext(9F)`

Writing Device Drivers

STREAMS Programming Guide

Warnings

Drivers are responsible for both testing a queue with `bcanput()` and refraining from placing a message on the queue if `bcanput()` fails.

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

Architecture independent level 1 (DDI/DKI).

Parameters

s1	Pointer to the first character string.
s2	Pointer to the second character string.
len	Number of bytes to be compared.

Description

The `bcmp()` function compares two byte arrays of length *len*.

Return Values

The `bcmp()` function returns `0` if the arrays are identical, or `1` if they are not.

Context

The `bcmp()` function can be called from user, interrupt, or kernel context.

See Also

`strcmp`(9F)

Writing Device Drivers

Notes

Unlike `strcmp`(9F), `bcmp()` does not terminate when it encounters a null byte.

Name

bcopy – copy data between address locations in the kernel

Synopsis

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



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

Interface 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

The `bcopy()` function 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.

Context

The `bcopy()` function can be called from user, interrupt, or kernel context.

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.

```
 1 #define RAMDNBLK 1000                /* blocks in the RAM disk */
 2 #define RAMDBSIZ 512                 /* bytes per block */
 3 char ramdblks[RAMDNBLK][RAMDBSIZ];   /* blocks forming RAM
                                        /* disk
        ...
 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_bcount);
10
11 else                             /* else write request, */
```

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

```
                               /* copy data from a */
                               /* system buffer to RAM disk */
                               /* data block */
        bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0],
            bp->b_bcount);
```

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.

Name bioclone – clone another buffer

Synopsis `#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);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

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.

Description The `bioclone()` function 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.

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 The `bioclone()` function can be called from user, interrup, 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.

```
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_x);
       return (0);
}

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

static int
stripe_done(struct buf *bp)
{
       ...
```

EXAMPLE 1 Using `bioclone()` for Disk Striping *(Continued)*

```
        freerbuf(bp);
        fragment--;
        if (fragment == 0) {
                /* get bp_orig */
                biodone(bp_orig);
        }
        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

The `biodone()` function 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.

The `biodone()` function provides the capability to call a completion routine if *bp* describes a kernel buffer. The address of the routine is specified in the `b_iodone` 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

The `biodone()` function can be called from user, interrupt, or kernel context.

Examples

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).

```
 1   #define RAMDNBLK    1000    /* Number of blocks in RAM disk */
 2   #define RAMDBSIZ    512    /* Number of bytes per block */
 3   char ramdblks[RAMDNBLK][RAMDBSIZ];    /* 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)) {
```

```
       /*
        * If read is for block beyond RAM disk
        * limits, mark EOF condition.
        */
        bp->b_resid = bp->b_bcount;    /* compute return value */

     } else {    /* I/O attempt is beyond */
        bp->b_error = ENXIO;    /*    limits of RAM disk */
        bp->b_flags |= B_ERROR;    /* return error */
     }
     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 `read`(2), `strategy`(9E), `biowait`(9F), `ddi_add_intr`(9F), `delay`(9F), `timeout`(9F), `untimeout`(9F), `buf`(9S)

Writing Device Drivers

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_iodone` field of the `buf`(9S) structure to specify a substitute completion routine should save the value of `b_iodone` before changing it, and then restore the old value before calling `biodone()` to release the buffer.

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 occured 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

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



voidbiofini(struct buf *bp);
```

Interface Level Solaris DDI specific (Solaris DDI).

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()`

```
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

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



voidbioinit(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()`

```
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

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



intbiomodified(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

```
#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.

Name biosize – returns size of a buffer structure

Synopsis

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



size_tbiosize(void)
```

Interface Level Solaris DDI specific (Solaris DDI).

Description 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.

Context 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

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

Name bp_copyin – copy from a buf(9S) into a driver buffer

Synopsis

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



int bp_copyin(struct buf *bp, void *driverbuf, offset_t offset,
     size_t size);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

bp	Pointer to the buffer header structure to copy from.
driverbuf	Driver buffer to copy to.
offset	Offset into *bp* where to start copying.
size	Size of copy.

Description The `bp_copyin()` function copies *size* bytes into the memory associated with *bp* to the destination driver buffer *driverbuf*. The *offset* only applies to *bp*.

Return Values Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if *bp* references invalid pages.

Context The `bp_copyin()` function can be called from user or kernel context only.

See Also `bp_copyout`(9F), `bp_mapin`(9F), `bp_mapout`(9F), `ddi_copyout`(9F), `buf`(9S)

Name bp_copyout – copy from a driver buffer into a buf(9S)

Synopsis

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



int bp_copyout(void *driverbuf, struct buf *bp, offset_t offset,
     size_t size);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

bp	Pointer to the buffer header structure to copy to.
driverbuf	Driver buffer to copy from.
offset	Offset into *bp* where to start copying.
size	Size of copy.

Description The `bp_copyout()` function copies *size* bytes starting from the driver buffer *driverbuf* to *offset* bytes into the memory associated with *bp*. The *offset* only applies to *bp*.

Return Values Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if *bp* references invalid pages.

Context The `bp_copyout()` function can be called from user or kernel context only.

See Also `bp_copyin`(9F), `bp_mapin`(9F), `bp_mapout`(9F), `ddi_copyout`(9F), `buf`(9S)

Name bp_mapin – allocate virtual address space

Synopsis

```
#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

Name bp_mapout – deallocate virtual address space

Synopsis

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



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

Interface Level Architecture independent level 1 (DDI/DKI).

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 *numbytes* Number of bytes.

Description The `btop()` function 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 The `btop()` function can be called from user, interrupt, or kernel context.

See Also `btopr`(9F), `ddi_btop`(9F), `ptob`(9F)

Writing Device Drivers

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 Architecture independent level 1 (DDI/DKI).

Parameters

numbytes	Number of bytes.

Description The `btopr()` function 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 The `btopr()` function can be called from user, interrupt, or kernel context.

See Also `btop`(9F), `ddi_btopr`(9F), `ptob`(9F)

Writing Device Drivers

Name

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 *funcvoid *arg, 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

The `bufcall()` function 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

The `bufcall()` function can be called from user, interrupt, or kernel 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

EXAMPLE 1 Calling a function when a buffer becomes available: *(Continued)*

(`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.

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.

```
 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 */
                 canputnext(q)) {    /* normal & can be passed */
15            if (mp->b_datap->db_type != M_DATA)
16                 putnext(q, mp);    /* send all but M_DATA */
17            else {
18                bp = allocb(sizeof(struct hdr), BPRI_LO);
19                if (bp == NULL) {    /* if unsuccessful */
20                     putbq(q, mp);    /* put it back */
21                     if (!bufcall(sizeof(struct hdr), BPRI_LO,
                           qenable, q))    /* try to reschedule */
22                         timeout(qenable, q, drv_usectohz(500000));
23                        return (0);
24                 }
25                 hp = (struct hdr *)bp->b_wptr;
26                 hp->h_size = msgdsize(mp);    /* initialize header */
27                 hp->h_version = 1;
28                 bp->b_wptr += sizeof(struct hdr);
29                 bp->b_datap->db_type = M_PROTO;    /* make M_PROTO  */
30                 bp->b_cont = mp;    /* link it */
31                 putnext(q, bp);    /* pass it on */
32            }
33         } else {    /* normal priority, canputnext failed */
```

EXAMPLE 1 Calling a function when a buffer becomes available: *(Continued)*

```
         putbq(q, mp);    /* put back on the message queue */
         return (0);
       }
      }
return (0);
}
```

See Also `srv`(9E), `allocb`(9F), `canputnext`(9F), `esballoc`(9F), `esbbcall`(9F), `putnext`(9F), `qenable`(9F), `testb`(9F), `timeout`(9F), `unbufcall`(9F)

Writing Device Drivers

STREAMS Programming Guide

Warnings 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

```
#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 The `bzero()` function clears a contiguous portion of memory by filling it with zeros.

Context The `bzero()` function can be called from user, interrupt, or kernel 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.

Name

canput – test for room in a message queue

Synopsis

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



int canput(queue_t *q);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Pointer to the message queue.

Description

`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.

Return Values

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

Context

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

See Also

`bcanput`(9F), `bcanputnext`(9F), `canputnext`(9F), `putbq`(9F), `putnext`(9F)

Writing Device Drivers

STREAMS Programming Guide

Warnings

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

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 The `canputnext()` and `bcanputnext()` functions can be called from user, interrupt, or kernel 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 clrbuf – erase the contents of a buffer

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 The `clrbuf()` function 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 The `clrbuf()` function can be called from user, interrupt, or kernel context.

See Also `getrbuf(9F)`, `buf(9S)`

Writing Device Drivers

Name

cmn_err, vcmn_err, zcmn_err – display an error message or panic the system

Synopsis

```
#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);

#include <sys/types.h>



void zcmn_err(zoneid_t zoneid, int level, char *format...);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

cmn_err( )

level	A constant indicating the severity of the error condition.
format	Message to be displayed.

vcmn_err( )

The `vcmn_err()` function takes *level* and *format* as described for `cmn_err()`, but its third argument is different:

ap	Variable argument list passed to the function.

zcmn_err( )

The `zcmn_err()` function works exactly like `cmn_err()`, but includes an additional argument:

zoneid	Zone to which log messages should be directed. See `zones(5)`.

Description

cmn_err( )

The `cmn_err()` function 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.

The `cmn_err()` function sends log messages to the log of the global zone. `cmn_err()` appends a `\n` to each *format*, except when *level* is `CE_CONT`.

vcmn_err() The `vcmn_err()` function 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.

zcmn_err() With the exception of its first argument (`zoneid`), `zcmn_err()` is identical to `cmn_err()`. `zoneid` is the numeric ID of the zone to which the message should be directed. Note that `zoneid` only has an effect if the message is sent to the system log. Using `zoneid` will cause messages to be sent to the log associated with the specified local zone rather than the log in the global zone. This is accomplished by the message being received and processed by the `syslogd`(1M) process running in the specified zone instead of the one running in the global zone. You can retrieve a process zone ID from its credential structure using `crgetzoneid`(9F).

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

The `cmn_err()` function 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).

```
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%d: xxopen function called.",
18          ddi_binding_name(xsp->dip), getminor(dev));
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, "\020\3Intr\2Err\1Enable");
```

EXAMPLE 3 Using *regval*

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

```
reg=0xd<Intr,,Enable>
```

EXAMPLE 4 Error Routine

This example shows 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);
vsprintf(buf, fmt, ap);
va_end(ap);

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

cmn_err(level, "%s%d: %s", name, instance, buf);

}
```

EXAMPLE 5 Log to Current Zone

This example shows how messages can be sent to the log of the zone in which a thread is currently running, when applicable. Note that most hardware-related messages should instead be sent to the global zone using `cmn_err()`.

```
zcmn_err(crgetzoneid(ddi_get_cred()), CE_NOTE, "out of processes0);
```

See Also `dmesg`(1M), `kernel`(1M), `printf`(3C), `zones`(5), `ddi_binding_name`(9F), `ddi_cred`(9F), `sprintf`(9F), `va_arg`(9F), `va_end`(9F), `va_start`(9F), `vsprintf`(9F)

Writing Device Drivers

Warnings The `cmn_err()` function 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()`.

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 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()`. `cv_signal()` and `cv_broadcast()` can be called by a thread even if it does not hold the mutex passed into `cv_wait()`, though holding the mutex is necessary to ensure predictable scheduling.

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.

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.

```
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.

Not all threads can receive signals from user level processes. In cases where such reception is impossible (such as during execution of `close`(9E) due to `exit`(2)), `cv_wait_sig()` behaves as `cv_wait()`, and `cv_timedwait_sig()` behaves as `cv_timedwait()`. To avoid unkillable processes, users of these functions may need to protect against waiting indefinitely for events that might not occur. The `ddi_can_receive_sig`(9F) function is provided to detect when signal reception is possible.

See Also `kill`(2), `ddi_can_receive_sig`(9F), `ddi_get_lbolt`(9F), `mutex`(9F), `mutex_init`(9F)

Writing Device Drivers

Name

copyb – copy a message block

Synopsis

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



mblk_t *copyb(mblk_t *bp);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

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

Description

The `copyb()` function 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

The `copyb()` function can be called from user, interrupt, or kernel 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  {
```

EXAMPLE 1 Using copyb *(Continued)*

```
     mblk_t *bp, *mp;
     struct protoheader *php;

     while (ret) {
       if (!canputnext(ret->r_outq)) {   /* no room */
              ret = ret->r_next;
              continue;
       }
       bp = copyb(header);               /* copy header msg. block */
       if (bp == NULL)
             break;
       mp = dupmsg(ret->r_mp);           /* duplicate data */
       if (mp == NULL) {                 /* if unsuccessful */
            freeb(bp);                   /* free the block */
            break;
       }
       php = (struct protoheader *)bp->b_rptr;
       php->h_address = ret->r_address;   /* new header */
       bp->bp_cont = mp;                  /* link the message */
       putnext(ret->r_outq, bp);          /* send downstream */
       ret = ret->r_next;
     }
     /* reschedule */
     (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
}
```

See Also

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

Writing Device Drivers

STREAMS Programming Guide

Name

copyin – copy data from a user program to a driver buffer

Synopsis

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



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

Interface Level

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

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.

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

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  */
```

EXAMPLE 1 An `ioctl()` Routine *(Continued)*

```
     int      status;    /* physical device status word   */
     short    recv_char; /* receive character from device */
     short    xmit_char; /* transmit character to device  */
};

extern struct device xx_addr[];   /* phys. device regs. location */
  . . .
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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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*.

Name

copymsg – copy a message

Synopsis

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



mblk_t *copymsg(mblk_t *mp);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

mp	Pointer to the message to be copied.

Description

The `copymsg()` function 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

The `copymsg()` function can be called from user, interrupt, or kernel 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’))
```

EXAMPLE 1 : Using copymsg *(Continued)*

```
                            *cp -= 0x20;
           }
   }
   return(cmp);
}
```

See Also `allocb`(9F), `copyb`(9F), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

Name

copyout – copy data from a driver to a user program

Synopsis

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



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

Interface Level

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

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.

```
 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 */
```

EXAMPLE 1 An `ioctl()` Routine *(Continued)*

```
     short    xmit_char;   /* transmit character to device  */
};

extern struct device xx_addr[]; /* phys. device regs. location */
  . . .
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 (copyout(rp, arg, sizeof(struct device)))
              return(EFAULT);
          break;
             ...
    }
             ...
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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 ioctls 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*.

Name csx_AccessConfigurationRegister – read or write a PC Card Configuration Register

Synopsis

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



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

Interface Level Solaris DDI Specific (Solaris DDI)

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 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*/
```

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.
`Action`	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.
`Offset`	Specifies the byte offset for the desired configuration register from the PC Card configuration register base specified in `csx_RequestConfiguration(9F)`.
`Value`	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.

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

Name csx_ConvertSize – convert device sizes

Synopsis

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



int32_t csx_ConvertSize(convert_size_t *cs);
```

Interface Level Solaris DDI Specific (Solaris DDI)

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

Name

csx_ConvertSpeed – convert device speeds

Synopsis

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



int32_t csx_ConvertSpeed(convert_speed_t *cs);
```

Interface Level

Solaris DDI Specific (Solaris DDI)

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 Members

The structure members of `convert_speed_t` are:

```
uint32_t     Attributes;
uint32_t     nS;
uint32_t     devspeed;
```

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

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:

EXAMPLE 1 : Using csx_CS_DDI_Info *(Continued)*

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

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.

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 Level

Solaris DDI Specific (Solaris DDI)

Parameters

handle1	The access handle returned from `csx_RequestIO`(9F) or `csx_RequestWindow`(9F) that is to be duplicated.
handle2	A pointer to the newly-created duplicated data access handle.
flags	The access attributes that will be applied to the new handle.

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

Name csx_Error2Text – convert error return codes to text strings

Synopsis

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



int32_t csx_Error2Text(error2text_t *er);
```

Interface Level Solaris DDI Specific (Solaris DDI)

Parameters

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.

Structure Members The structure members of `error2text_t` are:

```
uint32_t     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

```
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

Name csx_Event2Text – convert events to text strings

Synopsis

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



int32_t csx_Event2Text(event2text_t *ev);
```

Interface Level Solaris DDI Specific (Solaris DDI)

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:

```
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()`

```
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

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

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_GetFirstClient, csx_GetNextClient – return first or next client

Synopsis

```
#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:

```
uint32_t         Socket;          /* socket number */
uint32_t         Attributes;      /* attributes */
client_handle_t  client_handle;   /* client handle */
uint32_t         num_clients;     /* number of clients */
```

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)

PC Card 95 Standard, PCMCIA/JEIDA

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

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 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_TU` 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

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

handle	Access handle returned by `csx_RequestIRQ`(9F) or `csx_RequestIO`(9F).
offset	Pointer to a `uint32_t` in which the current access handle offset is returned.

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

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 Solaris DDI Specific (Solaris DDI)

Parameters

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

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

CS_SOCK_STATUS_WRITE_PROTECT_CHANGE	Write Protect
ECS_SOCK_STATUS_CARD_LOCK_CHANGE	Card Lock Change
CS_SOCK_STATUS_EJECTION_PENDING	Ejection Request
CS_SOCK_STATUS_INSERTION_PENDING	Insertion Request

CS_SOCK_STATUS_BATTERY_DEAD_CHANGE
Battery Dead

CS_SOCK_STATUS_BATTERY_LOW_CHANGE
Battery Low

CS_SOCK_STATUS_CARD_READY_CHANGE
Ready Change

CS_SOCK_STATUS_CARD_INSERTION_CHANGE
Card is inserted

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)`

PC Card 95 Standard, PCMCIA/JEIDA

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 Level 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.

`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

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 Solaris DDI Specific (Solaris DDI)

Parameters

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()`:

REMOVE_DEVICE_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

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 *ls);
```

Interface Level Solaris DDI Specific (Solaris DDI)

Parameters

ch	Client handle returned from `csx_RegisterClient`(9F).
ls	Pointer to a `map_log_socket_t` structure.

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

Structure Members 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

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 Level Solaris DDI Specific (Solaris DDI)

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 Members 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

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`.)

`Vpp1`, `Vpp2`	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`.)

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.

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 Solaris DDI Specific (Solaris DDI)

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.

0	Reserved: do not use
1	`250 nsec`
2	`200 nsec`
3	`150 nsec`
4	`100 nsec`
5 - 7	Reserved: do not use

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

`CS_SUCCESS`	Successful operation.
`CS_BAD_HANDLE`	Window handle is invalid.
`CS_NO_CARD`	No PC Card in socket.
`CS_BAD_OFFSET`	Error getting/setting window hardware parameters.
`CS_BAD_WINDOW`	Error getting/setting window hardware parameters.
`CS_BAD_SPEED`	`AccessSpeed` is invalid.
`CS_UNSUPPORTED_FUNCTION`	No PCMCIA hardware installed.

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

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 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)`.
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

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);
```

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).
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 Members 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.

TPLBYTEORD_LOW	Little endian order
TPLBYTEORD_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.
`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

Name

csx_Parse_CISTPL_CFTABLE_ENTRY – parse 16-bit Card Configuration Table Entry tuple

Synopsis

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



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

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).
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:

```
uint32_t                      flags;     /* valid descriptions */
uint32_t                      ifc;       /* interface description */
                                         /*    information */
uint32_t                      pin;       /* values for PRR */
uint32_t                      index;     /* configuration index number */
cistpl_cftable_entry_pd_t     pd;        /* power requirements */
                                         /* description */
cistpl_cftable_entry_speed_t  speed;     /* device speed description */
cistpl_cftable_entry_io_t     io;        /* device I/O map */
cistpl_cftable_entry_irq_t    irq;       /* device IRQ utilization */
cistpl_cftable_entry_mem_t    mem;       /* device memory space */
cistpl_cftable_entry_misc_t   misc;      /* miscellaneous
                                         /* device features */
```

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

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:

```
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:

```
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:

```
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:

```
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
CISTPL_CFTABLE_TPCE_FS_IO_BUS16	16-bit flag
CISTPL_CFTABLE_TPCE_FS_IO_RANGE	IO address ranges exist

The structure members of `cistpl_cftable_entry_io_range_t` are:

```
uint32_t     addr;     /* I/O start address */
uint32_t     length;   /* I/O register length */
```

The structure members of `cistpl_cftable_entry_irq_t` are:

```
uint32_t     flags;   /* direct copy of TPCE_IR byte in tuple */
uint32_t     irqs;    /* bit mask for each allowed IRQ */
```

The structure members of `cistpl_cftable_entry_mem_t` are:

```
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];
```

The `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 `cistpl_cftable_entry_mem_window_t` are:

```
uint32_t     length;     /* length of this window */
uint32_t     card_addr;  /* card address */
uint32_t     host_addr;  /* host address */
```

The structure members of `cistpl_cftable_entry_misc_t` are:

```
uint32_t     flags;      /* miscellaneous features flags */
```

The `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
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), `csx_ValidateCIS`(9F), `tuple`(9S)

PC Card 95 Standard, PCMCIA/JEIDA

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 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)`.
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.

CONFIG_OPTION_REG_PRESENT	Configuration Option Register present
CONFIG_STATUS_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
CONFIG_IOBASE1_REG_PRESENT	IO Base 1 Register present

CONFIG_IOBASE2_REG_PRESENT	IO Base2 Register present
CONFIG_IOBASE3_REG_PRESENT	IO Base3 Register present
CONFIG_IOLIMIT_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_GetFirstTuple`(9F), `csx_GetTupleData`(9F), `csx_Parse_CISTPL_CFTABLE_ENTRY`(9F), `csx_RegisterClient`(9F), `csx_ValidateCIS`(9F), `tuple`(9S)

PC Card 95 Standard, PCMCIA/JEIDA

Notes

PC Card drivers should not attempt to use configurations beyond the "last" member in the `cistpl_config_t` structure.

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 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_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

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`, `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:

```
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:

```
uint32_t     flags;             /* flags specific to this device */
uint32_t     speed;             /* device speed in device
                                /*  speed code format */
uint32_t     nS_speed;          /* device speed in nS */
uint32_t     type;              /* device type */
uint32_t     size;              /* device size */
uint32_t     size_in_bytes;     /* device size in bytes */
```

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 `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_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

Name csx_Parse_CISTPL_DEVICEGEO – parse the Device Geo tuple

Synopsis

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



int32_t csx_Parse_CISTPL_DEVICEGEO(client_handle_t ch, tuple_t *tp,
     cistpl_devicegeo_t *pt);
```

Interface Level Solaris DDI Specific (Solaris DDI)

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, `CISTPL_DEVICEGEO`, into a form usable by PC Card drivers.

The `CISTPL_DEVICEGEO` tuple describes the device geometry of common 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

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_GetNextTuple`(9F), `csx_GetTupleData`(9F), `csx_Parse_CISTPL_DEVICEGEO_A`(9F), `csx_RegisterClient`(9F), `tuple`(9S)

PC Card 95 Standard, PCMCIA/JEIDA

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 Solaris DDI Specific (Solaris DDI)

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.
`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_CISTPL_DEVICEGEO`(9F), `csx_RegisterClient`(9F), `tuple`(9S)

PC Card 95 Standard, PCMCIA/JEIDA

Name

csx_Parse_CISTPL_FORMAT – parse the Data Recording Format tuple

Synopsis

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



int32_t csx_Parse_CISTPL_FORMAT(client_handle_t ch, tuple_t *tu, cistpl_format_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_format_t` structure which contains the parsed `CISTPL_FORMAT` tuple information upon return from this function.

Description

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.

Structure Members

The structure members of `cistpl_format_t` are:

```
uint32_t     type;
uint32_t     edc_length;
uint32_t     edc_type;
uint32_t     offset;
uint32_t     nbytes;
uint32_t     dev.disk.bksize;
uint32_t     dev.disk.nblocks;
uint32_t     dev.disk.edcloc;
uint32_t     dev.mem.flags;
uint32_t     dev.mem.reserved;
caddr_t      dev.mem.address;
uint32_t     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.
`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

Name csx_Parse_CISTPL_FUNCE – parse Function Extension tuple

Synopsis

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



int32_t csx_Parse_CISTPL_FUNCE(client_handle_t ch, tuple_t *tu, cistpl_funce_t *cf,
     uint32_t fid);
```

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).
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).

Description 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.

Structure Members 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 */
                uchar_t   cd[16];  /* CCITT country code */
        } data_modem;
        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
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
TPLFE_UC_STOP_15	1.5 stop bits supported
TPLFE_UC_STOP_2	2 stop bits 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:

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
TPLFE_FS_POLLING	Polling support
TPLFE_FS_FTP	File transfer support
TPLFE_FS_PASSWORD	Password support

`tech` This identifies the LAN technology type and is defined as follows:

TPLFE_LAN_TECH_ARCNET	Arcnet
TPLFE_LAN_TECH_ETHERNET	Ethernet
TPLFE_LAN_TECH_TOKENRING	Token Ring
TPLFE_LAN_TECH_LOCALTALK	Local Talk
TPLFE_LAN_TECH_FDDI	FDDI/CDDI
TPLFE_LAN_TECH_ATM	ATM
TPLFE_LAN_TECH_WIRELESS	Wireless

`media` This identifies the LAN media type and is defined as follows:

TPLFE_LAN_MEDIA_INHERENT	Generic interface
TPLFE_LAN_MEDIA_UTP	Unshielded twisted pair
TPLFE_LAN_MEDIA_STP	Shielded twisted pair
TPLFE_LAN_MEDIA_THIN_COAX	Thin coax
TPLFE_LAN_MEDIA_THICK_COAX	Thick coax
TPLFE_LAN_MEDIA_FIBER	Fiber
TPLFE_LAN_MEDIA_SSR_902	Spread spectrum radio 902-928 MHz
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

Name csx_Parse_CISTPL_FUNCID – parse Function Identification tuple

Synopsis

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



int32_t csx_Parse_CISTPL_FUNCID(client_handle_t ch, tuple_t *tu, cistpl_funcid_t *cf);
```

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).
cf	Pointer to a `cistpl_funcid_t` structure which contains the parsed `CISTPL_FUNCID` tuple information upon return from this function.

Description 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)).

Structure Members 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` This is the function type for `CISTPL_FUNCID`:

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
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_FUNCE`(9F), `csx_RegisterClient`(9F), `csx_ValidateCIS`(9F), `tuple`(9S)

PC Card 95 Standard, PCMCIA/JEIDA

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

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 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).
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

Name csx_Parse_CISTPL_LINKTARGET – parse the Link Target tuple

Synopsis

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


int32_t csx_Parse_CISTPL_LINKTARGET(client_handle_t ch, tuple_t *tu,
     cistpl_linktarget_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_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:

```
uint32_t   length;
char       tpltg_tag[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

```
#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 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_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_A`, 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.
`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_MFC – parse the Multi-Function tuple

Synopsis

```
#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

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_mfc_t` structure which contains the parsed `CISTPL_LONGLINK_MFC` tuple information upon return from this function.

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:

```
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

`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_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 Members The structure members of `cistpl_manfid_t` are:

```
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 – parse the Data Organization tuple

Synopsis

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



int32_t csx_Parse_CISTPL_ORG(client_handle_t ch, tuple_t *tu, cistpl_org_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_org_t` structure which contains the parsed `CISTPL_ORG` tuple information upon return from this function.

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

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 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).
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.
`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_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 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_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

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 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).
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 Members 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

Name csx_Parse_CISTPL_VERS_2 – parse Level-2 Version and Information tuple

Synopsis

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



int32_t csx_Parse_CISTPL_VERS_2(client_handle_t ch, tuple_t *tu, cistpl_vers_2_t *cv2);
```

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)`.
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

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 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)`.

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 Members 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.
`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_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

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

Name csx_RegisterClient – register a client

Synopsis

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



int32_t csx_RegisterClient(client_handle_t *ch, client_reg_t *cr);
```

Interface Level Solaris DDI Specific (Solaris DDI)

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 Members 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 */
csfunction_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.

`INFO_CARD_SHARE`
`INFO_CARD_EXCL`

If either of these bits is set, the client will receive a `CS_EVENT_REGISTRATION_C` event when Card Services has completed its internal client registration processing and after a sucessful call to `csx_RequestSocketMask(9`

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(9` 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_t` structure members are:

```
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.
`CS_OUT_OF_RESOURCE`	Card Services is unable to register client.
`CS_BAD_VERSION`	Card Services version is incompatible with client.
`CS_BAD_HANDLE`	Client has already registered for this socket.
`CS_UNSUPPORTED_FUNCTION`	No PCMCIA hardware installed.

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

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 Level Solaris DDI Specific (Solaris DDI)

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.

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

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 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

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_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

`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.
`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

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 address 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.

`BasePort1.base`
`BasePort1.handle`
`BasePort2.base`
`BasePort2.handle`
	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 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`.

`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

Notes 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 Solaris DDI Specific (Solaris DDI)

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 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 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:
	`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)`

PC Card Card 95 Standard, PCMCIA/JEIDA

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.
`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 */
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 offest */
```

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`	Points to I/O space.
`WIN_MEMORY_TYPE_CM`	Points to common memory space.
`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`	Describes device as 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).
`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. 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. 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. 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.
`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:

`Code`	`Speed`
0	(Reserved - do not use).
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nse
5-7	(Reserved—do not use.)

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

Name csx_ResetFunction – reset a function on a PC card

Synopsis

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



int32_t csx_ResetFunction(client_handle_t ch, reset_function_t *rf);
```

Interface Level Solaris DDI Specific (Solaris DDI)

Parameters

ch	Client handle returned from `csx_RegisterClient`(9F).
rf	Pointer to a `reset_function_t` structure.

Description `csx_ResetFunction()` requests that the specified function on the PC card initiate a reset operation.

Structure Members 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`.

Return Values

`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.

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

See Also `csx_event_handler`(9E), `csx_RegisterClient`(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Notes `csx_ResetFunction()` 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

```
#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.

`CS_BAD_SOCKET`	`csx_RequestSocketMask`(9F) not called for `CONF_EVENT_MASK_CLIENT`.
`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_RegisterClient`(9F), `csx_ReleaseSocketMask`(9F), `csx_RequestSocketMask`(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Name csx_SetHandleOffset – set current access handle offset

Synopsis

```
#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

Name csx_ValidateCIS – validate the Card Information Structure (CIS)

Synopsis

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



int32_t csx_ValidateCIS(client_handle_t ch, cisinfo_t *ci);
```

Interface Level Solaris DDI Specific (Solaris DDI)

Parameters

ch	Client handle returned from `csx_RegisterClient(9F)`.
ci	Pointer to a `cisinfo_t` structure.

Description This function validates the Card Information Structure (CIS) on the PC Card in the specified socket.

Structure Members The structure members of `cisinfo_t` are:

```
uint32_t     Socket;    /* socket number to validate CIS on   */
uint32_t     Chains;    /* number of tuple chains in CIS */
uint32_t     Tuples;    /* total number of tuples in CIS */
```

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.
`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.

Return Values

`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.

Context 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

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

Architecture independent level 1 (DDI/DKI).

Parameters

type The type of message to be tested. The `db_type` field of the `datab`(9S) structure contains the message type. This field may be accessed through the message block using `mp->b_datap->db_type`.

Description

The `datamsg()` function 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

The `datamsg()` function can be called from user, interrupt, or kernel context.

Examples

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.

```
xxxput(q, mp)
     queue_t *q;
     mblk_t *mp;
{
   if (datamsg(mp->b_datap->db_type)) {
           putq(q, mp);
           return;
   }
   switch (mp->b_datap->db_type) {
    case M_FLUSH:
             ...
    }
}
```

See Also

`put`(9E), `srv`(9E), `allocb`(9F), `datab`(9S), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

Name DB_BASE, DB_LIM, DB_REF, DB_TYPE – Data block access macros

Synopsis

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



uchar_t *DB_BASE(mblk_t *mp);

uchar_t *DB_LIM(mblk_t *mp);

uchar_t DB_TYPE(mblk_t *mp);

uchar_t DB_REF(mblk_t *mp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	Message block to be accessed.

Description These macros provide compact access to public members of the `datab(9S)` structure associated with the specified message block.

In all cases, these macros are equivalent to directly accessing the underlying fields of the `datab(9S)` associated with the specified message block. Specifically:

DB_BASE(mp) is equivalent to *mp->b_datap->db_base*.

DB_LIM(mp) is equivalent to *mp->b_datap->db_lim*.

DB_TYPE(mp) is equivalent to *mp->b_datap->db_type*.

DB_REF(mp) is equivalent to *mp->b_datap->db_ref*.

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

See Also `msgb(9S)`, `datab(9S)`

STREAMS Programming Guide

Name ddi_add_event_handler – add an NDI event service callback handler

Synopsis

```
#include <sys/dditypes.h>
#include <sys/sunddi.h>

int ddi_add_event_handler(dev_info_t *dip, ddi_eventcookie_t cookie,
     void (*handler)(dev_info_t *, ddi_eventcookie_t, void *, void *),
     void *arg, ddi_registration_id_t *id);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

`dev_info_t` **dip*
Device node registering the callback.

`ddi_eventcookie_t` *cookie*
Cookie returned from call to `ddi_get_eventcookie`(9F).

`void (`**handler*`)(dev_info_t *, ddi_eventcookie_t, void *, void *)`
Callback handler responsible for handling an NDI event service notification.

`void` **arg*
Pointer to opaque data supplied by the caller. Typically, this would be a pointer to the driver's `softstate` structure.

`ddi_registration_id_t` **id*
Pointer to registration ID where a unique registration id will be returned. Registration ID must be saved and used when calling `ddi_remove_event_handler`(9F) to unregister a callback.

Description The `ddi_add_event_handler()` function adds a callback handler to be invoked in the face of the event specifed by *cookie*. The process of adding a callback handler is also known as subscribing to an event. Upon successful subscription, the handler will be invoked by the system when the event occurs. The handler can be unregistered by using `ddi_remove_event_handler`(9F).

An instance of a driver can register multiple handlers for an event or a single handler for multiple events. Callback order is not defined and should assumed to be random.

The routine handler will be invoked with the following arguments:

`dev_info_t` **dip*	Device node requesting the notification.
`ddi_eventcookie_t` *cookie*	Structure describing event that occurred.
`void` **arg*	Opaque data pointer provided, by the driver, during callback registration.
`void` **impl_data*	Pointer to event specific data defined by the framework which invokes the callback function.

Return Values

`DDI_SUCCESS`	Callback handler registered successfully.
`DDI_FAILURE`	Failed to register callback handler. Possible reasons include lack of resources or a bad cookie.

Context

The `ddi_add_event_handler()` and `handler()` function can be called from user and kernel contexts only.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Evolving

See Also

`attributes(5)`, `ddi_get_eventcookie(9F)`, `ddi_remove_event_handler(9F)`

Writing Device Drivers

Notes

Drivers must remove all registered callback handlers for a device instance by calling `ddi_remove_event_handler(9F)` before detach completes.

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

Solaris DDI specific (Solaris DDI). These interfaces are obsolete. Use the new interrupt interfaces referenced in `Intro`(9F). Refer to *Writing Device Drivers* for more information.

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.
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.

Description

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), and `driver.conf`(4)). If only one interrupt is associated with the device, *inumber* should be `0`.

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.

ddi_add_intr() `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_cookiep* 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_cookiep* is no longer useful and should be set to `NULL`.

On a successful return, *idevice_cookiep* 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_cookiep* 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()` returns `DDI_SUCCESS`. If the interrupt information cannot be found on the sun4u architecture, either `DDI_INTR_NOTFOUND` or `DDI_FAILURE` can be returned. On i86pc and sun4m architectures, if the interrupt information cannot be found, `DDI_INTR_NOTFOUND` is returned.

ddi_remove_intr() `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 All Three Functions: For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA and SBus buses. See `sysbus`(4), `isa`(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.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also

`driver.conf`(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

```
#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). These interfaces are obsolete. Use the new interrupt interfaces referenced in `Intro`(9F). Refer to *Writing Device Drivers* for more information.

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.

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 *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

EXAMPLE 1 device using high-level interrupts *(Continued)*

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).

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 */
```

EXAMPLE 2 sample `attach()` routine *(Continued)*

```
        }

        /* 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 */
        }

        /* 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)
{
```

EXAMPLE 3 High-level interrupt routine *(Continued)*

```
      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)
                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 */
```

EXAMPLE 3 High-level interrupt routine *(Continued)*

```
                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);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also

`ddi_add_intr`(9F), `ddi_in_panic`(9F), `ddi_intr_hilevel`(9F), `ddi_remove_intr`(9F), `Intro`(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_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.

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 Level Solaris DDI specific (Solaris DDI).

Description This set of routines use the parent nexus driver to perform conversions in page size units.

The `ddi_btop()` function 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.

The `ddi_btopr()` function 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.

The `ddi_ptob()` function 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 The `ddi_btop()` and `ddi_btopr()` functions 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, interrupt, or kernel 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

Name ddi_can_receive_sig – Test for ability to receive signals

Synopsis

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



boolean_t ddi_can_receive_sig(void);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters None.

Description The `ddi_can_receive_sig()` function returns a boolean value indicating whether the current thread can receive signals sent by `kill`(2). If the return value is `B_FALSE`, then the calling thread cannot receive signals, and any call to `qwait_sig`(9F), `cv_wait_sig`(9F), or `cv_timedwait_sig`(9F) implicitly becomes `qwait`(9F), `cv_wait`(9F), or `cv_timedwait`(9F), respectively. Drivers that can block indefinitely awaiting an event should use this function to determine if additional means (such as `timeout`(9F)) may be necessary to avoid creating unkillable threads.

Return Values

`B_FALSE`	The calling thread is in a state in which signals cannot be received. For example, the thread is not associated with a user process or is in the midst of `exit`(2) handling.
`B_TRUE`	The calling thread may receive a signal while blocked on a condition variable. Note that this function does not check to determine whether signals are blocked (see `sigprocmask`(2)).

Context The `ddi_can_receive_sig()` function may be called from user, kernel, or interrupt context.

See Also `close`(9E), `cv_wait`(9F), `qwait`(9F)

Name ddi_cb_register, ddi_cb_unregister – register and unregister a device driver callback handler

Synopsis

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

int ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags,
      ddi_cb_func_t cbfunc, void *arg1, void *arg2,
      ddi_cb_handle_t * ret_hdlp);

int ddi_cb_unregister(ddi_cb_handle_t hdl);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_cb_register()`

dip	Pointer to the `dev_info` structure.
flags	Flags to determine which callback events can be handled.
cbfunc	Callback handler function.
arg1	First argument to the callback handler.
arg2	Second (optional) argument to the callback handler.
ret_hdlp	Pointer to return a handle to the registered callback.

`ddi_cb_unregister()`

hdl	Handle to the registered callback handler that is to be unregistered.

Description The `ddi_cb_register()` function installs a callback handler which processes various actions that require the driver's attention while it is attached. The driver specifies which callback actions it can handle through the flags parameter. With each relevant action, the specified callback function passes the *arg1* and *arg2* arguments along with the description of each callback event to the driver.

The `ddi_cb_unregister()` function removes a previously installed callback handler and prevents future processing of actions.

The *flags* parameter consists of the following:

`DDI_CB_FLAG_INTR`	The device driver participates in interrupt resource management. The device driver may receive additional interrupt resources from the system, but only because it can accept callback notices informing it when it has more or less resources available. Callback notices can occur at anytime after the driver is attached. Interrupt availability varies based on the overall needs of the system.

The cdfunc is a callback handler with the following prototype:

```
typedef int (*ddi_cb_func_t)(dev_info_t *dip,
              ddi_cb_action_t action, void *cbarg,
              void *arg1, void *arg2);
```

The *cbfunc* routine with the arguments *dip*, *action*, *cbarg*, *arg1* and *arg2* is called upon receipt of any callbacks for which the driver is registered. The callback handler returns `DDI_SUCCESS` if the callback was handled successfully, `DDI_ENOTSUP` if it received a callback action that it did not know how to process, or `DDI_FAILURE` if it has an internal failure while processing an action.

The *action* parameter can be one of the following:

DDI_CB_INTR_ADD	For interrupt resource management, the driver has more available interrupts. The driver can allocate more interrupt vectors and then set up more interrupt handling functions by using `ddi_intr_alloc`(9F).
DDI_CB_INTR_REMOVE	For interrupt resource management, the driver has fewer available interrupts. The driver must release any previously allocated interrupts in excess of what is now available by using `ddi_intr_free`(9F).

The *cbarg* parameter points to an action-specific argument. Each class of registered actions specifies its own data structure that a callback handler should dereference when it receives those actions.

The *cbarg* parameter is defined as an integer in the case of `DDI_CB_INTR_ADD` and `DDI_CB_INTR_REMOVE` actions. The callback handler should cast the *cbarg* parameter to an integer. The integer represents how many interrupts have been added or removed from the total number available to the device driver.

If a driver participates in interrupt resource management, it must register a callback with the `DDI_CB_FLAG_INTR` flag. The driver then receives the actions `DDI_CB_INTR_ADD` and `DDI_CB_INTR_REMOVE` whenever its interrupt availability has changed. The callback handler should use the interrupt functions `ddi_intr_alloc`(9F) and `ddi_intr_free`(9F) functions to respond accordingly. A driver is not required to allocate all interrupts that are available to it, but it is required to manage its allocations so that it never uses more interrupts than are currently available.

Return Values

The `ddi_cb_register()` and `ddi_cb_unregister()` functions return:

`DDI_SUCCESS`	on success
`DDI_EINVAL`	An invalid parameter was given when registering a callback handler, or an invalid handle was given when unregistering.
`DDI_EALREADY`	An attempt was made to register a callback handler while a previous registration still exists.

The *cbfunc* routine must return:

`DDI_SUCCESS`	on success
`DDI_ENOTSUP`	The device does not support the operation
`DDI_FAILURE`	Implementation specific failure

Context These functions can be called from kernel, non-interrupt context.

Examples **EXAMPLE 1** ddi_cb_register

```
/*
    * attach(9F) routine.
    *
    * Creates soft state, registers callback handler, initializes
    * hardware, and sets up interrupt handling for the driver.
    */
    xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
    {
        xx_state_t              *statep = NULL;
        xx_intr_t               *intrs = NULL;
        ddi_intr_handle_t       *hdls;
        ddi_cb_handle_t         cb_hdl;
        int                     instance;
        int                     type;
        int                     types;
        int                     nintrs;
        int                     nactual;
        int                     inum;

        /* Get device instance */
        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_ATTACH:

             /* Get soft state */
             if (ddi_soft_state_zalloc(state_list, instance) != 0)
                     return (DDI_FAILURE);
             statep = ddi_get_soft_state(state_list, instance);
             ddi_set_driver_private(dip, (caddr_t)statep);
             statep->dip = dip;

             /* Initialize hardware */
             xx_initialize(statep);

             /* Register callback handler */
             if (ddi_cb_register(dip, DDI_CB_FLAG_INTR, xx_cbfunc,
```

EXAMPLE 1 ddi_cb_register *(Continued)*

```
            statep, NULL, &cb_hdl) != 0) {
                ddi_soft_state_free(state_list, instance);
                return (DDI_FAILURE);
        }
        statep->cb_hdl = cb_hdl;

        /* Select interrupt type */
        ddi_intr_get_supported_types(dip, &types);
        if (types & DDI_INTR_TYPE_MSIX) {
                type = DDI_INTR_TYPE_MSIX;
        } else if (types & DDI_INTR_TYPE_MSI) {
                type = DDI_INTR_TYPE_MSI;
        } else {
                type = DDI_INTR_TYPE_FIXED;
        }
        statep->type = type;

        /* Get number of supported interrupts */

        ddi_intr_get_nintrs(dip, type, &nintrs);

        /* Allocate interrupt handle array */
        statep->hdls_size = nintrs * sizeof (ddi_intr_handle_t);
        hdls = kmem_zalloc(statep->hdls_size, KMEM_SLEEP);

        /* Allocate interrupt setup array */
        statep->intrs_size = nintrs * sizeof (xx_intr_t);
        statep->intrs = kmem_zalloc(statep->intrs_size, KMEM_SLEEP);

        /* Allocate interrupt vectors */
        ddi_intr_alloc(dip, hdls, type, 0, nintrs, &nactual, 0);
        statep->nactual = nactual;

        /* Configure interrupt handling */
        xx_setup_interrupts(statep, nactual, statep->intrs);

        /* Install and enable interrupt handlers */
        for (inum = 0; inum < nactual; inum++) {
                ddi_intr_add_handler(&statep->hdls[inum],
                    statep->intrs[inum].inthandler,
                    statep->intrs[inum].arg1,
                    statep->intrs[inum].arg2);
                ddi_intr_enable(statep->hdls[inum]);
        }
```

EXAMPLE 1 ddi_cb_register *(Continued)*

```
                break;

            case DDI_RESUME:

                    /* Get soft state */
                    statep = ddi_get_soft_state(state_list, instance);
                    if (statep == NULL)
                            return (DDI_FAILURE);

                    /* Resume hardware */
                    xx_resume(statep);

                    break;
            }

            return (DDI_SUCESS);
        }

        /*
         * detach(9F) routine.
         *
         * Stops the hardware, disables interrupt handling, unregisters
         * a callback handler, and destroys the soft state for the driver.
         */
        xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
        {
            xx_state_t      *statep = NULL;
            int             instance;
            int             inum;


            /* Get device instance */
            instance = ddi_get_instance(dip);

            switch (cmd) {
            case DDI_DETACH:

                    /* Get soft state */
                    statep = ddi_get_soft_state(state_list, instance);
                    if (statep == NULL)
                            return (DDI_FAILURE);

                    /* Stop device */
                    xx_uninitialize(statep);
```

EXAMPLE 1 ddi_cb_register *(Continued)*

```
                /* Disable and free interrupts */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_disable(statep->hdls[inum]);
                        ddi_intr_remove_handler(statep->hdls[inum]);
                        ddi_intr_free(statep->hdls[inum]);
                }

                /* Unregister callback handler */
                ddi_cb_unregister(statep->cb_hdl);

                /* Free interrupt handle array */
                kmem_free(statep->hdls, statep->hdls_size);

                /* Free interrupt setup array */
                kmem_free(statep->intrs, statep->intrs_size);

                /* Free soft state */
                ddi_soft_state_free(state_list, instance);

                break;

        case DDI_SUSPEND:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Suspend hardware */
                xx_quiesce(statep);

                break;
        }

        return (DDI_SUCCESS);
    }

    /*
     * (*ddi_cbfunc)() routine.
     *
     * Adapt interrupt usage when availability changes.
     */
    int
    xx_cbfunc(dev_info_t *dip, ddi_cb_action_t cbaction, void *cbarg,
        void *arg1, void *arg2)
```

EXAMPLE 1 ddi_cb_register *(Continued)*

```
{
    xx_state_t      *statep = (xx_state_t *)arg1;
    int             count;
    int             inum;
    int             nactual;

    switch (cbaction) {
    case DDI_CB_INTR_ADD:
    case DDI_CB_INTR_REMOVE:

         /* Get change in availability */
         count = (int)(uintptr_t)cbarg;

         /* Suspend hardware */
         xx_quiesce(statep);

         /* Tear down previous interrupt handling */
         for (inum = 0; inum < statep->nactual; inum++) {
                 ddi_intr_disable(statep->hdls[inum]);
                 ddi_intr_remove_handler(statep->hdls[inum]);
         }

         /* Adjust interrupt vector allocations */
         if (cbaction == DDI_CB_INTR_ADD) {

                 /* Allocate additional interrupt vectors */
                 ddi_intr_alloc(dip, statep->hdls, statep->type,
                     statep->nactual, count, &nactual, 0);

                 /* Update actual count of available interrupts */
                 statep->nactual += nactual;

         } else {

                 /* Free removed interrupt vectors */
                 for (inum = statep->nactual - count;
                     inum < statep->nactual; inum++) {
                         ddi_intr_free(statep->hdls[inum]);
                 }

                 /* Update actual count of available interrupts */
                 statep->nactual -= count;
         }

         /* Configure interrupt handling */
```

EXAMPLE 1 ddi_cb_register *(Continued)*

```
                xx_setup_interrupts(statep, statep->nactual, statep->intrs);

                /* Install and enable interrupt handlers */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_add_handler(&statep->hdls[inum],
                            statep->intrs[inum].inthandler,
                            statep->intrs[inum].arg1,
                            statep->intrs[inum].arg2);
                        ddi_intr_enable(statep->hdls[inum]);
                }

                /* Resume hardware */
                xx_resume(statep);

                break;

        default:
                return (DDI_ENOTSUP);
        }

        return (DDI_SUCCESS);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Private
MT-Level	Unsafe

See Also

`attributes(5)`, `ddi_intr_alloc(9F)`, `ddi_intr_free(9F)`, `ddi_intr_set_nreq(9F)`

Notes

Users of these interfaces that register for `DDI_CB_FLAG_INTR` become participants in interrupt resource management. With that participation comes a responsibility to properly adjust interrupt usage. In the case of a `DDI_CB_INTR_ADD` action, the system guarantees that a driver can allocate a total number of interrupt resources up to its new number of available interrupts. The total number of interrupt resources is the sum of all resources allocated by the function `ddi_intr_alloc(9F)`, minus all previously released by the function `ddi_intr_free(9F)`. In the case of a `DDI_CB_INTR_REMOVE` action, the driver might have more interrupts allocated than are now currently available. It is necessary for the driver to release the excess interrupts, or it will have a negative impact on the interrupt availability for other drivers in the system.

A failure to release interrupts in response to a `DDI_CB_INTR_REMOVE` callback generates the following warning on the system console:

```
WARNING: <driver><instance>: failed to release interrupts for
        IRM (nintrs = ##, navail=##).
```

Participation in interrupt resource management ends when a driver uses the `ddi_cb_unregister()` function to unregister its callback function. The callback function must still operate properly until after the call to the `ddi_cb_unregister()` function completes. If `addinterrupts` were given to the driver because of its participation, then a final use of the callback function occurs to release the additional interrupts. The call to the `ddi_cb_unregister()` function blocks until the final use of the registered callback function is finished.

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.

Return Values

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.

Examples

```
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);
    }
    \\&...
```

Context

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)

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 Level Solaris DDI specific (Solaris DDI).

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 ioctls. `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.

Examples

EXAMPLE 1 `ddi_copyin()` example

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.

```
 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             */
```

EXAMPLE 1 `ddi_copyin()` example *(Continued)*

```
          mutex_exit(&sp->reg_mutex);

          break;
    }
}
```

See Also `ioctl`(9E), `bcopy`(9F), `copyin`(9F), `copyout`(9F), `ddi_copyout`(9F), `uiomove`(9F)

Writing Device Drivers

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*.

Name

ddi_copyout – copy data from a driver

Synopsis

```
#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.

Examples

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.

```
 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 registers to arg */
26            mutex_enter(&sp->reg_mutex);
27            /*
28             * Copy data from device registers to
29             * temporary device register buffer
30             * e.g. reg_buf.control = rp->control;
31             */
32            mutex_exit(&sp->reg_mutex);
33            if (ddi_copyout(&reg_buf, arg,
34                sizeof (struct device), mode) != 0) {
```

EXAMPLE 1 `ddi_copyout()` example *(Continued)*

```
                  return (EFAULT);
          }

          break;
    }
}
```

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*.

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 Solaris DDI specific (Solaris DDI).

Parameters

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 literal 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_FD`	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`.

Description `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.

Return Values

`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.

Context

The `ddi_create_minor_node()` function can be called from user context. It is typically called from `attach`(9E) or `ioctl`(9E).

Examples

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.

```
ddi_create_minor_node(dip, "foo", S_IFBLK, 0, DDI_NT_BLOCK, 0);
```

See Also

`add_drv`(1M), `devlinks`(1M), `disks`(1M), `drvconfig`(1M), `ports`(1M), `tapes`(1M), `attach`(9E), `ddi_remove_minor_node`(9F)

Writing Device Drivers

Notes

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.

Non-`gld`(7D)-based DLPI network streams drivers are encouraged to switch to `gld`(7D). Failing this, a driver that creates DLPI style-2 minor nodes must specify CLONE_DEV for its style-2 `ddi_create_minor_node()` nodes and use `qassociate`(9F). A driver that supports both style-1 and style-2 minor nodes should return DDI_FAILURE for DDI_INFO_DEVT2INSTANCE and DDI_INFO_DEVT2DEVINFO `getinfo`(9E) calls to style-2 minor nodes. (The correct association is already established by `qassociate`(9F)). A driver that only supports style-2 minor nodes can use `ddi_no_info`(9F) for its `getinfo`(9E) implementation. For drivers that do not follow these rules, the results of a `modunload`(1M) of the driver or a `cfgadm`(1M) remove of hardware controlled by the driver are undefined.

Warning

Drivers must remove references to GLOBAL_DEV, NODEBOUND_DEV, NODESPECIFIC_DEV, and ENUMERATED_DEV to compile under Solaris 10 and later versions.

Name

ddi_cred, crgetuid, crgetruid, crgetsuid, crgetgid, crgetrgid, crgetsgid, crgetzoneid, crgetgroups, crgetngroups – access and change parts of the cred_t structure

Synopsis

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



uid_t crgetuid(const cred_t *cr);

uid_t crgetruid(const cred_t *cr);

uid_t crgetsuid(const cred_t *cr);

gid_t crgetgid(const cred_t *cr);

gid_t crgetrgid(const cred_t *cr);

gid_t crgetsgid(const cred_t *cr);

zoneid_t crgetzoneid(const cred_t *cr);

const gid_t *crgetgroups(const cred_t *cr);

int crgetngroups(const cred_t *cr);

int crsetresuid(cred_t *cr, uid_t ruid, uid_t euid, uid_t suid);

int crsetresgid(cred_t *cr, gid_t rgid, gid_t egid, gid_t sgid);

int crsetugid(cred_t *cr, uid_t uid, gid_t gid);

int crsetgroups(cred_t *cr, int ngroups, gid_t gids);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

cr	pointer to the user credential structure
uid, *ruid*, *euid*, *suid*	new user id, real, effective and saved user id
gid, *rgid*, *egid*, *sgid*	new group id, real, effective and saved group id
ngroups	number of groups in the group array
gids	pointer to array of new groups

Description

The user credential is a shared, read-only, ref-counted data structure. Its actual size and layout are subject to change. The functions described in this page allow the programmer to retrieve fields from the structure and to initialize newly allocated credential structures.

`crgetuid()`, `crgetruid()`, and `crgetsuid()` return, respectively, the effective, real, and saved user id from the user credential pointed to by *cr*.

`crgetgid()`, `crgetrgid()`, and `crgetsgid()` return, respectively, the effective, real, and saved group id from the user credential pointed to by *cr*.

`crgetzoneid()` returns the zone id from the user credential pointed to by *cr*.

`crgetgroups()` returns the group list of the user credential pointed to by *cr*.

`crgetngroups()` returns the number of groups in the user credential pointed to by *cr*.

`crsetresuid()` sets the real, effective and saved user id. All but one can be specified as -1, which causes the original value not to change.

`crsetresgid()` sets the real, effective and saved group id. All but one can be specified as -1, which causes the original value not to change.

`crsetugid()` initializes the real, effective and saved user id `all` to *uid*. It initializes the real, effective, and saved group id `all` to *gid*.

`crsetgroups()` sets the number of groups in the user credential to *ngroups* and copies the groups from *gids* to the user credential. If *ngroups* is 0, *gids* need not point to valid storage.

It is an error to call this any of the `crset*()` functions on a user credential structure that was newly allocated.

Return Values The `crget*()` functions return the requested information.

The `crset*id()` functions return 0 on success and -1 if any of the specified ids are invalid. The functions might cause a system panic if called on a user credential structure that is referenced by other parts of the system.

Context These functions can be called from user and kernel contexts.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Interface Stability	Evolving

See Also `attributes(5)`, `privileges(5)`, `drv_priv(9F)`

Writing Device Drivers

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);
```

Interface Level Solaris DDI specific (Solaris DDI).

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

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.
`DDI_FAILURE`	The byte count is not a multiple *dev_datasz*.

Context `ddi_device_copy()` can be called from user, kernel, or interrupt context.

See Also `ddi_regs_map_free`(9F), `ddi_regs_map_setup`(9F)

Writing Device Drivers

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 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	Beginning of the device address.
bytecount	Number of bytes to zero.
dev_advcnt	Number of *dev_datasz* units to advance on every access.
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

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

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_get, 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);

int ddi_devid_get(dev_info_t *dip, ddi_devid_t *retdevid);

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.
retdevid	The return address of the device ID.
retminor_name	The return address of a minor name. Free string with `ddi_devid_str_free()`.

Interface Level Solaris DDI specific (Solaris DDI).

Description 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.

`ddi_devid_get()` returns a pointer to the device ID structure through *retdevid* if there is already a registered device ID associated with the `dev_info` node. A driver can use this interface to check and get the device ID associated with the `dev_info` node. If no device ID is registered for the node then it returns `DDI_FAILURE`.

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_get()` returns the following values:

`DDI_SUCCESS`	Device ID is present and a pointer to it is returned in *retdevid*.
`DDI_FAILURE`	No device ID is defined for this `dev_info` node.

`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 – inform the system that a device's component is required

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



int ddi_dev_is_needed(dev_info_t *dip, int component, int level);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

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.

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:

```
     static int
xxdisk_spun_down(struct xxstate *xsp)
{
        return (xsp->power_level[DISK_COMPONENT] < POWER_SPUN_UP);
}
static int
```

EXAMPLE 1 disk driver code *(Continued)*

```
xxdisk_strategy(struct buf *bp)
{

. . .

        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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Obsolete

See Also `pm(7D)`, `pm-components(9P)`, `attach(9E)`, `detach(9E)`, `power(9E)`, `pm_busy_component(9F)`, `pm_idle_component(9F)`

Writing Device Drivers

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

The `ddi_dev_is_sid()` function 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

The `ddi_dev_is_sid()` function can be called from user, interrupt, or kernel context.

Examples

```
...
int
bz_probe(dev_info_t *dip)
{
    ...
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS) {
        /*
         * This is the self-identifying version (OpenBoot).
         * No need to probe for it because we know it is there.
         * The existence of dip && ddi_dev_is_sid() proves this.
         */
            return (DDI_PROBE_DONTCARE);
    }
    /*
     * Not a self-identifying variant of the device. Now we have to
     * do some work to see whether it is really attached to the
     * system.
     */
...
```

See Also

`probe`(9E) *Writing Device Drivers*

Name 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). This interface is obsolete. Use the new interrupt interfaces referenced in `Intro(9F)`. Refer to *Writing Device Drivers* for more information.

Description The `ddi_dev_nintrs()` function returns the number of interrupt specifications a device has in **resultp*.

Return Values The `ddi_dev_nintrs()` function 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 The `ddi_dev_nintrs()` function can be called from user, interrupt, or kernel context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `isa(4)`, `sbus(4)`, `ddi_add_intr(9F)`, `ddi_dev_nregs(9F)`, `ddi_dev_regsize(9F)`, `Intro(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 `ddi_dev_nregs()` function returns the number of sets of registers the device has.

Return Values The `ddi_dev_nregs()` function returns:

`DDI_SUCCESS`	A successful return. The number of register sets is returned in *resultp*.
`DDI_FAILURE`	The device has no registers.

Context The `ddi_dev_nregs()` function can be called from user, interrupt, or kernel context.

See Also `ddi_dev_nintrs`(9F), `ddi_dev_regsize`(9F)

Writing Device Drivers

Name

ddi_dev_regsize – return the size of a device's register

Synopsis

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



int ddi_dev_regsize(dev_info_t *dip, uint_t rnumber, off_t *resultp);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

dip	A pointer to the device's `dev_info` structure.
rnumber	The ordinal register number. Device registers are associated with a `dev_info` and are enumerated in arbitrary sets from `0` on up. The number of registers a device has can be determined from a call to `ddi_dev_nregs`(9F).
resultp	Pointer to an integer that holds the size, in bytes, of the described register (if it exists).

Description

The `ddi_dev_regsize()` function returns the size, in bytes, of the device register specified by *dip* and *rnumber*. This is useful when, for example, one of the registers is a frame buffer with a varying size known only to its proms.

Return Values

The `ddi_dev_regsize()` function returns:

`DDI_SUCCESS`	A successful return. The size, in bytes, of the specified register, is set in *resultp*.
`DDI_FAILURE`	An invalid (nonexistent) register number was specified.

Context

The `ddi_dev_regsize()` function can be called from user, interrupt, or kernel context.

See Also

`ddi_dev_nintrs`(9F), `ddi_dev_nregs`(9F)

Writing Device Drivers

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 Level Solaris DDI specific (Solaris DDI)

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_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:

```
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) {
        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

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



int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, struct as *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);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

handle	The DMA handle previously allocated by a call to `ddi_dma_alloc_handle(9F)`.
as	A pointer to an address space structure. This parameter 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	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 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, *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.

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 **countp*-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

`ddi_dma_addr_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_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)

Writing Device Drivers

Notes 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).

Name 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);
```

Interface Level 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 The `ddi_dma_addr_setup()` function 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 The `ddi_dma_addr_setup()` can be called from user, interrupt, or kernel context, except when *waitfp* is set to `DDI_DMA_SLEEP`, in which case it cannot be called from interrupt context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers

Name	ddi_dma_alloc_handle – allocate DMA handle

Synopsis

```
#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);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

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.

Description `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 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 `ddi_dma_alloc_handle()` returns:

`DDI_SUCCESS`	Successfully allocated a new DMA handle.
`DDI_DMA_BADATTR`	The attributes specified in the `ddi_dma_attr`(9S) structure make it impossible for the system to allocate potential DMA resources.
`DDI_DMA_NORESOURCES`	No resources are available.

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

```
#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.

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. **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)` **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

Notes 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).

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

Parameters

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 The `ddi_dma_buf_setup()` function 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 The `ddi_dma_buf_setup()` function can be called from user, interrupt, or kernel context, except when *waitfp* is set to `DDI_DMA_SLEEP`, in which case it cannot be called from interrupt context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

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 burstsizes *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

```
#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 Solaris SPARC DDI (Solaris SPARC DDI). This interface is obsolete.

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 The `ddi_dma_coff()` function 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.

The `ddi_dma_coff()` function 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 The `ddi_dma_coff()` function returns:

`DDI_SUCCESS`	Successfully filled in *offp*.
`DDI_FAILURE`	Failed to successfully fill in *offp*.

Context The `ddi_dma_coff()` function can be called from user, interrupt, or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `ddi_dma_setup`(9F), `ddi_dma_sync`(9F), `ddi_dma_cookie`(9S)

Writing Device Drivers

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 The `ddi_dma_curwin()` function 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 The `ddi_dma_curwin()` function returns:

`DDI_SUCCESS`	The current length and offset can be established.
`DDI_FAILURE`	Otherwise.

Context The `ddi_dma_curwin()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

See Also `attributes`(5), `ddi_dma_getwin`(9F), `ddi_dma_movwin`(9F), `ddi_dma_setup`(9F), `ddi_dma_req`(9S)

Writing Device Drivers

Name ddi_dma_devalign – find DMA mapping alignment and minimum transfer size

Synopsis

```
#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). This interface is obsolete.

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_x86`(9S)). This also is guaranteed to be a power of two.

Description The `ddi_dma_devalign()` Function determines after a successful DMA mapping (see `ddi_dma_setup`(9F)) the minimum required data alignment and minimum DMA transfer size.

Return Values The `ddi_dma_devalign()` function returns:

`DDI_SUCCESS`	The *alignment* and *minxfr* values have been filled.
`DDI_FAILURE`	The handle was illegal.

Context The `ddi_dma_devalign()` function can be called from user, interrupt, or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `ddi_dma_setup`(9F), `ddi_iomin`(9F), `ddi_dma_lim_sparc`(9S), `ddi_dma_lim_x86`(9S), `ddi_dma_req`(9S)

Writing Device Drivers

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

dip	A `dev_info` pointer that identifies the device.
chnl	A DMA channel number. On ISA 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_x86`(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 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()` 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_x86`(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 can be called from user, interrupt, or kernel context.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	x86

See Also

`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 Level This interface is obsolete. `ddi_dma_free_handle`(9F) should be used instead.

Parameters

handle	The handle filled in by a call to `ddi_dma_setup`(9F).

Description The `ddi_dma_free()` function 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 The `ddi_dma_free()` function returns:

`DDI_SUCCESS`	Successfully released resources
`DDI_FAILURE`	Failed to free resources

Context The `ddi_dma_free()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

Name ddi_dma_free_handle – free DMA handle

Synopsis

```
#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

Name ddi_dma_get_attr – get the device DMA attribute structure from a DMA handle

Synopsis

```
#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)

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 Level

Solaris DDI specific (Solaris DDI).

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()`.

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:

`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

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.
cookiep	A pointer to a `ddi_dma_cookie(9S)` structure.

Description The `ddi_dma_htoc()` function 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 The `ddi_dma_htoc()` function returns:

`DDI_SUCCESS`	Successfully filled in the cookie pointed to by *cookiep*.
`DDI_FAILURE`	Failed to successfully fill in the cookie.

Context The `ddi_dma_htoc()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

Name

ddi_dma_mem_alloc – allocate memory for DMA transfer

Synopsis

```
#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);
```

Interface Level

Solaris DDI specific (Solaris DDI).

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 `ddi_device_acc_attr()` structure of the device. See `ddi_device_acc_attr`(9S). The value in `devacc_attr_dataorder` is ignored in the current release. The value in `devacc_attr_endian_flags` is meaningful on the SPARC architecture only.

flags

Used to determine the data transfer mode and/or the cache attribute.

Possible values of the data transfer 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 The `ddi_dma_mem_alloc()` function 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.

The *flags* parameter 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.

The *flags* parameter 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.

Return Values The `ddi_dma_mem_alloc()` function returns:

`DDI_SUCCESS`	Memory successfully allocated.
`DDI_FAILURE`	Memory allocation failed.

Context The `ddi_dma_mem_alloc()` function can be called from user, interrupt, or kernel context except when *waitfp* is set to `DDI_DMA_SLEEP`, in which case it cannot be called from interrupt context.

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.

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

Name ddi_dma_movwin – shift current DMA window

Synopsis

```
#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 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 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.
lenp	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.
cookiep	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.

Description The `ddi_dma_movwin()` function 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.

The `ddi_dma_movwin()` function 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 The `ddi_dma_movwin()` function returns:

`DDI_SUCCESS`	The current length and offset are legal and have been set.
`DDI_FAILURE`	Otherwise.

Context The `ddi_dma_movwin()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

```
#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.

```
/* 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 */
    }
}
```

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

Name

ddi_dma_nextseg – get next DMA segment

Synopsis

```
#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

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

Parameters

win	A DMA window.
seg	The current DMA segment or `NULL`.
nseg	A pointer to the next DMA segment to be filled in. If *seg* is `NULL`, a pointer to the first segment within the specified window is returned.

Description

The `ddi_dma_nextseg()` function 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

The `ddi_dma_nextseg()` function 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

The `ddi_dma_nextseg()` function can be called from user, interrupt, or kernel context.

Examples

For an example, see `ddi_dma_segtocookie`(9F).

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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_x86`(9S), `ddi_dma_req`(9S)

Writing Device Drivers

Name ddi_dma_nextwin – get next DMA window

Synopsis

```
#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);
```

Interface Level This interface is obsolete. `ddi_dma_getwin`(9F) should be used instead.

Parameters

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.

Description The `ddi_dma_nextwin()` function 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`(9F) or `ddi_dma_addr_setup`(9F) or as part of a `ddi_dma_req`(9S) structure in a call to `ddi_dma_setup`(9F).

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.

The `ddi_dma_nextwin()` function 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`(9F) are required.

Return Values The `ddi_dma_nextwin()` function 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.

Context The `ddi_dma_nextwin()` function can be called from user, interrupt, or kernel context.

Examples For an example see `ddi_dma_segtocookie`(9F).

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

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

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

Parameters

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

The `ddi_dma_segtocookie()` function 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

The `ddi_dma_segtocookie()` function returns:

`DDI_SUCCESS`	Successfully filled in all values.
`DDI_FAILURE`	Failed to successfully fill in all values.

Context

The `ddi_dma_segtocookie()` function can be called from user, interrupt, or kernel 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 */
```

EXAMPLE 1 `ddi_dma_segtocookie()` example *(Continued)*

```
            }
        }
    }
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

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 Level Solaris DDI specific (Solaris DDI).

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 64–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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	SBus

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.

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 The `ddi_dma_setup()` function 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 allocated DMA resources by calling `ddi_dma_free()`

Return Values The `ddi_dma_setup()` function 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 The `ddi_dma_setup()` function can be called from user, interrupt, or kernel 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 cannot be called from interrupt context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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.

Name ddi_dma_sync – synchronize CPU and I/O views of memory

Synopsis

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



int ddi_dma_sync(ddi_dma_handle_t handle, off_t offset,
     size_t length, uint_t type);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

handle	The *handle* filled in by a call to `ddi_dma_alloc_handle`(9F).
offset	The offset into the object described by the *handle*.
length	The length, in bytes, of the area to synchronize. When *length* is zero, the entire range starting from *offset* to the end of the object has the requested operation applied to it.
type	Indicates the caller's desire about what view of the memory object to synchronize. The possible values are `DDI_DMA_SYNC_FORDEV`, `DDI_DMA_SYNC_FORCPU` and `DDI_DMA_SYNC_FORKERNEL`.

Description The `ddi_dma_sync()` function 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 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 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`(9F), 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)).

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 The `ddi_dma_sync()` function 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 The `ddi_dma_sync()` function can be called from user, interrupt, or kernel 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

```
#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

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

Name ddi_driver_name – return normalized driver name

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.

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(unsignedint 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 The `ddi_enter_critical()` function 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, interrupt, or kernel 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

Name ddi_ffs, ddi_fls – find first (last) bit set in a long integer

Synopsis

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



intddi_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, interrupt, or kernel context.

See Also *Writing Device Drivers*

Name ddi_fm_acc_err_clear, ddi_fm_dma_err_clear – clear the error status for an access or DMA handle

Synopsis

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

void ddi_fm_acc_err_clear(ddi_acc_handle_t acc_handle,
     int version);

void ddi_fm_dma_err_clear(ddi_dma_handle_t dma_handle,
     int version);
```

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 to a similar function.
dma_handle	DMA handle obtained from a previous call to `ddi_dma_mem_alloc`(9F) or one of its derivatives.
version	Version number of `ddi_fm_error_t`.

Description The `ddi_fm_dma_err_clear()` and `ddi_fm_acc_err_clear()` functions clear the error status of a DMA or access handle respectively.

Once cleared, the driver is again able to access the mapped registers or memory using programmed I/O through the handle.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes`(5), `ddi_dma_mem_alloc`(9F), `ddi_fm_acc_err_get`(9F), `ddi_fm_dma_err_get`(9F), `ddi_regs_map_setup`(9F)

Writing Device Drivers

Name

ddi_fm_acc_err_get, ddi_fm_dma_err_get – get the error status for an access or DMA handle

Synopsis

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

void ddi_fm_acc_err_get(ddi_acc_handle_t acc_handle,
     ddi_fm_error_t *error_status, int version);

void ddi_fm_dma_err_get(ddi_dma_handle_t dma_handle,
     ddi_fm_error_t *error_status, int version);
```

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 to a similar function.
dma_handle	DMA handle obtained from a previous call to `ddi_dma_mem_alloc`(9F) or one of its derivatives.
error_status	Pointer to where the error status for the access or DMA handle should be returned.
version	Version number of `ddi_fm_error_t`. The driver should always set this to `DDI_FME_VERSION`.

Description

The `ddi_fm_dma_err_get()` and `ddi_fm_acc_err_get()` functions return the error status for a DMA or access handle respectively. If a fault has occurred that affects the resource mapped by the supplied handle, the supplied *error_status* structure is updated to reflect error information captured during error handling by a bus or other device driver in the I/O data path.

If an error is indicated for an access handle, the driver might no longer be able to access the mapped registers or memory using programmed I/O through the handle. Typically, this might occur after the device has failed to respond to an I/O access – in the case of a bus error, for instance, or a timeout. The effect of programmed I/O access made at the time of a fault is undefined. Read access via `ddi_get8`(9F), for example, can return random values, while write access via `ddi_put8`(9F) might or might not have an effect. It is possible, however, that the error might be transient. In that case, the driver can attempt to recover by calling `ddi_fm_acc_err_clear()`, resetting the device to return it to a known state, then retrying any potentially failed transactions.

If an error is indicated for a DMA handle, it implies that an error has been detected that has or will affect DMA transactions between the device and the memory currently bound to the handle – or the memory 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 contents of any memory currently or previously bound to the handle should be considered indeterminate. The driver might be able to continue by freeing memory that is bound to the handle back to the system, resetting the device to return it to a known state, then retrying any potentially failed transactions.

If the driver is unable to recover, the operating state should be changed by a call to `ddi_fm_service_impact()` that specifies `DDI_SERVICE_LOST` for the impacted device instance. If the recovery and retry succeed, a call should still be made to `ddi_fm_service_impact()` but `DDI_SERVICE_UNAFFECTED` should be specified.

Context The `ddi_fm_acc_err_get()` and `ddi_fm_dma_err_get()` functions can be called from user, kernel, or high-level interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes(5)`, `ddi_dma_mem_alloc(9F)`, `ddi_fm_acc_err_clear(9F)`, `ddi_fm_service_impact(9F)`, `ddi_get8(9F)`, `ddi_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_fm_error(9S)`,

Writing Device Drivers

Name ddi_fm_ereport_post – post an FMA Protocol Error Report Event

Synopsis

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

void ddi_fm_ereport_post(dev_info_t *dip, char *ereport_class,
     uint64_t ena, int *sflag, ... /* name-value pair args */);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the `dev_info` structure
ereport_class	FMA Event Protocol error class
ena	Error Numeric Association
sflag	Determines whether caller can sleep for memory or other event resources.

Description The `ddi_fm_ereport_post()` function causes an encoded fault management error report name-value pair list to be queued for delivery to the Fault Manager daemon, `fmd`(1M). The *sflag* parameter indicates whether or not the caller is willing to wait for system memory and event channel resources to become available.

The following *ereport_class* strings are available for use by any leaf device driver:

`device.inval_state`	A leaf driver discovers that the device is in an invalid or inconsistent state. For example, the driver might detect that receive or send ring descriptor indices are corrupted. It might also find an invalid value in a register or a driver-to-device protocol violation.
`device.no_response`	A leaf driver times out waiting for a response from the device. For example, timeouts can occur when no confirmation is seen after resetting, enabling, or disabling part of the device.
`device.badint_limit`	A leaf device sends too many consecutive interrupts with no work to do.
`device.intern_corr`	A leaf device reports to the driver that it has itself detected an internal correctable error.
`device.intern_uncorr`	A leaf device reports to the driver that it has itself detected an internal uncorrectable error.
`device.stall`	A leaf driver determines that data transmission has stalled indefinitely.

The *ena* indicates the Format 1 Error Numeric Association for this error report. It might have already been initialized by another error-detecting software module. For example, if `ddi_fm_ereport_post()` is called from an error handler callback function, the *fme_ena* field from the passed-in `ddi_fm_error` argument should be used. Otherwise it should be set to `0` and will be initialized by `ddi_fm_ereport_post()`.

The name-value pair *args* variable argument list contains one or more (names, type, value pointer) nvpair tuples for non-array `data_type_t` types or one or more (name, type, number of elements, value pointer) tuples for `data_type_t` array types. There is one mandatory tuple to describe the `ereport` version. This should contain the following values:

name - `FM_VERSION`
type - `DATA_TYPE_UINT8`
value - `FM_EREPORT_VERS0`

Additional nvpair tuples can describe error conditions for logging purposes, but are not interpreted by the `I/O` framework or fault manager. The end of the argument list is specified by `NULL`.

Context

The `ddi_fm_ereport_post()` function can be called from user, kernel, or high-level interrupt context.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`fmd(1M)`, `attributes(5)`, `ddi_fm_service_impact(9F)`

Name

ddi_fm_handler_register, ddi_fm_handler_unregister – register or unregister an error handling callback

Synopsis

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

void ddi_fm_handler_register(dev_info_t *dip,
     ddi_err_func_t error_handler, void *impl_data);

void ddi_fm_handler_unregister(dev_info_t *dip);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the `dev_info` structure
error_handler	Pointer to an error handler callback function
impl_data	Pointer to private data for use by the caller

Description

The `ddi_fm_handler_register()` function registers an error handler callback routine with the I/O Fault Management framework. The error handler callback, *error_handler*, is called to process error conditions detected by the system. In addition to its device instance, *dip*, the error handler is called with a pointer to a fault management error status structure, `ddi_fm_error_t`. For example:

```
int (*ddi_err_func_t)(dev_info_t *dip, ddi_fm_error_t *error_status);
```

A driver error handling callback is passed the following arguments:

- a pointer to the device instance registered for this callback.
- a data structure containing common fault management data and status for error handling.

The primary responsibilities of the error handler include:

- to check for outstanding hardware or software errors.
- where possible, to isolate the device that might have caused the errors.
- to report errors that were detected.

During the invocation of an error handler, a device driver might need to quiesce or suspend all I/O activities in order to check for error conditions or status in:

- hardware control and status registers.
- outstanding I/O transactions.
- access or DMA handles.

For each error detected, the driver must formulate and post an error report via `ddi_fm_ereport_post()` for problem analysis by the Solaris Fault Manager `fmd`(1M).

For a PCI, `PCI/X`, or `PCI Express` leaf device, the `pci_ereport_post()` function is provided to carry out reporting responsibilities on behalf of the driver. In many cases, an error handler callback function of the following form can be used:

```
xxx_err_cb(dev_info_t *dip, ddi_fm_error_t *errp) {
      pci_ereport_post(dip, errp, NULL);
      return (errp->fme_status);
}
```

In addition, the driver might be able to carry out further device specific checks within the error handler.

Error handlers can be called from kernel, interrupt, or high-level interrupt context. The interrupt block cookie returned from `ddi_fm_init()` should be used to allocate and initialize any synchronization variables and locks that might be used within the error handler callback function. Such locks may not be held by the driver when a device register is accessed with functions such as `ddi_get8`(9F) and `ddi_put8`(9F).

The data structure, `ddi_fm_error_t`, contains an FMA protocol (format 1) ENA for the current error propagation chain, the status of the error handler callback, an error expectation flag, and any potential access or DMA handles associated with an error detected by the parent nexus.

The `ddi_fm_handler_unregister()` function removes a previously registered error handling callback for the device instance specified by the *dip*.

Context The `ddi_fm_handler_register()` and `ddi_fm_handler_unregister()` functions must be called from kernel context in an `attach`(9E) or `detach`(9E) entry point. The registered error handler, *error_handler*, callback can be called from kernel, interrupt, or high level interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `fmd`(1M), `attributes`(5), `attach`(9E), `detach`(9E), `ddi_fm_ereport_post`(9F), `ddi_fm_init`(9F), `ddi_get8`(9F), `ddi_put8`(9F), `pci_ereport_post`(9F), `ddi_fm_error`(9S)

Writing Device Drivers

Name

ddi_fm_init, ddi_fm_fini, ddi_fm_capable – initialize and get the FM capabilities for a device instance

Synopsis

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

void ddi_fm_init(dev_info_t *dip, int *fm_capability,
     ddi_iblock_cookie_t *ibcp);

void ddi_fm_fini(dev_info_t *dip);

int ddi_fm_capable(dev_info_t *dip, int *fm_capability);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

`ddi_fm_init()`

dip	Pointer to the `dev_info` structure
fm_capability	Fault Management capability bit mask
ibcp	Pointer to where the interrupt block cookie should be returned.

Description

A device driver can declare its fault management capabilities to the `I/O` Fault Management framework by calling `ddi_fm_init()`. The `ddi_fm_init()` function allocates and initializes resources according to the bitwise-inclusive-OR of the fault management capabilities, described in the following and supported by the driver's immediate nexus parent.

`DDI_FM_NOT_CAPABLE`	The driver does not support any FMA features. This is the default value assigned to device drivers.
`DDI_FM_EREPORT_CAPABLE`	The driver generates FMA protocol error events (`ereports`) upon the detection of an error condition.
`DDI_FM_ACCCHK_CAPABLE`	The driver checks for errors upon the completion of one or more access `I/O` transactions.
`DDI_FM_DMACHK_CAPABLE`	The driver checks for errors upon the completion of one or more DMA `I/O` transactions.
`DDI_FM_ERRCB_CAPABLE`	The driver is capable of error handler callback registration.

If the parent nexus is not capable of supporting any one of the requested capabilities, the associated bit will not be set and returned as such to the driver. Before returning from `ddi_fm_init()`, the `I/O` Fault Management framework creates a set of fault management capability properties: `fm-ereport-capable`, `fm-errcb-capable`, `fm-accchk-capable`, and `fm-dmachk-capable`. The current supported fault management capability levels are observable via `prtconf`(1M).

A driver can support the administrative selection of fault management capabilities by exporting and setting a fault management capability level property in its `driver.conf`(4) file to the values described above. The `fm_capable` properties must be set and read prior to calling `ddi_fm_init()` with the desired capability list.

`ddi_fm_fini()`	This function cleans up resources allocated to support fault management for the *dip* structure.
`ddi_fm_capable()`	This function returns the capability bit mask currently set for the *dip* structure.

Context These functions can be called from kernel context in a driver `attach`(9E) or `detach`(9E) operation.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `prtconf`(1M), `driver.conf`(4), `attributes`(5), `attach`(9E), `detach`(9E)

Writing Device Drivers

Name ddi_fm_service_impact – report the impact of an error

Synopsis

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

void ddi_fm_service_impact(dev_info_t *dip, int *impact);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the `dev_info` structure
impact	Impact of error

Description The following service impact values are accepted by `ddi_fm_service_impact()`:

`DDI_SERVICE_LOST`	The service provided by the device is unavailable due to an error. The operational state of the device will transition to `DEVI_DEVICE_DOWN`.
`DDI_SERVICE_DEGRADED`	The driver is unable to provide normal service, but it can provide a partial or degraded level of service. The driver might 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. The operational state of the device will transition to `DEVI_DEVICE_DEGRADED`.
`DDI_SERVICE_RESTORED`	The service provided by the device has been restored. The operational state of the device will transition to its pre-error condition state and `DEVI_DEVICE_DOWN` or `DEVI_DEVICE_DEGRADED` is removed.
`DDI_SERVICE_UNAFFECTED`	The service provided by the device was unaffected by the error.

Context The `ddi_fm_service_impact()` function can be called from user, kernel, or high-level interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `fmd(1M)`, `attributes(5)`, `ddi_fm_ereport_post(9F)`, `pci_ereport_post(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 and SBus buses. See `sysbus`(4), `isa`(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:

Previous Name	New Name
`ddi_getb`	`ddi_get8`
`ddi_getw`	`ddi_get16`
`ddi_getl`	`ddi_get32`
`ddi_getll`	`ddi_get64`

Name ddi_get_cred – returns a pointer to the credential structure of the caller

Synopsis

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



cred_t *ddi_get_cred(void)
```

Interface Level Solaris DDI specific (Solaris DDI).

Description `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*

Name 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_DEVSTATE_QUIESCED`	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_DEGRADED`	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)

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



void ddi_set_driver_private(dev_info_t *dip, void *data);

void *ddi_get_driver_private(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_get_driver_private()`

dip	Pointer to device information structure to get from.

`ddi_set_driver_private()`

dip	Pointer to device information structure to set.
data	Data area address to set.

Description The `ddi_get_driver_private()` function returns the address of the device's private data area from the device information structure pointed to by *dip*.

The `ddi_set_driver_private()` function 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 The `ddi_get_driver_private()` function 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 , interrupt, or kernel context.

See Also *Writing Device Drivers*

Name

ddi_get_eventcookie – retrieve a NDI event service cookie handle

Synopsis

```
#include <sys/dditypes.h>
#include <sys/sunddi.h>

int ddi_get_eventcookie(dev_info_t *dip, char *name,
     ddi_eventcookie_t *event_cookiep);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

`dev_info_t` *`*dip`*	Child device node requesting the cookie.
`char` *`*name`*	NULL-terminated string containing the name of the event.
`ddi_eventcookie_t` *`*event_cookiep`*	Pointer to cookie where event cookie will be returned.

Description

The `ddi_get_eventcookie()` function queries the device tree for a cookie matching the given event name and returns a reference to that cookie. The search is performed by a calling up the device tree hierarchy until the request is satisfied by a bus nexus driver, or the top of the `dev_info` tree is reached.

The cookie returned by this function can be used to register a callback handler, unregister a callback handler, or post an event.

Return Values

`DDI_SUCCESS`	Cookie handle is returned.
`DDI_FAILURE`	Request was not serviceable by any nexus driver in the driver's ancestral device tree hierarchy.

Context

The `ddi_get_eventcookie()` function can be called from user and kernel contexts only.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Evolving

See Also

`attributes(5)`, `ddi_add_event_handler(9F)`, `ddi_remove_event_handler(9F)`

Writing Device Drivers

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 and SBus buses. See `sysbus`(4), `isa`(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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

See Also `attributes`(5), `getmajor`(9F), `getminor`(9F), `makedevice`(9F)

Writing Device Drivers

Warnings Drivers are required to replace calls to `ddi_getminor.9f` by `getminor`(9F)) in order to compile under Solaris 10 and later versions.

Name ddi_get_instance – get device instance number

Synopsis

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



int ddi_get_instance(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *dip* Pointer to `dev_info` structure.

Description The `ddi_get_instance()` function 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).

Return Values The `ddi_get_instance()` function returns the instance number of the device corresponding to *dip*.

Context The `ddi_get_instance()` function can be called from user, interrupt, or kernel context.

See Also `path_to_inst`(4)

Writing Device Drivers

Name ddi_get_kt_did – get identifier of current thread

Synopsis

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



kt_did_t ddi_get_kt_did(void);
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `ddi_get_kt_did()` function returns a unique 64-bit identifier for the currently running thread.

Context This routine can be called from user, kernel, or interrupt context. This routine cannot be called from a high-level interrupt context.

Return Values `ddi_get_kt_did()` always returns the identifier for the current thread. There are no error conditions.

See Also *Writing Device Drivers*

Notes 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.

Name ddi_get_lbolt, ddi_get_lbolt64 – return the number of clock ticks since boot

Synopsis

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

clock_t ddi_get_lbolt(void);

int64_t ddi_get_lbolt64(void);
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `ddi_get_lbolt()` function returns a value that represents the number of clock ticks since the system booted. 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.

The `ddi_get_lbolt64()` behaves essentially the same as `ddi_get_lbolt()`, except the value is returned in a longer data type (`int64_t`) that will not wrap for 2.9 billion years.

Return Values The `ddi_get_lbolt()` function returns the number of clock ticks since boot in a `clock_t` type.

The `ddi_get_lbolt64()` function returns the number of clock ticks since boot in a `int64_t` type.

Context These routines can be called from any context.

See Also `ddi_get_time`(9F), `drv_getparm`(9F), `drv_usectohz`(9F)

Writing Device Drivers

STREAMS Programming Guide

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 The `ddi_get_parent()` function returns a pointer to the device information structure which is the parent of the one pointed to by *dip*.

Return Values The `ddi_get_parent()` function returns a pointer to a device information structure.

Context The `ddi_get_parent()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

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

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 Level Solaris DDI specific (Solaris DDI).

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

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 Level Solaris DDI specific (Solaris DDI).

Description 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.

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

Name ddi_intr_add_handler, ddi_intr_remove_handler – add or remove interrupt handler

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>



int ddi_intr_add_handler(ddi_intr_handle_t *h,
     ddi_intr_handler_t inthandler, void *arg1,
      void *arg2);

int ddi_intr_remove_handler(ddi_intr_handle_t h);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_add_handler()`

h	Pointer to the DDI interrupt handle
inthandler	Pointer to interrupt handler
arg1	First argument for the interrupt handler
arg2	Second, optional, argument for the interrupt handler

`ddi_intr_remove_handler()`

h	DDI interrupt handle

Description The `ddi_intr_add_handler()` function adds an interrupt handler given by the *inthandler* argument to the system with the handler arguments *arg1* and *arg2* for the previously allocated interrupt handle specified by the *h* pointer. The arguments *arg1* and *arg2* are passed as the first and second arguments, respectively, to the interrupt handler *inthandler*. See `<sys/ddi_intr.h>` for the definition of the interrupt handler.

The routine *inthandler* with the arguments *arg1* and *arg2* is called upon receipt of the appropriate interrupt. The interrupt handler should return `DDI_INTR_CLAIMED` if the interrupt is claimed and `DDI_INTR_UNCLAIMED` otherwise.

The `ddi_intr_add_handler()` function must be called after `ddi_intr_alloc()`, but before `ddi_intr_enable()` is called. The interrupt must be enabled through `ddi_intr_enable()` or `ddi_intr_block_enable()` before it can be used.

The `ddi_intr_remove_handler()` function removes the handler association, added previously with `ddi_intr_add_handler()`, for the interrupt identified by the interrupt handle *h* argument. Unloadable drivers should call this routine during their `detach(9E)` routine to remove the interrupt handler from the system.

The `ddi_intr_remove_handler()` function is used to disassociate the handler after the interrupt is disabled to remove *dup-ed* interrupt handles. See `ddi_intr_dup_handler`(9F) for *dup-ed* interrupt handles. If a handler is duplicated with the `ddi_intr_dup_handler()` function, all added and duplicated instances of the handler must be removed with `ddi_intr_remove_handler()` in order for the handler to be completely removed.

Return Values

The `ddi_intr_add_handler()` and `ddi_intr_remove_handler()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.

Context

The `ddi_intr_add_handler()` and `ddi_intr_remove_handler()` functions can be called from kernel non-interrupt context.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

`attributes`(5), `attach`(9E), `detach`(9E), `ddi_intr_alloc`(9F), `ddi_intr_block_enable`(9F), `ddi_intr_disable`(9F), `ddi_intr_dup_handler`(9F), `ddi_intr_enable`(9F), `ddi_intr_free`(9F), `ddi_intr_get_supported_types`(9F), `mutex`(9F), `mutex_init`(9F), `rw_init`(9F), `rwlock`(9F)

Writing Device Drivers

Notes

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses `MSI` and `MSI-X` interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any `MSI` and `MSI-X` interrupt usage settings that have been applied.

The second argument, *arg2*, is optional. Device drivers are free to use the two arguments however they see fit. There is no officially recommended model or restrictions. For example, an interrupt handler may wish to use the first argument as the pointer to its softstate and the second argument as the value of the MSI vector.

Name ddi_intr_add_softint, ddi_intr_remove_softint, ddi_intr_trigger_softint, ddi_intr_get_softint_pri, ddi_intr_set_softint_pri – software interrupt handling routines

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>



int ddi_intr_add_softint(dev_info_t *dip,
     ddi_softint_handle_t *h, int soft_pri,
     ddi_intr_handler_t handler, void *arg1);

int ddi_intr_trigger_softint(ddi_softint_handle_t h,
      void *arg2);

int ddi_intr_remove_softint(ddi_softint_handle_t h);

int ddi_intr_get_softint_pri(ddi_softint_handle_t h,
      uint *soft_prip);

int ddi_intr_set_softint_pri(ddi_softint_handle_t h,
      uint soft_pri);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_add_softint()`

dip	Pointer to a `dev_info` structure
h	Pointer to the DDI soft interrupt handle
soft_pri	Priority to associate with a soft interrupt
handler	Pointer to soft interrupt handler
arg1	Argument for the soft interrupt handler

`ddi_intr_trigger_softint()`

h	DDI soft interrupt handle
arg2	Additional argument for the soft interrupt handler

`ddi_intr_remove_softint()`

h	DDI soft interrupt handle

`ddi_intr_get_softint_pri()`

h	DDI soft interrupt handle
soft_prip	Soft interrupt priority of the handle

`ddi_intr_set_softint_pri()`

h	DDI soft interrupt handle
soft_prip	Soft interrupt priority of the handle

Description The `ddi_intr_add_softint()` function adds the soft interrupt handler given by the *handler* argument *arg1*. The *handler* runs at the soft interrupt priority given by the *soft_pri* argument.

The value returned in the location pointed at by *h* is the soft interrupt handle. This value is used in later calls to `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()` and `ddi_intr_set_softint_pri()`.

The software priority argument *soft_pri* is a relative priority value within the range of `DDI_INTR_SOFTPRI_MIN` and `DDI_INTR_SOFTPRI_MAX`. If the driver does not know what priority to use, the default *soft_pri* value of `DDI_INTR_SOFTPRI_DEFAULT` could be specified. The default value is the lowest possible soft interrupt priority value.

The *soft_pri* argument contains the value needed to initialize the lock associated with a soft interrupt. See `mutex_init(9F)` and `rw_init(9F)`. The handler cannot be triggered until the lock is initiatized.

The `ddi_intr_remove_softint()` function removes the handler for the soft interrupt identified by the interrupt handle *h* argument. Once removed, the soft interrupt can no longer be triggered, although any trigger calls in progress can still be delivered to the handler.

Drivers must remove any soft interrupt handlers before allowing the system to unload the driver. Otherwise, kernel resource leaks might occur.

The `ddi_intr_trigger_softint()` function triggers the soft interrupt specified by the interrupt handler *h* argument. A driver may optionally specify an additional argument *arg2* that is passed to the soft interrupt handler. Subsequent `ddi_intr_trigger_softint()` events, along with *arg2*, will be dropped until the one pending is serviced and returns the error code `DDI_EPENDING`.

The routine *handler*, with the *arg1* and *arg2* arguments, is called upon the receipt of a software interrupt. These were registered through a prior call to `ddi_intr_add_softint()`. Software interrupt handlers must not assume that they have work to do when they run. Like hardware interrupt handlers, they may run because a soft interrupt has occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. Before triggering the soft interrupt, the driver must indicate to the soft interrupt handler that it has work to do. This is usually done by setting a flag in the state structure. The routine *handler* checks this flag, reached through *arg1* and *arg2*, 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 and `DDI_INTR_UNCLAIMED` otherwise.

The `ddi_intr_get_softint_pri()` function retrieves the soft interrupt priority, a small integer value, associated with the soft interrupt handle. The handle is defined by the *h* argument, and the priority returned is in the value of the integer pointed to by the *soft_prip* argument.

Return Values

The `ddi_intr_add_softint()`, `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()`, `ddi_intr_get_softint_pri()`, `ddi_intr_set_softint_pri()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EAGAIN`	On encountering internal error regarding currently unavailable resources.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.
`DDI_EPENDING`	On encountering a previously triggered softint event that is pending.

Context

The `ddi_intr_add_softint()`, `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()`, `ddi_intr_get_softint_pri()`, `ddi_intr_set_softint_pri()` functions can be called from either user or kernel non-interrupt context.

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_get_hilevel_pri(9F)`.

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_intr_handle_t       int_hdl;
  int                     high_pri;
  kmutex_t                high_mutex;
```

EXAMPLE 1 Device using high-level interrupts *(Continued)*

```
  ddi_softint_handle_t    soft_hdl;
  int                     low_soft_pri;
  kmutex_t                low_mutex;
  int                     softint_running;
  ...
};

struct xxstate *xsp;
static uint_t xxsoftint_handler(void *, void *);
static uint_t xxhighintr(void *, void *);
...
```

EXAMPLE 2 Sample `attach()` routine

The following code fragment would usually appear in the driver's `attach(9E)` routine. `ddi_intr_add_handler(9F)` is used to add the high-level interrupt handler and `ddi_intr_add_softint()` is used to add the low-level interrupt routine.

```
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
   int              types;
   int              *actual;
   int              nintrs;
   struct xxstate   *xsp;
   ...

   (void) ddi_intr_get_supported_types(dip, &types);
   (void) ddi_intr_get_nintrs(dip< DDI_INTR_TYPE_FIXED, *nintrs);
   (void) ddi_intr_alloc(dip, &xsp->int_hdl, DDI_INTR_TYPE_FIXED,
       1, nintrs, *actual, 0);

   /* initialize high-level mutex */
   (void) ddi_intr_get_pri(xsp->int_hdl, &>high_pri);
   mutex_init(&xsp->high_mutex, NULL, MUTEX_DRIVER,
       DDI_INTR_PRI(xsp->high_pri));

   /* Ensure that this is a hi-level interrupt */
   if (ddi_intr_get_hilevel_pri(h) != DDI_SUCCESS) {
           /* cleanup */
           return (DDI_FAILURE); /* fail attach */
   }

   /* add high-level routine - xxhighintr() */
   if (ddi_intr_add_handler(xsp->int_hdl, xxhighintr,
```

EXAMPLE 2 Sample `attach()` routine *(Continued)*

```
        arg1, NULL) != DDI_SUCCESS) {
            /* cleanup */
            return (DDI_FAILURE); /* fail attach */
   }

   /* Enable high-level routine - xxhighintr() */
   if (ddi_intr_enable(xsp->int_hdl) != DDI_SUCCESS) {
            /* cleanup */
            return (DDI_FAILURE); /* fail attach */
   }

   /* Enable soft interrupts */
   xsp->low_soft_pri = DDI_INTR_SOFTPRI_MIN;
   if (ddi_intr_add_softint(dip, &xsp>soft_hdl,
       xsp->low_soft_pri, xxsoftint_handler, arg1) != DDI_SUCCESS) {
           /* clean up */
           return (DDI_FAILURE); /* fail attach */
   }

   /* initialize low-level mutex */
   mutex_init(&xsp->low_mutex, NULL, MUTEX_DRIVER,
       DDI_INTR_PRI(xsp->low_soft_pri));

   ...
}
```

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_intr_trigger_softint()` 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(void *arg1, void *arg2)
{
   struct xxstate *xsp = (struct xxstate *)arg1;
   int need_softint;
   ...
   mutex_enter(&xsp->high_mutex);
   /*
   * Verify this device generated the interrupt
   * and disable the device interrupt.
   * Enqueue data for xxsoftint_handler() processing.
```

EXAMPLE 3 High-level interrupt routine *(Continued)*

```
   */

   /* is xxsoftint_handler() already running ? */
   need_softint = (xsp->softint_running) ? 0 : 1;
   mutex_exit(&xsp->high_mutex);

   /* read-only access to xsp->id, no mutex needed */
   if (xsp->soft_hdl && need_softint)
           ddi_intr_trigger_softint(xsp->soft_hdl, arg2);
   ...
   return (DDI_INTR_CLAIMED);
}


static uint_t
xxsoftint_handler(void *arg1, void *arg2)
{
   struct xxstate *xsp = (struct xxstate *)arg1;
   ...
   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);
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes(5)`, `attach(9E)`, `ddi_intr_alloc(9F)`, `ddi_intr_free(9F)`, `ddi_intr_get_hilevel_pri(9F)`, `mutex_init(9F)`, `rw_init(9F)`, `rwlock(9F)`

Writing Device Drivers

Notes Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

The `ddi_intr_add_softint()` 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 *arg1* in each of the calls to `ddi_intr_add_softint()`. 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 soft state structure of the device that could contain a `which_service` field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_intr_trigger_softint()`.

Every time a modifiable valid second argument, *arg2*, is provided when `ddi_intr_trigger_softint()` is invoked, the DDI framework saves *arg2* internally and passes it to the interrupt handler *handler*.

A call to `ddi_intr_set_softint_pri()` could fail if a previously scheduled soft interrupt trigger is still pending.

Name ddi_intr_alloc, ddi_intr_free – allocate or free interrupts for a given interrupt type

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_alloc(dev_info_t *dip, ddi_intr_handle_t *h_array, int type,
     int inum, int count, int *actualp, int behavior);

int ddi_intr_free(ddi_intr_handle_t h);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_alloc()`

dip	Pointer to the `dev_info` structure
h_array	Pointer to an array of DDI interrupt handles
type	Interrupt type
inum	Interrupt number
count	Number of interrupts requested. The *count* should not exceed the total number of interrupts supported by the device, as returned by a call to `ddi_intr_get_nintrs`(9F).
actualp	Pointer to the number of interrupts actually allocated
behavior	Flag to determine the allocation algorithm

`ddi_intr_free()`

h	DDI interrupt handle

Description The `ddi_intr_alloc()` function allocates interrupts of the interrupt type given by the *type* argument beginning at the interrupt number *inum*. If `ddi_intr_alloc()` allocates any interrupts, it returns the actual number of interrupts allocated in the integer pointed to by the *actualp* argument and returns the number of interrupt handles in the interrupt handle array pointed to by the *h_array* argument.

Specific interrupts are always specified by the combination of interrupt *type* and *inum*. For legacy devices, *inum* refers to the nth interrupt, typically as defined by the devices `interrupts` property. For PCI fixed interrupts, *inum* refers to the interrupt number. The *inum* is the relative interrupt vector number, from `0` to `31` for MSI, from `0` to `2047` for MSI-X. The first interrupt vector is `0`. The last relative vector is `31` for MSI or `2047` for MSI-X.

The *h_array* must be pre-allocated by the caller as a *count* sized array of ddi_intr_handle_t's.

If MSI interrupts are being allocated, the *count* argument passed should be a number between `1` and `32`, specified as a power of two. If *count* is not specified as a power of two, the error `DDI_EINVAL` is returned.

The behavior flag controls the interrupt allocation algorithm. It takes one of two input values: `DDI_INTR_ALLOC_NORMAL` or `DDI_INTR_ALLOC_STRICT`. If the *count* value used is greater than `NINTRs`, then the call fails with `DDI_EINVAL` unconditionally. When set to `DDI_INTR_ALLOC_STRICT`, the call succeeds if and only if *count* interrupts are allocated. Otherwise, the call fails, and the number of available interrupts is returned in *actualp*. When set to `DDI_INTR_ALLOC_NORMAL`, the call succeeds if at least one interrupt is allocated, and the number of allocated interrupts is returned in actualp.

The handle for each allocated interrupt, if any, is returned in the array of handles given by the *h_array* argument.

The `ddi_intr_free()` function releases the system resources and interrupt vectors associated with the ddi_intr_handle_t *h*, including any resources associated with the handle *h* itself. Once freed, the handle *h* should not be used in any further calls.

The `ddi_intr_free()` function should be called once for each handle in the handle array.

Return Values

The `ddi_intr_alloc()` and `ddi_intr_free()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EAGAIN`	Not enough interrupt resources.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_INTR_NOTFOUND`	On failure to find the interrupt.
`DDI_FAILURE`	On any implementation specific failure.

Context

The `ddi_intr_alloc()` and `ddi_intr_free()` functions can be called from kernel non-interrupt context.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`attributes(5)`, `ddi_intr_add_handler(9F)`, `ddi_intr_block_enable(9F)`, `ddi_intr_disable(9F)`, `ddi_intr_enable(9F)`, `ddi_intr_get_cap(9F)`, `ddi_intr_get_nintrs(9F)`, `ddi_intr_get_pri(9F)`, `ddi_intr_get_supported_types(9F)`, `ddi_intr_remove_handler(9F)`

Writing Device Drivers

Notes Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses `MSI` and `MSI-X` interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any `MSI` and `MSI-X` interrupt usage settings that have been applied.

Name ddi_intr_dup_handler – reuse interrupt handler and arguments for MSI-X interrupts

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_dup_handler(ddi_intr_handle_t primary, int vector,
     ddi_intr_handle_t *new);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

primary	Original DDI interrupt handle
vector	Interrupt number to duplicate
new	Pointer to new DDI interrupt handle

Description The `ddi_intr_dup_handler()` function is a feature for MSI-X interrupts that allows an unallocated interrupt vector of a device to use a previously initialized or added primary MSI-X interrupt vector in order to share the same vector address, vector data, interrupt handler, and handler arguments. This feature allows a driver to alias the resources provided by the Solaris Operating System to the unallocated interrupt vectors on an associated device. For example, if 2 MSI-X interrupts were allocated to a driver and 32 interrupts were supported on the device, the driver could alias the 2 interrupts it received to the 30 remaining on the device.

The `ddi_intr_dup_handler()` function must be called after the primary interrupt handle has been added to the system or enabled by `ddi_intr_add_handler`(9F) and `ddi_intr_enable`(9F) calls, respectively. If successful, the function returns the new interrupt handle for a given vector in the *new* argument passed to the function. The new interrupt handle must not have been previously allocated with `ddi_intr_alloc`(9F). Otherwise, the `ddi_intr_dup_handler()` call will fail.

The only supported calls on *dup-ed* interrupt handles are `ddi_intr_set_mask`(9F), `ddi_intr_clr_mask`(9F), `ddi_intr_get_pending`(9F), `ddi_intr_enable`(9F), `ddi_intr_disable`(9F), and `ddi_intr_free`(9F).

A call to `ddi_intr_dup_handler()` does not imply that the interrupt source is automatically enabled. Initially, the dup-ed handle is in the disabled state and must be enabled before it can be used by calling `ddi_intr_enable()`. Likewise, `ddi_intr_disable()` must be called to disable the enabled dup-ed interrupt source.

A dup-ed interrupt is removed by calling `ddi_intr_free()` after it has been disabled. The `ddi_intr_remove_handler`(9F) call is not required for a dup-ed handle.

Before removing the original MSI-X interrupt handler, all dup-ed interrupt handlers associated with this MSI-X interrupt must have been disabled and freed. Otherwise, calls to `ddi_intr_remove_handler()` will fail with `DDI_FAILURE`.

See the EXAMPLES section for code that illustrates the use of the `ddi_intr_dup_handler()` function.

Return Values

The `ddi_intr_dup_handler()` function returns:

`DDI_SUCCESS`	On success. Note that the interface should be verified to ensure that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.
`DDI_EINVAL`	On encountering invalid input parameters. `DDI_EINVAL` is also returned if a dup is attempted from a dup-ed interrupt or if the hardware device is found not to support MSI-X interrupts.
`DDI_FAILURE`	On any implementation specific failure.

Examples

EXAMPLE 1 Using the `ddi_intr_dup_handler()` function

```
int
add_msix_interrupts(intr_state_t *state)
{
    int x, y;

    /*
     * For this example, assume the device supports multiple
     * interrupt vectors, but only request to be allocated
     * 1 MSI-X to use and then dup the rest.
     */
    if (ddi_intr_get_nintrs(state->dip, DDI_INTR_TYPE_MSIX,
        &state->intr_count) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "Failed to retrieve the MSI-X interrupt count");
            return (DDI_FAILURE);
    }

    state->intr_size = state->intr_count * sizeof (ddi_intr_handle_t);
    state->intr_htable = kmem_zalloc(state->intr_size, KM_SLEEP);

    /* Allocate one MSI-X interrupt handle */
    if (ddi_intr_alloc(state->dip, state->intr_htable,
        DDI_INTR_TYPE_MSIX, state->inum, 1, &state->actual,
        DDI_INTR_ALLOC_STRICT) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "Failed to allocate MSI-X interrupt");
            kmem_free(state->intr_htable, state->intr_size);
            return (DDI_FAILURE);
    }

    /* Get the count of how many MSI-X interrupts we dup */
    state->dup_cnt = state->intr_count - state->actual;
```

EXAMPLE 1 Using the `ddi_intr_dup_handler()` function *(Continued)*

```
    if (ddi_intr_get_pri(state->intr_htable[0],
        &state->intr_pri) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "Failed to get interrupt priority");
            goto error1;
    }

    /* Make sure the MSI-X priority is below 'high level' */
    if (state->intr_pri >= ddi_intr_get_hilevel_pri()) {
            cmn_err(CE_WARN, "Interrupt PRI is too high");
            goto error1;
    }

    /*
     * Add the handler for the interrupt
     */
    if (ddi_intr_add_handler(state->intr_htable[0],
        (ddi_intr_handler_t *)intr_isr, (caddr_t)state,
        NULL) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "Failed to add interrupt handler");
            goto error1;
    }

    /* Enable the main MSI-X handle first */
    if (ddi_intr_enable(state->intr_htable[0]) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "Failed to enable interrupt");
            goto error2;
    }

    /*
     * Create and enable dups of the original MSI-X handler, note
     * that the inum we are using starts at 0.
     */
    for (x = 1; x < state->dup_cnt; x++) {
        if (ddi_intr_dup_handler(state->intr_htable[0],
            state->inum + x, &state->intr_htable[x]) != DDI_SUCCESS) {
                for (y = x - 1; y > 0; y--) {
                    (void) ddi_intr_disable(state->intr_htable[y]);
                    (void) ddi_intr_free(state->intr_htable[y]);
                }

            goto error2;
        }
        if (ddi_intr_enable(state->intr_htable[x]) != DDI_SUCCESS) {
            for (y = x; y > 0; y--) {
```

EXAMPLE 1 Using the `ddi_intr_dup_handler()` function *(Continued)*

```
                (void) ddi_intr_disable(state->intr_htable[y]);
                (void) ddi_intr_free(state->intr_htable[y]);
            }

            goto error2;
        }
    }

    return (DDI_SUCCESS);

error2:
    (void) ddi_intr_remove_handler(state->intr_htable[0]);
error1:
    (void) ddi_intr_free(state->intr_htable[0]);

    kmem_free(state->intr_htable, state->intr_size);
    return (DDI_FAILURE);
}

void
remove_msix_interrupts(intr_state_t *state)
{
    int x;

    /*
     * Disable all the handles and free the dup-ed handles
     * before we can remove the main MSI-X interrupt handle.
     */
    for (x = 1; x < state->dup_cnt; x++) {
        (void) ddi_intr_disable(state->intr_htable[x]);
        (void) ddi_intr_free(state->intr_htable[x]);
    }

    /*
     * We can remove and free the main MSI-X handler now
     * that all the dups have been freed.
     */
    (void) ddi_intr_disable(state->intr_htable[0]);
    (void) ddi_intr_remove_handler(state->intr_htable[0]);
    (void) ddi_intr_free(state->intr_htable[0]);

    kmem_free(state->intr_htable, state->intr_size);
}
```

Context The `ddi_intr_dup_handler()` function can be called from kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`, `ddi_intr_add_handler(9F)`, `ddi_intr_alloc(9F)`, `ddi_intr_clr_mask(9F)`, `ddi_intr_disable(9F)`, `ddi_intr_enable(9F)`, `ddi_intr_free(9F)`, `ddi_intr_get_pending(9F)`, `ddi_intr_get_supported_types(9F)`, `ddi_intr_set_mask(9F)`

Writing Device Drivers

Name ddi_intr_enable, ddi_intr_block_enable, ddi_intr_disable, ddi_intr_block_disable – enable or disable a given interrupt or range of interrupts

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>



int ddi_intr_enable(ddi_intr_handle_t h);

int ddi_intr_block_enable(ddi_intr_handle_t *h_array, int count);

int ddi_intr_disable(ddi_intr_handle_t h);

int ddi_intr_block_disable(ddi_intr_handle_t *h_array, int count);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_enable()`

h	DDI interrupt handle

`ddi_intr_block_enable()`

h_array	Pointer to an array of DDI interrupt handles
count	Number of interrupts

`ddi_intr_disable()`

h	DDI interrupt handle

`ddi_intr_block_disable()`

h_array	Pointer to an array of DDI interrupt handles
count	Number of interrupts

Description The `ddi_intr_enable()` function enables the interrupt given by the interrupt handle *h*.

The `ddi_intr_block_enable()` function enables a range of interrupts given by the *count* and *h_array* arguments, where *count* must be at least `1` and *h_array* is pointer to a count-sized array of interrupt handles.

The `ddi_intr_block_enable()` function can be used only if the device or host bridge supports the block enable/disable feature. The `ddi_intr_get_cap()` function returns the `RO` flag `DDI_INTR_FLAG_BLOCK` if the device or host bridge supports the interrupt block enable/disable feature for the given interrupt type. The `ddi_intr_block_enable()` function is useful for enabling MSI interrupts when the optional per-vector masking capability is not supported.

The `ddi_intr_enable()` or `ddi_intr_block_enable()` functions must be called after the required interrupt resources are allocated with `ddi_intr_alloc()`, the interrupt handlers are added through `ddi_intr_add_handler()`, and the required locks are initialized by `mutex`(9F) or `rwlock`(9F).

Once enabled by either of the enable calls, the interrupt can be taken and passed to the driver's interrupt service routine. Enabling an interrupt implies clearing any system or device mask bits associated with the interrupt.

The `ddi_intr_disable()` function disables the interrupt given by the interrupt handle *h*.

The `ddi_intr_block_disable()` function disables a range of interrupts given by the *count* and *h_array* arguments, where *count* must be at least `1` and *h_array* is pointer to a count-sized array of interrupt handles.

The `ddi_intr_block_disable()` function can be used only if the device or host bridge supports the block enable/disable feature. The `ddi_intr_get_cap()` function returns the `RO` flag `DDI_INTR_FLAG_BLOCK` if the device or host bridge supports the interrupt block enable/disable feature for the given interrupt type. The `ddi_intr_block_disable()` function is useful for disabling MSI interrupts when the optional per-vector masking capability is not supported.

The `ddi_intr_disable()` or `ddi_intr_block_disable()` functions must be called before removing the interrupt handler and freeing the corresponding interrupt with `ddi_intr_remove_handler()` and `ddi_intr_free()`, respectively. The `ddi_intr_block_disable()` function should be called if the `ddi_intr_block_enable()` function was used to enable the interrupts.

Return Values

The `ddi_intr_enable()`, `ddi_intr_block_enable()`, `ddi_intr_disable()`, and `ddi_intr_block_disable()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.

Context

The `ddi_intr_enable()`, `ddi_intr_block_enable()`, `ddi_intr_disable()`, and `ddi_intr_block_disable()` functions can be called from kernel non-interrupt context.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes`(5), `ddi_intr_add_handler`(9F), `ddi_intr_alloc`(9F), `ddi_intr_dup_handler`(9F), `ddi_intr_free`(9F), `ddi_intr_get_cap`(9F), `ddi_intr_remove_handler`(9F), `mutex`(9F), `rwlock`(9F)

Writing Device Drivers

Notes Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses `MSI` and `MSI-X` interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any `MSI` and `MSI-X` interrupt usage settings that have been applied.

Name ddi_intr_get_cap, ddi_intr_set_cap – get or set interrupt capabilities for a given interrupt type

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_cap(ddi_intr_handle_t h, int *flagsp);

int ddi_intr_set_cap(ddi_intr_handle_t h, int flags);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_get_cap()`

h	DDI interrupt handle
flagsp	Pointer to the capability flags returned for this handle

`ddi_intr_set_cap()`

h	DDI interrupt handle
flags	Contains the capability flag to be set

Description The `ddi_intr_get_cap()` function returns the interrupt capability flags for the interrupt handle *h*. Upon a successful return, the flags are returned in the integer pointed to by the *flagsp* argument.

These flags are typically combinations of the following:

`DDI_INTR_FLAG_EDGE`	For discrete interrupts, the host supports `edge` type of trigger. This flag is not returned for `DDI_INTR_TYPE_MSI` or `DDI_INTR_TYPE_MSIX` interrupt types. This is a read-write (RW) flag.
`DDI_INTR_FLAG_LEVEL`	For discrete interrupts the host supports `level`, `edge`, or both types of triggers. This flag is not returned for `DDI_INTR_TYPE_MSI` or `DDI_INTR_TYPE_MSIX` interrupt types.
`DDI_INTR_FLAG_MASKABLE`	The interrupt can be masked either by the device or by the host bridge, or optionally by the host. This is a read-only (`RO`) flag.
`DDI_INTR_FLAG_PENDING`	The interrupt supports an interrupt pending bit. This is a read-only (`RO`) flag.
`DDI_INTR_FLAG_BLOCK`	All interrupts of the given type must be block-enabled and are not individually maskable. This is a read-only (`RO`) flag.

The `ddi_intr_set_cap()` function allows a driver to specify the capability flags for the interrupt handle *h*. Only `DDI_INTR_FLAG_LEVEL` and `DDI_INTR_FLAG_EDGE` flags can be set.

Some devices can support both level and edge capability and either can be set by using the `ddi_intr_set_cap()` function. Setting the capability flags is device and platform dependent.

The `ddi_intr_set_cap()` function can be called after interrupts are allocated and prior to adding the interrupt handler. For all other times it returns failure.

Return Values

The `ddi_intr_get_cap()` and `ddi_intr_set_cap()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.
`DDI_ENOTSUP`	On device not supporting operation.

Context

The `ddi_intr_get_cap()` and `ddi_intr_set_cap()` functions can be called from either user or kernel non-interrupt context.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

`attributes(5)`, `ddi_intr_alloc(9F)`, `ddi_intr_block_enable(9F)`, `ddi_intr_get_nintrs(9F)`, `ddi_intr_get_pending(9F)`, `ddi_intr_get_supported_types(9F)`, `ddi_intr_set_mask(9F)`

Writing Device Drivers

Notes

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

Name ddi_intr_get_hilevel_pri – get minimum priority level for a high-level interrupt

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_hilevel_pri(void);
```

Interface Level Solaris DDI specific (Solaris DDI).

Description Upon a successful return, the `ddi_intr_get_hilevel_pri()` function returns the minimum priority level for a high-level interrupt. The return priority value can be used to compare to other priority values, such as those returned from `ddi_intr_get_pri`(9F), to determine if a given interrupt priority is a high-level interrupt.

High-level interrupts must be handled without using system services that manipulate thread or process states, because such interrupts are not blocked by the scheduler.

In addition, high-level interrupt handlers must take care to do a minimum of work because they cannot be preempted.

A typical high-level interrupt handler puts data into a circular buffer and schedule a soft interrupt by calling `ddi_intr_trigger_softint()`. The circular buffer can be protected by using a mutex that is properly initialized for the interrupt handler.

The `ddi_intr_get_hilevel_pri()` function can be used before calling `ddi_intr_add_handler()` to help determine which type of interrupt handler can be used. Most device drivers are designed with the knowledge that supported devices always generate low level interrupts. On some machines, however, interrupts are high-level above the scheduler level and on other machines they are not. Devices such as those those using SBus interrupts or VME bus level 6 or 7 interrupts must use the `ddi_intr_get_hilevel_pri()` function to test the type of interrupt handler that can be used.

Return Values The `ddi_intr_get_hilevel_pri()` function returns the priority value for a high-level interrupt.

Context The `ddi_intr_get_hilevel_pri()` function can be called from either user or kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also attributes(5), ddi_intr_add_handler(9F), ddi_intr_alloc(9F), ddi_intr_enable(9F), ddi_intr_get_pri(9F), ddi_intr_trigger_softint(9F), mutex(9F)

Writing Device Drivers

Name

ddi_intr_get_nintrs, ddi_intr_get_navail – return number of interrupts supported or available for a given interrupt type

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_nintrs(dev_info_t *dip, int type, int *nintrsp);

int ddi_intr_get_navail(dev_info_t *dip, int type, int *navailp);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

`ddi_intr_get_nintrs()`

dip	Pointer to `dev_info` structure
type	Interrupt type
nintrsp	Pointer to number of interrupts of the given type that are supported by the system

`ddi_intr_get_navail()`

dip	Pointer to `dev_info` structure
type	Interrupt type
navailp	Pointer to number of interrupts of the given type that are currently available from the system

Description

The `ddi_intr_get_nintrs()` function returns the number of interrupts of the given *type* supported by a particular hardware device. On a successful return, the number of supported interrupts is returned as an integer pointed to by the *nintrsp* argument.

If the hardware device is not found to support any interrupts of the given *type*, the `DDI_INTR_NOTFOUND` failure is returned rather than a zero in *nintrsp*.

The `ddi_intr_get_navail()` function returns the number of interrupts of a given *type* that is available to a particular hardware device. On a successful return, the number of available interrupts is returned as an integer pointed to by *navailp*.

The hardware device may support more than one interrupt and can request that all interrupts be allocated. The host software can then use policy-based decisions to determine how many interrupts are made available to the device. Based on the determination, a value is returned that should be used to allocate interrupts with the `ddi_int_alloc()` function.

If the device participates in resource management, a call to `ddi_intr_get_navail()` tells the device driver the number of interrupts of the given type that should be used. The host software can then use a policy-based decision to determine the number of interrupts to be allowed to the device. If the number is more than the number of interrupts currently being used, the device driver can ask for more resources. If the number is less than the number of interrupts currently being used, the device driver should prepare to disable and free the extra interrupts. The number of interrupts currently available is always a snapshot in time and can change if the interface is called again.

See `ddi_intr_get_supported_types(9F)` for a list of valid supported types for a given hardware device. The `ddi_intr_get_supported_types()` function must be called prior to calling either `ddi_intr_get_nintrs()` or `ddi_intr_get_navail()`.

Return Values The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_INTR_NOTFOUND`	On not finding any interrupts for the given interrupt type.
`DDI_FAILURE`	On any implementation specific failure.

Context The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions can be called from either user or kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`, `ddi_intr_alloc(9F)`, `ddi_intr_enable(9F)`, `ddi_intr_get_supported_types(9F)`

Writing Device Drivers

Notes The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions can be called at any time, even if the driver has added an interrupt handler for a given interrupt specification.

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

Name ddi_intr_get_pending – get pending bit for a given interrupt

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_pending(ddi_intr_handle_t h, int *pendingp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

h	DDI interrupt handle
pendingp	Pointer to the pending capability returned for this handle

Description The `ddi_intr_get_pending()` function returns non-zero as the integer pointed to by the *pendingp* argument if a corresponding interrupt is pending. The corresponding interrupt handle *h* must already be allocated. The call succeeds if the device or host bridge supports the ability to read the interrupt pending bits of its interrupts. The driver should use `ddi_intr_get_cap()` function to see if the `DDI_INTR_FLAG_PENDING` flag is returned to indicate that interrupts support interrupt pending bits.

If the `DDI_INTR_FLAG_PENDING` capability is not supported, `ddi_intr_get_pending()` returns `DDI_ENOTSUP` and zero in *pendingp*.

Return Values The `ddi_intr_get_pending()` function returns:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.
`DDI_ENOTSUP`	On device not supporting operation.

Context The `ddi_intr_get_pending()` function can be called from either user or kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`, `ddi_intr_block_enable(9F)`, `ddi_intr_block_disable(9F)`, `ddi_intr_clr_mask(9F)`, `ddi_intr_disable(9F)`, `ddi_intr_enable(9F)`, `ddi_intr_set_mask(9F)`

Writing Device Drivers

Notes Any consumer of this interface should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

Name ddi_intr_get_pri, ddi_intr_set_pri – get or set priority of a given interrupt

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_pri(ddi_intr_handle_t h, uint_t *prip);

int ddi_intr_set_pri(ddi_intr_handle_t h, uint_t pri);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters `ddi_intr_get_pri()`

h	DDI interrupt handle
prip	Pointer to the priority returned for this handle

`ddi_intr_set_pri()`

h	DDI interrupt handle
pri	Contains the priority to be set

Description The `ddi_intr_get_pri()` function returns the current priority of the interrupt handle *h* of a given device. Upon a successful return, *prip* points to a small integer value, typically in the `DDI_INTR_PRI_MIN`...`DDI_INTR_PRI_MAX` range, that represents the current software priority setting for the interrupt. See `<sys/ddi_intr.h>` for values of `DDI_INTR_PRI_MIN` or `DDI_INTR_PRI_MAX`.

The `ddi_intr_get_pri()` function can be called any time, even if the driver adds an interrupt handler for the interrupt specification.

The software priority returned from `ddi_intr_get_pri()` can be used in calls to `mutex_init()` and `rw_init()`.

The `ddi_intr_set_pri()` function sets the priority *pri* of the interrupt handle *h* of a given device. The function validates that the argument is within the supported range.

The `ddi_intr_set_pri()` function can only be called prior to adding the interrupt handler or when an interrupt handler is unassigned. `DDI_FAILURE` is returned in all other cases.

Return Values The `ddi_intr_get_pri()` and `ddi_intr_set_pri()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_FAILURE`	On any implementation specific failure.
`DDI_ENOTSUP`	On device not supporting operation.

Context The `ddi_intr_get_pri()` and `ddi_intr_set_pri()` functions can be called from kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes(5)`, `ddi_intr_alloc(9F)`, `ddi_intr_enable(9F)`, `mutex_init(9F)`, `rw_init(9F)`

Writing Device Drivers

Notes The priority returned from `ddi_intr_get_pri()` should be typecast by calling the `DDI_INTR_PRI` macro before passing it onto `mutex_init(9F)`.

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

Name ddi_intr_get_supported_types – return information on supported hardware interrupt types

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_get_supported_types(dev_info_t *dip, int *typesp);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to `dev_info` structure
typesp	Pointer to supported interrupt types

Description The `ddi_intr_get_supported_types()` function retrieves the interrupt types supported by a particular hardware device and by the system software. Upon successful return, the supported types are returned as a bit mask in the integer pointed to by the *typesp* argument. See `<sys/ddi_intr.h>` for a list of interrupts that can be returned by a hardware device.

For PCI devices that support MSI and/or MSI-X based hardware, this interface returns only the interrupt types that are supported by all the hardware in the path to the hardware device.

An interrupt type is usable by the hardware device if it is returned by the `ddi_intr_get_supported_types()` function. The device driver can be programmed to use one of the returned interrupt types to receive hardware interrupts.

Return Values The `ddi_intr_get_supported_types()` function returns:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters.
`DDI_INTR_NOTFOUND`	Returned when the hardware device is found not to support any hardware interrupts.

Context The `ddi_intr_get_supported_types()` function can be called from user or kernel non-interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `pci(4)`, `attributes(5)`, `pcmcia(7D)`, `sysbus(4)`, `ddi_intr_add_handler(9F)`, `ddi_intr_alloc(9F)`, `ddi_intr_enable(9F)`

Writing Device Drivers

Notes The `ddi_intr_get_supported_types()` function can be called by the device driver even at any time if the driver has added an interrupt handler for a given interrupt type.

Soft interrupts are always usable and are not returned by this interface.

Any consumer of this interface should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

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). This interface is obsolete. Use the new interrupt interfaces referenced in `Intro(9F)`. Refer to *Writing Device Drivers* for more information.

Parameters

dip	Pointer to `dev_info` structure.
inumber	Interrupt number.

Description The `ddi_intr_hilevel()` function 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.

The `ddi_intr_hilevel()` function 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 useruser, interrupt, or kernel context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `ddi_add_intr`(9F), `Intro`(9F), `mutex`(9F)

Writing Device Drivers

Name ddi_intr_set_mask, ddi_intr_clr_mask – set or clear mask for a given interrupt

Synopsis

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_set_mask(ddi_intr_handle_t h);

int ddi_intr_clr_mask(ddi_intr_handle_t h);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *h* DDI interrupt handle

Description The `ddi_intr_set_mask()` function masks the given interrupt pointed to by the device's interrupt handle *h* if the device or host bridge supports the masking operation. The `ddi_intr_get_cap()` function returns the `RO` flag `DDI_INTR_FLAG_MASKABLE` if the device or host bridge supports interrupt mask bits for the given interrupt type. In flight interrupts can still be taken and delivered to the driver.

The `ddi_intr_clr_mask()` function unmasks the given interrupt pointed by the device's interrupt handle *h* if the device or host bridge supports the masking operation.

The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions should be called only if an interrupt is enabled. Otherwise the framework will return `DDI_EINVAL` to such calls.

The mask cannot be cleared directly if the OS implementation has also temporarily masked the interrupt. A call to `ddi_intr_clr_mask()` must be preceded by a call to `ddi_intr_set_mask()`. It is not necessary to call `ddi_intr_clr_mask()` when adding and enabling the interrupt.

Return Values The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions return:

`DDI_SUCCESS`	On success.
`DDI_EINVAL`	On encountering invalid input parameters or when an interrupt is not enabled.
`DDI_FAILURE`	On any implementation specific failure.
`DDI_ENOTSUP`	On device not supporting operation.

Context The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions can be called from any context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes`(5), `ddi_intr_block_disable`(9F), `ddi_intr_block_enable`(9F), `ddi_intr_disable`(9F), `ddi_intr_enable`(9F), `ddi_intr_get_pending`(9F)

Writing Device Drivers

Notes Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

Name ddi_intr_set_nreq – set the number of interrupts requested for a device driver instance

Synopsis

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



int ddi_intr_set_nreq(dev_info_t *dip, int nreq);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

dip	Pointer to the `dev_info` structure.
nreq	Number of interrupts requested.

Description The `ddi_intr_set_nreq()` function changes the number of interrupts requested by a device driver instance.

The *nreq* parameter is the total number of interrupt resources that this instance of the device driver would like to have available. The *nreq* parameter includes any interrupt resources already allocated by the driver. For example, if the driver instance already has two MSI-X vectors and it wants two more, it should call this function with an *nreq* parameter set to four.

The *nreq* parameter can be any value between one and the maximum number of interrupts supported by the device hardware, as reported by a call to the `ddi_intr_get_nintrs`(9F) function. The driver receives a callback notifying it in cases when it must release any previously allocated interrupts, or when it is allowed to allocate more interrupts as a result of its new *nreq* parameter.

The `ddi_intr_set_nreq()`function is not supported unless a driver is already consuming interrupts, and if it has a registered callback handler that can process actions related to changes in interrupt availability. See `ddi_cb_register`(9F) for an explanation on how to enable this functionality.

Return Values The `ddi_intr_set_nreq()` function returns:

`DDI_SUCCESS`	on success
`DDI_EINVAL`	The operation is invalid because the *nreq* parameter is not a legal value
`DDI_ENOTSUP`	The operation is not supported. The driver must have a registered callback, and the system must have interrupt pools implemented.
`DDI_FAILURE`	Implementation specific failure

Context These functions can be called from kernel, non-interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Private
MT-Level	MT-Safe

See Also `attributes`(5), `attach`(9E), `ddi_cb_register`(9F), `ddi_intr_alloc`(9F), `ddi_intr_get_nintrs`(9F)

Notes The Interrupt Resource Management feature is limited to device driver instances that are using MSI-X interrupts (interrupt type `DDI_INTR_TYPE_MSIX`). Attempts to use this function for any other type of interrupts fails with `DDI_ENOTSUP`.

The total number of interrupts requested by the driver is initially defined by the *count* parameter provided by the driver's first call to the `ddi_intr_alloc`(9F) function, specifically during the driver instance's `attach`(9E) routine. The `ddi_intr_set_nreq()` function is only used if the driver instance experiences changes in its I/O load. In response to increased I/O load, the driver may want to request additional interrupt resources. In response to diminished I/O load. the driver may volunteer to return extra interrupt resources back to the system.

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 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 x86 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:

Previous Name	New Name
ddi_io_getb	ddi_io_get8
ddi_io_getw	ddi_io_get16

Previous Name	New Name
`ddi_io_getl`	`ddi_io_get32`

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 Level Solaris DDI specific (Solaris DDI). This interface is obsolete.

Parameters

dip	A pointer to the device's `dev_info` structure.
initial	The initial minimum DMAtransfer 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_x86`(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 The `ddi_iomin()` function, finds out the minimum DMAtransfer 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 The `ddi_iomin()` function returns the minimum DMAtransfer 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, interrupt, or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `ddi_dma_devalign`(9F), `ddi_dma_setup`(9F), `ddi_dma_sync`(9F), `ddi_dma_lim_sparc`(9S), `ddi_dma_lim_x86`(9S), `ddi_dma_req`(9S)

Writing Device Drivers

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()`.

Parameters

`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_x86(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 The `ddi_iopb_alloc()` function 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.

The `ddi_iopb_free()` function frees up memory allocated by `ddi_iopb_alloc()`.

Return Values The `ddi_iopb_alloc()` function returns:

`DDI_SUCCESS`	Memory successfully allocated.
`DDI_FAILURE`	Allocation failed.

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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

```
#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 x86 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:

Previous Name	New Name
ddi_io_putb	ddi_io_put8

Previous Name	New Name
`ddi_io_putw`	`ddi_io_put16`
`ddi_io_putl`	`ddi_io_put32`

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 x86 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:

Previous Name	New Name
`ddi_io_rep_getb`	`ddi_io_rep_get8`
`ddi_io_rep_getw`	`ddi_io_rep_get16`
`ddi_io_rep_getl`	`ddi_io_rep_get32`

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, 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 x86 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:

Previous Name	New Name
`ddi_io_rep_putb`	`ddi_io_rep_put8`
`ddi_io_rep_putw`	`ddi_io_rep_put16`
`ddi_io_rep_putl`	`ddi_io_rep_put32`

Name

ddi_log_sysevent – log system event for drivers

Synopsis

```
#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

Solaris DDI specific (Solaris DDI).

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

The `ddi_log_sysevent()` function 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:

```
uint64_t   eid_seq;        /* sysevent sequence number */
hrtime_t   eid_ts;         /* sysevent timestamp */
```

Return Values The `ddi_log_sysevent()` function returns:

`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` supports the following data types:

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

Context The `ddi_log_sysevent()` function can be called from user, interrupt, or kernel context, except when *sleep_flag* is `DDI_SLEEP`, in which case it cannot be called from interrupt context.

Examples **EXAMPLE 1** Logging System Event with No Attributes

```
    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

```
nvlist_t    *attr_list;
sysevent_id_t   eid;

if (nvlist_alloc(&attr_list, NV_UNIQUE_NAME_TYPE, KM_SLEEP) == 0)
{
    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.

```
static int
    xx_se_timeout_handler(xx_state_t *xx)
    {
        xx->xx_timeoutid = (xx_generate_event(xx) ?
            timeout(xx_se_timeout_handler, xx, 4) : 0);
    }

    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

Name

ddi_map_regs, ddi_unmap_regs – map or unmap registers

Synopsis

```
#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()`.

Parameters

`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

The `ddi_map_regs()` function 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.

The `ddi_unmap_regs()` function 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.

Return Values The `ddi_map_regs()` function returns:

`DDI_SUCCESS`	on success.

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

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

See Also `attributes(5)`, `sbus(4)`, `ddi_regs_map_free(9F)`, `ddi_regs_map_setup(9F)`

Writing Device Drivers

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.

Parameters

`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_x86`(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

The `ddi_mem_alloc()` function 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.

The `ddi_mem_free()` function frees up memory allocated by `ddi_mem_alloc()`.

Return Values

The `ddi_mem_alloc()` function returns:

`DDI_SUCCESS`	Memory successfully allocated.
`DDI_FAILURE`	Allocation failed.

Context

The `ddi_mem_alloc()` function can be called from user, interrupt, or kernel context, except when *flags* is set to `1`, in which case it cannot be called from interrupt context.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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 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 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:

Previous Name	New Name
ddi_mem_getb	ddi_mem_get8
ddi_mem_getw	ddi_mem_get16

Previous Name	New Name
`ddi_mem_getl`	`ddi_mem_get32`
`ddi_mem_getll`	`ddi_mem_get64`

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:

Previous Name	New Name
`ddi_mem_putb`	`ddi_mem_put8`

Previous Name	New Name
`ddi_mem_putw`	`ddi_mem_put16`
`ddi_mem_putl`	`ddi_mem_put32`
`ddi_mem_putll`	`ddi_mem_put64`

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.

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:

Previous Name	New Name
`ddi_mem_rep_getb`	`ddi_mem_rep_get8`
`ddi_mem_rep_getw`	`ddi_mem_rep_get16`
`ddi_mem_rep_getl`	`ddi_mem_rep_get32`
`ddi_mem_rep_getll`	`ddi_mem_rep_get64`

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);
```

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 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:

Previous Name	New Name
ddi_mem_rep_putb	ddi_mem_rep_put8
ddi_mem_rep_putw	ddi_mem_rep_put16
ddi_mem_rep_putl	ddi_mem_rep_put32
ddi_mem_rep_putll	ddi_mem_rep_put64

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;
```

EXAMPLE 1 : Using `ddi_mmap_get_model()` *(Continued)*

```
                da32p = (struct data32 *)shared_area;
                dp = &dtc;
                dp->len = da32p->len;
                dp->address = da32->address;
                break;
        }
        case DDI_MODEL_NONE:
                break;
        }
        /* continues along using dp */
       ...
}
```

See Also `mmap`(9E), `ddi_model_convert_from`(9F)

Writing Device Drivers

Name ddi_model_convert_from – determine data model type mismatch

Synopsis

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

 uint_tddi_model_convert_from(uint_t model);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

model	The data model type of the current thread.

Description `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 an 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.

Return Values

`DDI_MODEL_ILP32`	A conversion to/from `ILP32` is necessary.
`DDI_MODEL_NONE`	No conversion is necessary. Current thread and driver use the same data model.

Context `ddi_model_convert_from()` can be called from any context.

Examples **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.

```
struct passargs32 {
        int len;
        caddr32_t addr;
};

struct passargs {
        int len;
        caddr_t addr;
};
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp) {
        struct passargs pa;
```

EXAMPLE 1 : Using `ddi_model_convert_from()` in the `ioctl()` entry point to support both 32-bit and 64-bit applications. *(Continued)*

```
        switch (ddi_model_convert_from(mode & FMODELS)) {
            case DDI_MODEL_ILP32:
            {
                struct passargs32 pa32;

                ddi_copyin(arg, &pa32, sizeof (struct passargs32), mode);
                pa.len = pa32.len;
                pa.address = pa32.address;
                break;
            }
            case DDI_MODEL_NONE:
                ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
                break;
        }

        do_ioctl(&pa);
        . . . .
}
```

See Also

`ioctl`(9E), `mmap`(9E), `ddi_mmap_get_model`(9F)

Writing Device Drivers

Name ddi_node_name – return the devinfo node name

Synopsis

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



char *ddi_node_name(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

dip	A pointer the device's `dev_info` structure.

Description The `ddi_node_name()` function returns the device node name contained in the `dev_info` node pointed to by *dip*.

Return Values The `ddi_node_name()` function returns the device node name contained in the `dev_info` structure.

Context The `ddi_node_name()` function can be called from user, interrupt, or kernel context.

See Also `ddi_binding_name`(9F)

Writing Device Drivers

Name

ddi_no_info – stub for getinfo(9E)

Synopsis

```
#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

Solaris DDI specific (Solaris DDI)

Parameters

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)

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, interrupt, or kernel 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 2 Reading and logging the device type of a particular device:

```
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:

Previous Name	New Name
ddi_peekc	ddi_peek8
ddi_peeks	ddi_peek16
ddi_peekl	ddi_peek32
ddi_peekd	ddi_peek64

Name

ddi_periodic_add – issue nanosecond periodic timeout requests

Synopsis

```
#include <sys/dditypes.h>
#include <sys/sunddi.h>

ddi_periodic_t ddi_periodic_add(void (*func)(void *), void arg,
     hrtime_t interval, int level);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

func	The callback function is invoked periodically in the specified interval. If the argument level is zero, the function is invoked in kernel context. Otherwise, it's invoked in interrupt context at the specified level.
arg	The argument passed to the callback function.
interval	Interval time in nanoseconds.
level	Callback interrupt level. If the value is zero, the callback function is invoked in kernel context. If the value is more than zero, but less than or equal to ten, the callback function is invoked in interrupt context at the specified interrupt level, which may be used for real time applications. This value must be in range of 0-10, which can be either a numeric number, a pre-defined macro (`DDI_IPL_0`, ... , `DDI_IPL_10`), or the `DDI_INTR_PRI` macro with the interrupt priority.

Description

The `ddi_periodic_add()` function schedules the specified function to be periodically invoked in the nanosecond interval time.

As with `timeout`(9F), the exact time interval over which the function takes effect cannot be guaranteed, but the value given is a close approximation.

Return Values

`ddi_periodic_add()` returns the non-zero opaque value (`ddi_periodic_t`), which might be used for `ddi_periodic_delete`(9F) to specify the request.

Context

The `ddi_periodic_add()` function may be called from user or kernel context.

Examples

EXAMPLE 1 Using `ddi_periodic_add()` for a periodic callback function

In the following example, the device driver registers a periodic callback function invoked in kernel context.

```
static void
my_periodic_func(void *arg)
{
         /*
          * This handler is invoked periodically.
          */
         struct my_state *statep = (struct my_state *)arg;
```

EXAMPLE 1 Using `ddi_periodic_add()` for a periodic callback function *(Continued)*

```
         mutex_enter(&statep->lock);
         if (load_unbalanced(statep)) {
             balance_tasks(statep);
         }
         mutex_exit(&statep->lock);
}

static void
start_periodic_timer(struct my_state *statep)
{
         hrtime_t interval = CHECK_INTERVAL;

         mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
             (void *)DDI_IPL_0);

         /*
          * Register my_callback which is invoked periodically
          * in CHECK_INTERVAL in kernel context.
          */
          statep->periodic_id = ddi_periodic_add(my_periodic_func,
              statep, interval, DDI_IPL_0);
```

In the following example, the device driver registers a callback function invoked in interrupt context at level 7.

```
/*
 * This handler is invoked periodically in interrupt context.
 */
 static void
 my_periodic_int7_func(void *arg)
 {
          struct my_state *statep = (struct my_state *)arg;
          mutex_enter(&statep->lock);
          monitor_device(statep);
          mutex_exit(&statep->lock);
  }

  static void
  start_monitor_device(struct my_state *statep)
  {
          hrtime_t interval = MONITOR_INTERVAL;

          mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
              (void *)DDI_IPL_7);
```

```
            /*
             * Register the callback function invoked periodically
             * at interrupt level 7.
             */
            statep->periodic_id = ddi_periodic_add(my_periodic_int7_func,
                statep, interval, DDI_IPL_7);
      }
```

See Also `cv_timedwait`(9F), `ddi_intr_get_pri`(9F), `ddi_periodic_delete`(9F), `ddi_intr_get_softint_pri`(9F), `delay`(9F), `drv_usectohz`(9F), `qtimeout`(9F), `quntimeout`(9F), `timeout`(9F), `untimeout`(9F)

Notes A caller can only specify an interval in an integral multiple of 10ms. No other values are supported at this time. The interval specified is a lower bound on the interval on which the callback occurs.

Name ddi_periodic_delete – cancel nanosecond periodic timeout requests

Synopsis

```
#include <sys/dditypes.h>
#include <sys/sunddi.h>

void ddi_periodic_delete(ddi_periodic_t req);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

req	`ddi_periodic_t` opaque value returned by `ddi_periodic_add(9F)`

Description The `ddi_periodic_delete()` function cancels the `ddi_periodic_add(9F)` request that was previously issued.

As with `untimeout(9F)`, calling `ddi_periodic_delete()` against a periodic *timeout* request which is either running on another CPU, or has already been canceled causes no problems. Unlike `untimeout(9F)`, there are no restrictions on the lock which might be held across the call to `ddi_periodic_delete()`.

Context The `ddi_periodic_delete()` function may be called from user or kernel context.

Examples **EXAMPLE 1** Cancelling a timeout request

In the following example, the device driver cancels the *timeout* request by calling `ddi_periodic_delete()` against the request that was previously issued.

```
/*
* Stop the periodic timer
*/
static void
stop_periodic_timer(struct my_state *statep)
{
         ddi_periodic_delete(statep->periodic_id);
         delay(1); /* wait for one tick */
         mutex_destory(&statep->lock);
}

static void
start_periodic_timer(struct my_state *statep)
{
         hrtime_t interval = CHECK_INTERVAL;

         mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
             (void *)DDI_IPL_0);

         /*
          * Register my_callback which is invoked periodically
          * in CHECK_INTERVAL in kernel context.
          */
```

EXAMPLE 1 Cancelling a timeout request *(Continued)*

```
        statep->periodic_id = ddi_periodic_add(my_periodic_func,
            statep, interval, DDI_IPL_0);
}

static void
my_periodic_func(void *arg)
{
           /*
            * This handler is invoked periodically.
            */
           struct my_state *statep = (struct my_state *)arg;

           mutex_enter(&statep->lock);
           if (load_unbalanced(statep)) {
               balance_tasks(statep);
           }
           mutex_exit(&statep->lock);
}
```

See Also

`cv_timedwait`(9F), `ddi_intr_get_pri`(9F), `ddi_periodic_add`(9F), `delay`(9F), `drv_usectohz`(9F), `qtimeout`(9F), `quntimeout`(9F), `timeout`(9F), `untimeout`(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, interrupt, or kernel 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:

Previous Name	New Name
`ddi_pokec`	`ddi_poke8`
`ddi_pokes`	`ddi_poke16`
`ddi_pokel`	`ddi_poke32`
`ddi_poked`	`ddi_poke64`

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.

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()`	`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_int`(9F) or `ddi_prop_lookup`(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.

Return Values

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

`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.

The `ddi_prop_ undefine()` function returns the following values:

`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.

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

`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_NOT_FOUND`	On property search failure.

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

`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 cannot be called from interrupt context. Otherwise, they can be called from user, interrupt, or kernel context.

Examples

EXAMPLE 1 Creating a Property

The following example creates a property called *nblocks* for each partition on a disk.

```
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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	`ddi_prop_create()` and `ddi_prop_modify()` are Obsolete

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_exists – check for the existence of a property

Synopsis

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



int ddi_prop_exists(dev_t match_dev, dev_info_t *dip, uint_t flags,
     char *name);
```

Interface Level Solaris DDI specific (Solaris DDI).

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.

Description `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.

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()`.

```
/*
    * Enable "whizzy" mode if the "whizzy-mode" property exists
    */
    if (ddi_prop_exists(xx_dev, xx_dip, DDI_PROP_NOTPROM,
            "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_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:
name	String containing the name of the property.
defvalue	An integer value that is returned if the property cannot be found.

Flag values for *flags*:

`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).

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()`.

```
/*
    * 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

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);

int ddi_prop_lookup_byte_array(dev_t match_dev, dev_info_t *dip, uint_t flags,
     char *name, uchar_t **datap, uint_t *nelementsp);

void ddi_prop_free(void *data);
```

Parameters

<table>
<tr><td><i>match_dev</i></td><td>Device number associated with property or <code>DDI_DEV_T_ANY.</code></td></tr>
<tr><td><i>dip</i></td><td>Pointer to the device info node of device whose property list should be searched.</td></tr>
<tr><td><i>flags</i></td><td>Possible flag values are some combination of:
<table>
<tr><td><code>DDI_PROP_DONTPASS</code></td><td>Do not pass request to parent device information node if the property is not found.</td></tr>
<tr><td><code>DDI_PROP_NOTPROM</code></td><td>Do not look at PROM properties (ignored on platforms that do not support PROM properties).</td></tr>
</table></td></tr>
<tr><td><i>name</i></td><td>String containing the name of the property.</td></tr>
<tr><td><i>nelementsp</i></td><td>The address of an unsigned integer which, upon successful return, will contain the number of elements accounted for in the memory pointed at by <i>datap</i>. The elements are either integers, strings or bytes depending on the interface used.</td></tr>
<tr><td><i>datap</i></td><td>
<table>
<tr><td><code>ddi_prop_lookup_int_array()</code></td><td>The address of a pointer to an array of integers which, upon successful</td></tr>
</table></td></tr>
</table>

	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.

Interface Level Solaris DDI specific (Solaris DDI).

Description 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()`.

EXAMPLE 1 Using `ddi_prop_lookup_int_array()` *(Continued)*

```
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

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.

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 The `ddi_prop_op()` function 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.

The `ddi_getprop()` function 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.

The `ddi_getlongprop()` function 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*.

The `ddi_getlongprop_buf()` function 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.

The `ddi_getproplen()` function 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

The `ddi_prop_op()`, `ddi_getlongprop()`, `ddi_getlongprop_buf()`, and `ddi_getproplen()` functions 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.

The `ddi_getprop()` function 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, interrupt, or kernel context, provided `DDI_PROP_CANSLEEP` is not set; if it is set, they cannot be called from interrupt context.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	`ddi_getlongprop()`, `ddi_getlongprop_buf()`, `ddi_getprop()`, and `ddi_getproplen()` functions are Obsolete

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_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()`

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.

Interface Level

Solaris DDI specific (Solaris DDI).

Description

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 `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

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 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 and SBus buses. See `sysbus`(4), `isa`(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:

Previous Name	New Name
`ddi_putb`	`ddi_put8`
`ddi_putw`	`ddi_put16`
`ddi_putl`	`ddi_put32`
`ddi_putll`	`ddi_put64`

Name ddi_regs_map_free – free a previously mapped register address space

Synopsis

```
#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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus, SBus, ISA

See Also `attributes`(5), `ddi_regs_map_setup`(9F)

Writing Device Drivers

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

dip	Pointer to the device's `dev_info` structure.
rnumber	Index number to the register address space set.
addrp	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.
offset	Offset into the register address space.
len	Length to be mapped.
accattrp	Pointer to a device access attribute structure of this mapping (see `ddi_device_acc_attr`(9S)).
handlep	Pointer to a data access handle.

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:

`DDI_SUCCESS`	Successfully set up the mapping for data access.
`DDI_FAILURE`	Invalid register number *rnumber*, offset *offset*, or length *len*.
`DDI_ME_RNUMBER_RANGE`	Invalid register number *rnumber* or unable to find *reg* property.

`DDI_REGS_ACC_CONFLICT`	Cannot enable the register mapping due to access conflicts with other enabled mappings.

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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus, SBus, ISA

See Also

`attributes(5)`, `ddi_regs_map_free(9F)`, `ddi_device_acc_attr(9S)`

Writing Device Drivers

Name ddi_remove_event_handler – remove an NDI event service callback handler

Synopsis

```
#include <sys/dditypes.h>
#include <sys/sunddi.h>

int ddi_remove_event_handler(ddi_registration_id_t id);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

`ddi_registration_id_t` *id*	Unique system wide registration ID return by `ddi_add_event_handler(9F)` upon successful registration.

Description The `ddi_remove_event_handler()` function removes the callback handler specified by the registration *id* (`ddi_registration_id_t`). Upon successful removal, the callback handler is removed from the system and will not be invoked in the face of the event.

Return Values

`DDI_SUCCESS`	Callback handler removed successfully.
`DDI_FAILURE`	Failed to remove callback handler.

Context The `ddi_remove_event_handler()` function can be called from user and kernel contexts only.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Evolving

See Also `attributes(5)`, `ddi_add_event_handler(9F)`, `ddi_get_eventcookie(9F)`

Writing Device Drivers

Notes Device drivers must remove all registered callback handlers before `detach(9E)` processing for that device instance is complete.

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`:

```
ddi_remove_minor_node(dip, "dev1");
```

See Also `attach`(9E), `detach`(9E), `ddi_create_minor_node`(9F)

Writing Device Drivers

Name ddi_removing_power – check whether `DDI_SUSPEND` might result in power being removed from a device

Synopsis

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



int ddi_removing_power(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `ddi_removing_power()` function indicates whether a currently pending call into a driver's `detach(9E)` 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(9E)` code:

```
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 See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`, `cpr(7)`, `attach(9E)`, `detach(9E)`

Writing Device Drivers

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

```
#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);
```

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 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:

Previous Name	New Name
`ddi_rep_getb`	`ddi_rep_get8`
`ddi_rep_getw`	`ddi_rep_get16`
`ddi_rep_getl`	`ddi_rep_get32`
`ddi_rep_getll`	`ddi_rep_get64`

Name 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

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 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.

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:

Previous Name	New Name
`ddi_rep_putb`	`ddi_rep_put8`
`ddi_rep_putw`	`ddi_rep_put16`
`ddi_rep_putl`	`ddi_rep_put32`
`ddi_rep_putll`	`ddi_rep_put64`

Name ddi_root_node – get the root of the dev_info tree

Synopsis

```
#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 The `ddi_root_node()` function returns a pointer to the root node of the device information tree.

Return Values The `ddi_root_node()` function returns a pointer to a device information structure.

Context The `ddi_root_node()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

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 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):

`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.

credp	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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

See Also `mmap`(2), `attributes`(5), `devmap`(9E), `mmap`(9E), `segmap`(9E), `devmap_setup`(9F), `cb_ops`(9S), `ddi_device_acc_attr`(9S)

Writing Device Drivers

Name ddi_slaveonly – tell if a device is installed in a slave access only location

Synopsis

```
#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 The `ddi_slaveonly()` function 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 The `ddi_slaveonly()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

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 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`(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

The `ddi_soft_state_init()` and `ddi_soft_state_alloc()` functions can be called from user or kernel 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.

Examples

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)
```

EXAMPLE 1 Creating and Removing Data Structures *(Continued)*

```
                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.

```
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.

Name

ddi_strtol – String conversion routines

Synopsis

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

int ddi_strtol(const char *str, char **endptr, int base,
     long *result);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

str	Pointer to a character string to be converted.
endptr	Post-conversion final string of unrecognized characters.
base	Radix used for conversion.
result	Pointer to variable which contains the converted value.

Description

The `ddi_strtol()` function converts the initial portion of the string pointed to by *str* to a type `long` int representation and stores the converted value in result.

The function first decomposes the input string into three parts:

1. An initial (possibly empty) sequence of white-space characters (' ', '\t', '\n', '\r', '\f')
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The `ddi_strtol()` function then attempts to convert the subject sequence to an integer and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a plus ("+") or minus ("-") sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35 and only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign, if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed and the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Return Values Upon successful completion, `ddi_strtol()` returns `0` and stores the converted value in *result*. If no conversion is performed due to invalid *base*, `ddi_strtol()` returns `EINVAL` and the variable pointed by *result* is not changed.

If the correct value is outside the range of representable values, `ddi_strtol()` returns `ERANGE` and the value pointed to by *result* is not changed.

Context The `ddi_strtol()` function may be called from user, kernel or interrupt context.

See Also *Writing Device Drivers*

Name

ddi_strtoul – String conversion functions

Synopsis

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

int ddi_strtoul(const char *str, char **endptr, int base,
     unsigned long *result);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

str	Pointer to a character string to be converted.
endptr	Post-conversion final string of unrecognized characters.
base	Radix used for conversion.
result	Pointer to variable which contains the converted value.

Description

The `ddi_strtoul()` function converts the initial portion of the string pointed to by *str* to a type `unsigned long` int representation and stores the converted value in *result*.

The function first decomposes the input string into three parts:

1. An initial (possibly empty) sequence of white-space characters (' ', '\t', '\n', '\r', '\f')
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The `ddi_strtoul()` function then attempts to convert the subject sequence to an `unsigned` integer and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a plus (“+”) or minus (“-”) sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35 and only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed and the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Return Values Upon successful completion, `ddi_strtoul()` returns `0` and stores the converted value in *result*. If no conversion is performed due to invalid *base*, `ddi_strtoul()` returns `EINVAL` and the variable pointed by *result* is not changed.

If the correct value is outside the range of representable values, `ddi_strtoul()` returns `ERANGE` and the value pointed to by *result* is not changed.

Context The `ddi_strtoul()` function may be called from user, kernel or interrupt context.

See Also *Writing Device Drivers*

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 Level Solaris DDI specific (Solaris DDI).

Parameters

ddi_umem_alloc()	*size*	Number of bytes to allocate.
	flag	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.

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.

Name ddi_umem_iosetup – Setup I/O requests to application memory

Synopsis

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



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);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

cookie	The kernel memory cookie allocated by `ddi_umem_lock`(9F).
off	Offset from the start of the cookie.
len	Length of the I/O request in bytes.
direction	Must be set to `B_READ` for reads from the device or `B_WRITE` for writes to the device.
dev	Device number
blkno	Block number on device.
iodone	Specific `biodone`(9F) routine.
sleepflag	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.

Description The `ddi_umem_iosetup`(9F) 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 *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 `biodone`(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), `freerbuf`(9F), `physio`(9F), `buf`(9S)

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)

Parameters

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.

ddi_umem_unlock

cookie Kernel memory cookie allocated by `ddi_umem_lock()`.

Description The `ddi_umem_lock()` 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()`.

The process is not allowed to `exec(2)` or `fork(2)` while its physical pages are locked down by the device driver.

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, or locking *len* memory would exceed a limit or resource control on locked 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_unlock()` function consumes physical memory. The driver is responsible for a speedy unlock to free up the resources.

The `ddi_umem_unlock()` function can defer unlocking of the pages to a later time depending on the implementation.

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_usectohz`(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 */
          /* on system console */
14          cmn_err(CE_WARN, "^\007",
15                            (getminor(dev) & 0xf));
```

EXAMPLE 1 `delay()` Example *(Continued)*

```
            /* wait one minute and try again */
            delay(60 * drv_usectohz(1000000));
        }
```

See Also `biodone`(9F), `biowait`(9F), `cv_timedwait`(9F), `ddi_in_panic`(9F), `drv_hztousec`(9F), `drv_usectohz`(9F), `drv_usecwait`(9F), `timeout`(9F), `untimeout`(9F)

Writing Device Drivers

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 CTXMGT_SIZE    0x100000
#define NORMAL_SIZE    0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
```

EXAMPLE 1 Using devmap_default_access in devmap_access. *(Continued)*

```
 */
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_CTXMG) >= 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.
     */
    } 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
```

EXAMPLE 1 Using devmap_default_access in devmap_access. *(Continued)*

```
        return (-1);

    return (err);
}
```

See Also

`devmap_access`(9E), `devmap_do_ctxmgt`(9F), `devmap_callback_ctl`(9S)

Writing Device Drivers

Name

devmap_devmem_setup, devmap_umem_setup – set driver memory mapping parameters

Synopsis

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



int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip, struct devmap_callback_ctl *callb
     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 *callbac
     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

Solaris DDI specific (Solaris DDI).

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:

dhp	An opaque data structure 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.
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 *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).

See `devmap_map(9E)`, `devmap_access(9E)`, `devmap_dup(9E)`, and `devmap_unmap(9E)` for details on these entry points.

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 – perform device context switching on a mapping

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.

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.

```
. . .
#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_CTXMG) >= 0 && diff < CTXMGT_SIZE) {
        /*
         * calculates the length for context switching
         */
```

EXAMPLE 1 Using devmap_do_ctxmgt in the devmap_access entry point. *(Continued)*

```
        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.
     */
    } 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

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

The `devmap_set_ctx_timeout()` function 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

The `devmap_set_ctx_timeout()` function can be called from user, interrupt, or kernel context.

See Also

`devmap_contextmgt`(9E), `devmap_map`(9E), `timeout`(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_tlen, 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_tlen, uint_t prot, uint_t maxprot,
     uint_t flags, cred_t *cred);
```

Interface Level

Solaris DDI specific (Solaris DDI).

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.

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

`0`	Successful completion.
`Non-zero`	An error occurred. The return value of `devmap_setup()` and `ddi_devmap_segmap()` should be used directly in the `segmap`(9E) entry point.

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

Name devmap_unload, devmap_load – control validation of memory address translations

Synopsis

```
#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);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

	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.
devmap_load() only	*type*	Type of access operation.
	rw	Direction of access.

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

`0`	Successful completion.
`Non-zero`	An error occurred.

Context These routines can be called from user or kernel context only.

Examples **EXAMPLE 1** Managing a One-Page Device Context

The following shows an example of managing a device context that is one page in length.

```
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

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 Level Solaris DDI specific (Solaris DDI).

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, interrupt, or kernel context.

See Also `strategy`(9E), `buf`(9S)

Writing Device Drivers

Warnings The `disksort()` function does no locking. Therefore, any locking is completely the responsibility of the caller.

Name dlbindack, dlphysaddrack, dlokack, dlerrorack, dluderrorind – DLPI device driver helper functions

Synopsis

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



void dlokack(queue_t *wq, mblk_t *mp, t_uscalar_t correct_primitive);

void dlerrorack(queue_t *wq, mblk_t *mp, t_uscalar_t error_primitive,
     t_uscalar_t error, t_uscalar_t unix_errno);

void dlbindack(queue_t *wq, mblk_t *mp, t_scalar_t sap, const void *addrp,
     t_uscalar_t addrlen, t_uscalar_t maxconind,  t_uscalar_t xidtest);

void dlphysaddrack(queue_t *wq, mblk_t *mp, const void *addrp,
     t_uscalar_t addrlen);

void dluderrorind(queue_t *wq, mblk_t *mp, const void *addrp,
     t_uscalar_t addrlen, t_uscalar_t error, t_uscalar_t unix_errno);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

wq	Streams write queue.
mp	Pointer to the bind request message.
sap	Service access point being requested.
addrp	Pointer to the dlpi layer source address.
addrlen	Size of the `dlpi` layer address pointed to by *addr*.
maxconind	Maximum number of `DL_CONNECT_IND` messages allowed to be outstanding per stream.
xidtest	The `XID` and `TEST` responses supported.
correct_primitive	Identifies the `DL` primitive completing successfully.
error_primitive	Identifies the `DL` primitive in error.
error	`DLPI` error associated with the failure in the `DLPI` request.
unix_errno	Corresponding `UNIX` system error that can be associated with the failure in the `DLPI` request.

Description All functions described in this manpage take a pointer to the message passed to the `DLPI` provider (`mblk_t`) and attempt to reuse it in formulating the `M_PROTO` reply. If the message block is too small to be reused, it is freed and a new one is allocated.

All functions reply upstream using `qreply(9F)`. The write-side queue pointer must be provided.

The `dlokack()` function provides the successfull acknowledgement `DL_OK_ACK` message reply to the `DLPI` provider and is used to complete many of the `DLPI` requests in the `DLPI` consumer.

The `dlerrorack()` function provides the unsuccessfull acknowledgement `DL_ERROR_ACK` message reply to the `DLPI` provider and is used for error completions were required for `DLPI` requests in the `DLPI` consumer.

The `dlbindack()` function provides the `DL_BIND_ACK` message reply to the `DLPI` provider and is used to complete the `DL_BIND_REQ` processing in the `DLPI` consumer.

The `dlphysaddrack()` function provides the `DL_PHYS_ADDR_ACK` message reply used to complete the `DL_PHYS_ADDR_ACK` processing.

The `dluderrorind()` function provides the `DL_UDERROR_IND` message reply used to complete an unsuccessful `DL_UNITDATA_REQ`.

Return Values

None.

Notes

These functions are not required if you are are writing a `DLPI` device driver using `gld`(7D).

Context

All `DLPI` helper functions can be called from user, interrupt, or kernel context.

See Also

`gld`(7D), `dlpi`(7P), `qreply`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name drv_getparm – retrieve kernel state information

Synopsis

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



int drv_getparm(unsigned int parm, void *value_p);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

parm	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.
value_p	A pointer to the data space in which the value of the parameter is to be copied.

Description Since the release of the Solaris 2.6 operating environment, the `drv_getparm()` function has been replaced by `ddi_get_lbolt`(9F), `ddi_get_time`(9F), and `ddi_get_pid`(9F).

The `drv_getparm()` function verifies that *parm* corresponds to a kernel parameter that may be read. If the value of *parm* does not correspond to a parameter or corresponds to a parameter that may not be read, `-1` is returned. Otherwise, the value of the parameter is stored in the data space pointed to by *value_p*.

The `drv_getparm()` function does not explicitly check to see whether the device has the appropriate context when the function is called and the function does not check for correct alignment in the data space pointed to by *value_p*. It is the responsibility of the driver writer to use this function only when it is appropriate to do so and to correctly declare the data space needed by the driver.

Return Values The `drv_getparm()` function 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 The `drv_getparm()` function can be called from user context only when using `PPGRP`, `PPID`, `PSID`, `UCRED`, or `UPROCP`. It can be called from user, interrupt, or kernel 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

Name drv_hztousec – convert clock ticks to microseconds

Synopsis

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



clock_t drv_hztousec(clock_t hertz);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

hertz	The number of clock ticks to convert.

Description The `drv_hztousec()` function converts into microseconds the time expressed by *hertz*, which is in system clock ticks.

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. The `drv_hztousec()` function can be used by the driver to convert the reading from clock ticks to a known unit of time.

Return Values 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.

Context The `drv_hztousec()` function can be called from user, interrupt, or kernel context.

See Also `ddi_get_lbolt`(9F), `drv_usectohz`(9F), `drv_usecwait`(9F)

Writing Device Drivers

Name drv_priv – determine driver privilege

Synopsis

```
#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 The `drv_priv()` function 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 process that has the `{PRIV_SYS_DEVICES}` privilege asserted in its effective set. This function should be used only when file access modes, special minor device numbers, and the device policy (see `privileges`(5), `add_drv`(1M)) 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 The `drv_priv()` function can be called from user, interrupt, or kernel context.

See Also `add_drv`(1M), `update_drv`(1M), `privileges`(5)

Writing Device Drivers

Name drv_usectohz – convert microseconds to clock ticks

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 The `drv_usectohz()` function 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.

The `drv_usectohz()` function 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 The `drv_usectohz()` function can be called from user, interrupt, or kernel context.

See Also `delay`(9F), `drv_hztousec`(9F), `timeout`(9F)

Writing Device Drivers

Notes If the *microsecs* argument to `drv_usectohz()` is less than `drv_hztousec`(9F), `drv_usectohz()` returns one tick. This, coupled with multiplication, can result in significantly longer durations than expected. For example, on a machine where `hz` is 100, calling `drv_usectohz()` with a *microsecs* value less than 10000 returns a result equivalent to 10000 (1 tick). This type of mistake causes code such as“5000 * drv_usectohz(1000)” to compute a duration of 50 seconds instead of the intended 5 seconds.

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 Architecture independent level 1 (DDI/DKI).

Parameters

microsecs	The number of microseconds to busy-wait.

Description The `drv_usecwait()` function 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) can be invoked from user or kernel 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 The `drv_usecwait()` function can be called from user, interrupt, or kernel 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. The `drv_usecwait()` function does not mask out interrupts.

Name dupb – duplicate a message block descriptor

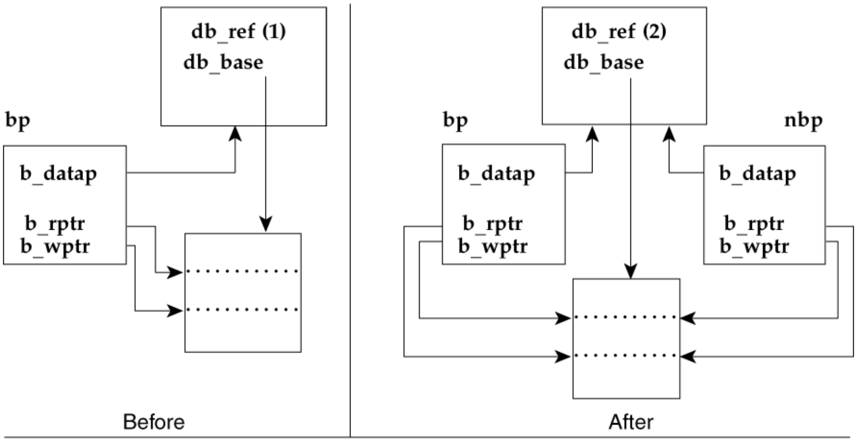
Synopsis

```
#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*.



nbp=dupb(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.

Examples

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).

```
  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
 9   while ((mp = getq(q)) != NULL) {
10        if (mp->b_datap->db_type >= QPCTL) {
11             putnext(q, mp);
12        } else if (mp->b_datap->db_type != M_DATA) {
13             if (canputnext(q))
14                  putnext(q, mp);
15             else {
16                  putbq(q, mp);
17                  return;
18             }
19        } else {  /* M_DATA */
20             if (canputnext(q)) {
```

EXAMPLE 1 Using `dupb()` *(Continued)*

```
                if ((bp = dupb(hdr)) == NULL)
                     bp = copyb(hdr);
                if (bp == NULL) {
                     size_t size = msgdsize(mp);
                     putbq(q, mp);
                     if (xx->xx_qbufcall_id) {
                          /* qbufcall pending */
                          return;
                     }
                     xx->xx_qbufcall_id = qbufcall(q, size,
                          BPRI_MED, xxxcallback, (intptr_t)q);
                     return;
                }
                linkb(bp, mp);
                putnext(q, bp);
           } else {
                putbq(q, mp);
                return;
           }
      }
 }
}
 void
 xxxcallback(q)
      queue_t *q;
 {
      struct xx *xx = (struct xx *)q->q_ptr;

      xx->xx_qbufcall_id = 0;
      qenable(q);
 }

 xxxclose(q, cflag, crp)
      queue_t *q;
      int  cflag;
      cred_t *crp;
 {
      struct xx *xx = (struct xx *)q->q_ptr;
      ...
      if (xx->xx_qbufcall_id)
           qunbufcall(q, xx->xx_qbufcall_id);
      ...
 }
```

See Also `srv`(9E), `copyb`(9F), `qbufcall`(9F), `datab`(9S), `msgb`(9S)

Writing Device Drivers STREAMS Programming Guide

Name dupmsg – duplicate a message

Synopsis

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



mblk_t *dupmsg(mblk_t *mp);
```

Interface Level 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

Name

enableok – reschedule a queue for service

Synopsis

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



void enableok(queue_t *q);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	A pointer to the queue to be rescheduled.

Description

The `enableok()` function enables queue *q* to be rescheduled for service. It reverses the effect of a previous call to `noenable`(9F) on *q* by turning off the `QNOENB` flag in the queue.

Context

The `enableok()` function can be called from user, interrupt, or kernel context.

Examples

EXAMPLE 1 Using `emableok()`

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.

```
void
qrestart(rdwr_q)
        register queue_t *rdwr_q;
{
   enableok(rdwr_q);
   /* re-enable a queue that has been disabled */
   (void) qenable(rdwr_q);
}
```

See Also

`noenable`(9F), `qenable`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name esballoc, desballoc – allocate a message block using a caller-supplied buffer

Synopsis

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



mblk_t *esballoc(uchar_t *base, size_t size, uint_t pri,
     frtn_t *fr_rtnp);

mblk_t *desballoc(uchar_t *base, size_t size, uint_t pri,
     frtn_t *fr_rtnp);
```

Interface Level `esballoc()`: Architecture independent level 1 (DDI/DKI)

`desballoc()`: Solaris DDI specific (Solaris DDI)

Parameters

base	Address of caller-supplied data buffer.
size	Number of bytes in data buffer.
pri	Priority of the request (no longer used).
fr_rtnp	Free routine data structure.

Description The `esballoc()` and `desballoc()` functions operate identically to `allocb`(9F), except that the data buffer to associate with the message is specified by the caller. The allocated message will have both the `b_wptr` and `b_rptr` set to the supplied data buffer starting at *base*. Only the buffer itself can be specified by the caller. The message block and data block header are allocated as if by `allocb`(9F).

When `freeb`(9F) is called to free the message, the driver's message-freeing routine, referenced through the `free_rtn`(9S) structure, is called with appropriate arguments to free the data buffer.

The `free_rtn`(9S) structure includes the following members:

```
void (*free_func)();     /* caller’s freeing routine */
caddr_t free_arg;        /* argument to free_func() */
```

Instead of requiring a specific number of arguments, the `free_arg` field is defined of type `caddr_t`. This way, the driver can pass a pointer to a structure if more than one argument is needed.

If `esballoc()` was used, then `free_func` will be called asynchronously at some point after the message is no longer referenced. If `desballoc()` was used, then `free_func` will be called synchronously by the thread releasing the final reference. See `freeb`(9F).

The `free_func` routine must not sleep, and must not access any dynamically allocated data structures that could be freed before or during its execution. In addition, because messages allocated with `desballoc()` are freed in the context of the caller, `free_func` must not call another module's `put` procedure, or attempt to acquire a private module lock which might be held by another thread across a call to a STREAMS utility routine that could free a message block. Finally, `free_func` routines specified using `desballoc` may run in interrupt context and thus must only use synchronization primitives that include an interrupt priority returned from `ddi_intr_get_pri`(9F) or `ddi_intr_get_softint_pri`(9F). If any of these restrictions are not followed, the possibility of lock recursion or deadlock exists.

Return Values On success, a pointer to the newly allocated message block is returned. On failure, `NULL` is returned.

Context The `esballoc()` and `desballoc()` functions can be called from user, interrupt, or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `allocb`(9F), `ddi_intr_get_pri`(9F), `ddi_intr_get_softint_pri`(9F), `freeb`(9F), `datab`(9S), `free_rtn`(9S)

Writing Device Drivers

STREAMS Programming Guide

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 The `esbbcall()` function, 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. The *func* argument can be a routine that calls `esballoc(9F)` or it may be another kernel function.

Return Values On success, a `bufcall` IDis 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 The `esbbcall()` function can be called from user, interrupt, or kernel context.

See Also `allocb(9F)`, `bufcall(9F)`, `esballoc(9F)`, `timeout(9F)`, `datab(9S)`, `unbufcall(9F)`

Writing Device Drivers STREAMS Programming Guide

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 Architecture independent level 1 (DDI/DKI).

Parameters

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 The `flushband()` function 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 The `flushband()` function can be called from user, interrupt, or kernel 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 Level

Architecture independent level 1 (DDI/DKI).

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

The `flushq()` function 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

The `flushq()` function can be called from user, interrupt, or kernel 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 `FLUSHR` 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 & FLUSHR)
11             flushq(RD(q), FLUSHALL);
12        if (*mp->b_rptr & FLUSHW)
13             flushq(q, FLUSHALL);
14        putnext(q, mp);
15        break;
          . . .
```

EXAMPLE 1 Using `flushq()` *(Continued)*

```
 }
}
```

See Also

`flushband`(9F), `freemsg`(9F), `putq`(9F), `qreply`(9F)

Writing Device Drivers STREAMS Programming Guide

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 Architecture independent level 1 (DDI/DKI).

Description The `freeb()` function 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 The `freeb()` function can be called from user, interrupt, or kernel 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

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 The `freemsg()` function calls `freeb`(9F) to free all message and data blocks associated with the message pointed to by *mp*.

Context The `freemsg()` function can be called from user, interrupt, or kernel 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.

Name freerbuf – free a raw buffer header

Synopsis

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



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

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

bp	Pointer to a previously allocated buffer header structure.

Description The `freerbuf()` function 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 The `freerbuf()` function can be called from user, interrupt, or kernel context.

See Also `getrbuf`(9F), `kmem_alloc`(9F), `kmem_free`(9F), `kmem_zalloc`(9F)

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.

Name geterror – return `I/O` error

Synopsis

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



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

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

bp	Pointer to a `buf(9S)` structure.

Description The `geterror()` function returns the error number from the error field of the buffer header structure.

Return Values An error number indicating the error condition of the `I/O` request is returned. If the `I/O` request completes successfully, `0` is returned.

Context The `geterror()` function can be called from user, interrupt, or kernel context.

See Also `buf(9S)`

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 `gethrtime(3C)`. Values returned by both of these functions can be interleaved for comparison purposes.

Name 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 The `getmajor()` function extracts the major number from a device number.

Return Values The major number.

Context The `getmajor()` function can be called from user, interrupt, or kernel 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;

#ifdef DEBUG
cmn_err(CE_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.

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 The `getminor()` function extracts the minor number from a device number.

Return Values The minor number.

Context The `getminor()` function can be called from user, interrupt, or kernel 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.

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

The `get_pktiopb()` function 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.

The `free_pktiopb()` function is used for freeing the packet and its associated resources.

Return Values The `get_pktiopb()` function returns a pointer to the newly allocated `scsi_pkt` or a `NULL` pointer.

Context If *callback* is `SLEEP_FUNC`, then this routine can be called only from user or kernel context. Otherwise, it can be called from user, interrupt, or kernel context. The *callback* function should not block or call routines that block.

The `free_pktiopb()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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)`.

The `get_pktiopb()` function uses scarce resources. For this reason and its obsolescence (see above), its use is discouraged.

Name getq – get the next message from a queue

Synopsis

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



mblk_t *getq(queue_t *q);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters *q* Pointer to the queue from which the message is to be retrieved.

Description The `getq()` function 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.

Context The `getq()` function can be called from user, interrupt, or kernel context.

Examples See `dupb(9F)`.

See Also `srv(9E)`, `bcanput(9F)`, `canput(9F)`, `dupb(9F)`, `putbq(9F)`, `putq(9F)`, `qenable(9F)`

Writing Device Drivers

STREAMS Programming Guide

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 Architecture independent level 1 (DDI/DKI).

Parameters

sleepflag	Indicates whether driver should sleep for free space.

Description The `getrbuf()` function 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.

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

Context The `getrbuf()` function can be called from user, interrupt, or kernel 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

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);

void gld_linkstate(gld_mac_info_t *macinfo, int32_t newstate);
```

Interface Level 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.
newstate	Media link state.

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`( ).

`gld_linkstate()` is called by the device driver to notify GLD of changes in the media link state. The newstate argument should be set to one of the following:

`GLD_LINKSTATE_DOWN`	The media link is unavailable.
`GLD_LINKSTATE_UP`	The media link is unavailable.
`GLD_LINKSTATE_UNKNOWN`	The status of the media link is unknown.

If a driver calls `gld_linkstate()`, it must also set the GLD_CAP_LINKSTATE bit in the gldm_capabilties field of the `gld_mac_info`(9S) structure.

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

Name hat_getkpfnum – get page frame number for kernel address

Synopsis

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

pfn_t hat_getkpfnum(caddr_t addr);
```

Interface Level This interface is obsolete. A driver `devmap(9E)` entry point should be provided instead.

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, device-mapped virtual address *addr*. Otherwise the return value is undefined.

Context `hat_getkpfnum()` can be called only from user or kernel context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Obsolete

See Also `attach(9E)`, `detach(9E)`, `devmap(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.

Name hook_alloc – allocate a hook_t data structure

Synopsis

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

hook_t *hook_alloc(const int version);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

version	must always be the symbol `HOOK_VERSION`.

Description The `hook_alloc()` function allocates a `hook_t` structure, returning a pointer for the caller to use.

Return Values Upon success, `hook_alloc()` returns a pointer to the allocated *hook_t* structure. On failure, `hook_alloc()` returns a `NULL` pointer.

Context The `hook_alloc()` function may be called from user or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `hook_free`(9F), `hook_t`(9S)

Name hook_free – free a hook_t data structure

Synopsis

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

void hook_free(hook_t * hook);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

hook	pointer returned by `hook_alloc`(9F).

Description The `hook_free()` function frees a `hook_t` structure that was originally allocated by `hook_alloc`(9F).

Context The `hook_free()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `hook_alloc`(9F), `hook_t`(9S)

Name id32_alloc, id32_free, id32_lookup – 32-bit driver ID management routines

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

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 Level

The functions described here are obsolete. For the `inb()`, `inw()`, and `inl()` functions, use, respectively, `ddi_get8`(9F), `ddi_get8`(9F), and `ddi_get8`(9F) instead. For `repinsb()`, `repinsw()`, and`repinsl()`, use, respectively, `ddi_rep_get8`(9F), `ddi_rep_get8`(9F), and `ddi_rep_get8`(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

The `inb()`, `inw()`, and `inl()` functions return the value that was read from the I/O port.

Context

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

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	x86
Stability Level	Obsolete

See Also `isa`(4), `attributes`(5), `ddi_get8`(9F), `ddi_get8`(9F), `ddi_get8`(9F), `ddi_rep_get8`(9F), `ddi_rep_get8`(9F), `ddi_rep_get8`(9F), `outb`(9F)

Writing Device Drivers

Name

insq – insert a message into a queue

Synopsis

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



int insq(queue_t *q, mblk_t *emp, mblk_t *nmp);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Pointer to the queue containing message *emp*.
emp	Enqueued message before which the new message is to be inserted. `mblk_t` is an instance of the `msgb`(9S) structure.
nmp	Message to be inserted.

Description

The `insq()` function 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

The `insq()` function returns `1` on success, and `0` on failure.

Context

The `insq()` function can be called from user, interrupt, or kernel 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.

```
 1  #include
 2  #include
 3
 4  static int
 5  xxxwput(queue_t *q, mblk_t *mp)
 6  {
 7   union T_primitives *tp;
 8   mblk_t *bp;
 9   union T_primitives *ntp;
10
11   switch (mp->b_datap->db_type) {
12   case M_PROTO:
13        tp = (union T_primitives *)mp->b_rptr;
14        switch (tp->type) {
15        case T_DATA_REQ:
16                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;
                  if (ntp->type != T_EXDATA_REQ)
                      break;
               }
             }
             (void)insq(q, bp, mp);
             /* End of region that must be protected */
             break;
     . . .
           }
 }
}
```

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.

Name	IOC_CONVERT_FROM – determine if there is a need to translate M_IOCTL contents.

Synopsis

```
#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 The `IOC_CONVERT_FROM()` function returns the following 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 The `IOC_CONVERT_FROM()` macro can be called from user, interrupt, or kernel 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

```
#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 The `kmem_alloc()` function 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.

The `kmem_zalloc()` function is like `kmem_alloc()` but returns zero-filled memory.

The `kmem_free()` function 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 The `kmem_alloc()` and `kmem_zalloc()` 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*. The `kmem_free()` function can be called from from user, interrupt, or kernel 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, intkmflag);

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

Solaris DDI specific (Solaris DDI)

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.
kmflag	Possible flags are:

`KM_SLEEP`	Allow sleeping (blocking) until memory is available.
`KM_NOSLEEP`	Return NULL immediately if memory is not available.
`KM_PUSHPAGE`	Allow the allocation to use reserved memory.

**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*	Propagated *kmflag* values.

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 `_fini`(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* has either `KM_SLEEP` or `KM_NOSLEEP` set, indicating whether it is acceptable to wait for memory if none is currently available.

A small pool of reserved memory is available to allow the system to progress toward the goal of freeing additional memory while in a low memory situation. The `KM_PUSHPAGE` flag enables use of this reserved memory pool on an allocation. This flag can be used by drivers that implement `strategy`(9E) on memory allocations associated with a single I/O operation. The driver guarantees that the I/O operation will complete (or timeout) and, on completion, that the memory will be returned. The `KM_PUSHPAGE` flag should be used only in `kmem_cache_alloc()` calls. All allocations from a given cache should be consistent in their use of the flag. A driver that adheres to these restrictions can guarantee progress in a low memory situation without resorting to complex private allocation and queuing schemes. If `KM_PUSHPAGE` is specified, `KM_SLEEP` can also be used without causing deadlock.

`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:

```
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 but 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 tags)
{
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
    return (0);
}

void
foo_destructor(void *buf, void *arg)
{
    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,
```

EXAMPLE 1 Object Caching *(Continued)*

```
        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);
```

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 `0`. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor must return `-1`.

`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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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 Summer 1994 Technical Conference (1994).

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 Level Solaris DDI specific (Solaris DDI)

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 `kstat`s, `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 no mounted partitions.) Note: Persistent `kstat`s 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

```
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

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

Name

kstat_named_init, kstat_named_setstr – initialize a named kstat

Synopsis

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

```
void kstat_named_init(kstat_named_t *knp, const 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`	Obsolete. Use `KSTAT_DATA_INT64`.
`KSTAT_DATA_ULONGLONG`	Obsolete. Use `KSTAT_DATA_UINT64`.
`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`. The string argument must remain valid even after the function that is calling `kstat_named_setstr()` is returned. 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_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`(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 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 ldi_add_event_handler – add NDI event service callback handler

Synopsis

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

int ldi_add_event_handler(ldi_handle_t lh, ddi_eventcookie_t ec,
     void (*handler)(ldi_handle_t, ddi_eventcookie_t,
     void *, void *) void *arg, ldi_callback_id_t *id);
```

Interface Level Solaris DDI Specific (Solaris DDI).

Parameters

ldi_handle_t lh
Layered handle representing event notification device.

ddi_eventcookie_t ec
Cookie returned from call to `ldi_get_eventcookie`(9F).

*void (*handler)(ldi_handle_t, ddi_eventcookie_t, void *, void *)*
Callback handler for NDI event service notification.

*void *arg*
Pointer to opaque data supplied by caller. Typically, this is a pointer to the layered driver's softstate structure.

*ldi_callback_id_t *id*
Pointer to registration id, where a unique registration id is returned. Registration id must be saved and used when calling `ldi_remove_event_handler`(9F) to unregister a callback handler.

Description The `ldi_add_event_handler()` function adds a callback handler to be invoked at the occurance of the event specified by the cookie. Adding a callback handler is also known as subscribing to an event. Upon successful subscription, the handler is invoked when the event occurs. You can unregister the handler by using `ldi_remove_event_handler`(9F).

An instance of a layered driver can register multiple handlers for an event or a single handler for multiple events. Callback order is not defined and should be assumed to be random.

The routine handler is invoked with the following arguments:

`ldi_handle_t` *lh*	Layered handle representing the device for which the event notification is requested.
`ddi_eventcookie_t` *ec*	Structure describing event that occurred.
`void` **arg*	Opaque data pointer provided by the driver during callback registration.
`void` **impl_data*	Pointer to event specific data defined by the framework that invokes the callback function.

Return Values

`DDI_SUCCESS`	Callback handler registered successfully.
`DDI_FAILURE`	Failed to register callback handler. Possible reasons include lack of resources or a bad cookie.

Context

The `ldi_add_event_handler()` function can be called from user and kernel contexts only.

See Also

`ldi_get_eventcookie`(9F), `ldi_remove_event_handler`(9F)

Writing Device Drivers

Notes

Layered drivers must remove all registered callback handlers for a device instance, represented by the layered handle, by calling `ldi_remove_event_handler`(9F) before the layered driver's `detach(9E)` routine completes.

Name ldi_aread, ldi_awrite – Issue an asynchronous read or write request to a device

Synopsis

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

int ldi_aread(ldi_handle_t lh,  struct aio_req *aio_reqp, cred_t *cr);

int  ldi_awrite(ldi_handle_t lh,  struct aio_req *aio_reqp, cred_t *cr);
```

Parameters

lh	Layered handle.
cr	Pointer to a credential structure.
aio_reqp	Pointer to the `aio_req(9S)` structure that describes where the data is to be stored or obtained from.

Description The `ldi_awrite()` function passes an asynchronous write request to a device entry point specified by the layered handle. This operation is supported for block and character devices.

The `ldi_aread()` function passes an asynchronous read request to a device entry point specified by the layered handle. This operation is supported for block and character devices.

Return Values The `ldi_awrite()` and `ldi_aread()` functions return `0` upon success. If a failure occurs before the request is passed on to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

Context These functions may be called from user context.

Name ldi_devmap – Issue a devmap request to a device

Synopsis

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

int ldi_devmap(ldi_handle_t lh, devmap_cookie_t dhp, offset_t off, size_t len,
     size_t *maplen, uint_t model);
```

Parameters

lh	Layered handle.
dhp	Opaque mapping handle used by the system to describe mapping.
off	User offset within the logical device memory at which mapping begins.
len	Mapping length (in bytes).
maplen	Pointer to length (in bytes) of validated mapping. (Less than or equal to *len*).
model	Data model type of current thread.

Description The `ldi_devmap()` function passes an devmap request to the device entry point for the device specified by the layered handle. This operation is supported for character devices.

Return Values The `ldi_devmap()` function returns `0` upon success. If a failure occurs before the request is passed to the device, possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

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

Name

ldi_dump – Issue a dump request to a device

Synopsis

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

int ldi_dump(ldi_handle_t lh, caddr_t addr, daddr_t blkno,  int nblk);
```

Parameters

lh	Layered handle.
addr	Area dump address.
blkno	Block offset to dump memory.
nblk	Number of blocks to dump.

Description

The `ldi_dump()` function passes a dump request to the device entry point specified by the layered handle. This operation is supported for block devices.

Return Values

The `ldi_dump()` function returns `0` upon success. If a failure occurs before the request is passed on to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

Context

These functions may be called from user or kernel context.

Name ldi_get_dev, ldi_get_otyp, ldi_get_devid, ldi_get_minor_name – Extract information from a layered handle

Synopsis

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

int ldi_get_dev(ldi_handle_t lh, dev_t *devp);

int ldi_get_otyp(ldi_handle_t lh, int *otyp);

int ldi_get_devid(ldi_handle_t lh, ddi_devid_t *devid);

int ldi_get_minor_name(ldi_handle_t lh, char **minor_name);
```

Parameters

lh	Layered handle
otyp	Indicates on which interface the driver was opened. Valid settings are: OTYP_BLK Open device block interface. OTYP_CHR Open device character interface.
devp	Pointer to a device number.
devid	Device ID.
minor_name	Minor device node name.

Description The `ldi_get_dev()` function retrieves the `dev_t` associated with a layered handle.

The `ldi_get_otyp()` retrieves the open flag that was used to open the device associated with the layered handle.

The `ldi_get_devid()` function retrieves a *devid* for the device associated with the layered handle. The caller should use `ddi_devid_free()` to free the devid when done with it.

The `ldi_get_minor_name()` function retrieves the name of the minor node opened for the device associated with the layered handle. `ldi_get_minor_name()` allocates a buffer containing the minor node name and returns it via the *minor_name* parameter. The caller should use `kmem_free()` to release the buffer when done with it.

Return Values The `ldi_get_dev()`, `ldi_get_otyp()`, `ldi_get_devid()`, and `ldi_get_devid()` functions return `0` upon success.

In case of an error, the following values may be returned:

EINVAL	Invalid input parameters.
ENOTSUP	The operation is not supported for this device.

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

Name	ldi_get_eventcookie – retrieve NDI event service cookie

Synopsis

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

int ldi_get_eventcookie(ldi_handle_t lh, char *name ddi_eventcookie_t *
ecp);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

ldi_handle_t lh	Layered handle.
*char *name*	null-terminated string containing the event name.
*ddi_eventcookie_t *ecp*	Pointer to the kernel event cookie.

Description The `ldi_get_eventcookie()` function queries the device tree for a cookie matching the given event name and returns a reference to that cookie. The search is performed by calling up the device tree hierarchy of the device represented by the layered driver handle until the request is satisfied by a bus nexus driver, or the top of the dev_info tree is reached.

The cookie returned by this function can be used to register a callback handler with `ldi_add_event_handler`(9F).

Return Values

`DDI_SUCCESS`	Cookie handle is returned.
`DDI_FAILURE`	Request was not serviceable by any nexus driver in the target device's ancestral device tree hierarchy.

Context This function may be called from user or kernel contexts.

See Also `ldi_add_event_handler`(9F), `ldi_remove_event_handler`(9F)

Writing Device Drivers

Name ldi_get_size – Retrieve device size

Synopsis

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

int ldi_get_size(ldi_handle_t lh, uint64_t *sizep);
```

Parameters

lh	Layered handle.
sizep	Pointer to the caller's unsigned 64-bit integer buffer.

Description The `ldi_get_size()` function uses the layered driver handle to calculate and return a device's size. The device size is returned within the caller supplied buffer (**sizep*). A valid layered driver handle must be obtained via the `ldi_open_by_name`(9F) interface prior to calling `ldi_get_size()`.

Return Values The `ldi_get_size()` function returns the following values:

DDI_SUCCESS	The device size has been returned within the caller supplied buffer.
DDI_FAILURE	The device size could not be found or determined.

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

See Also `ldi_open_by_name`(9F)

Writing Device Drivers

Name ldi_ident_from_dev, ldi_ident_from_stream, ldi_ident_from_dip, ldi_ident_release – ldi cookie management

Synopsis

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

int ldi_ident_from_dip(dev_info_t *dip, ldi_ident_t *lip);

int ldi_ident_from_dev(dev_t dev, ldi_ident_t  *lip);

void ldi_ident_from_stream(struct queue  *sq, ldi_ident_t  *lip);

void ldi_ident_release(ldi_ident_t li);
```

Parameters

li	ldi identifier.
lip	ldi identifier pointer.
dip	Pointer to device info node
dev	Device number
sq	Pointer to a stream queue

Description The `ldi_ident_from_dev()` function allocates and returns an ldi identifier that is associated with the device number specified by dev. The new ldi identifier is returned via the ldi identifier pointer parameter *lip*.

The `ldi_ident_from_dip()` function allocates and returns an ldi identifier that is associated with the device info node pointed to by dip. The new ldi identifier is returned via the ldi identifier pointer parameter *lip*.

The `ldi_ident_from_stream()` function allocates and returns an ldi identifier that is associated with the stream pointed to by *queue*. The new ldi identifier is returned via the ldi identifier pointer parameter *lip*.

The `ldi_ident_release()` function releases an identifier that was allocated via one of the `ldi_ident_from()`* functions.

Return Values The `ldi_ident_from_dev()`, `ldi_ident_from_dip()`, and `ldi_ident_from_stream()` functions return `0` upon success.

All of these functions return `EINVAL` for invalid input parameters.

Context These functions can be called from user or kernel context.

Name ldi_ioctl – send an ioctl to a device

Synopsis

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



int ldi_ioctl(ldi_handle_t lh, int cmd, intptr_t arg, int mode, cred_t  *cr,
     int *rvalp);
```

Parameters

lh	Layered handle.
cr	Pointer to a credential structure used to open a device.
rvalp	Caller return value. (May be set by driver and is valid only if the `ioctl()` succeeds).
cmd	Command argument. Interpreted by driver `ioctl()` as the operation to be performed.
arg	Driver parameter. Argument interpretation is driver dependent and usually depends on the command type.
mode	Bit field that contains: FKIOCTL — Inform the target device that the ioctl originated from within the kernel.

Description The `ldi_ioctl()` function passes an ioctl request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.

If *arg* is interpreted as a pointer (that is, as not an immediate value) and the data pointed to by *arg* is in the kernels address space, the `FKIOCTL` flag should be set. This indicates to the target driver that no data model conversion is necessary.

If the caller of `ldi_ioctl()` is not the originator of the ioctl data pointed to by *arg*, (for example, when passing on an ioctl request from a user process), the caller must pass on the mode parameter from the original ioctl. This is because the mode parameter contains the contains the `FMODELS` bits that enable the target driver to determine the data model of the process which originated the ioctl and perform any necessary conversions. See `ddi_model_convert_from`(9F) for more information.

Stream Ioctls For a general description of streams ioctls see `streamio`(7I). `ldi_ioctl()` supports a number of streams ioctls, using layered handles in the place of file descriptors. When issuing streams ioctls the `FKIOCTL` parameter should be specified. The possible return values for supported ioctl commands are also documented in `streamio`(7I).

The following streams ioctls are supported:

I_PLINK	Behaves as documented in streamio(7I). The layered handle *lh* should point to the streams multiplexer. The *arg* parameter should point to a layered handle for another streams driver.
I_UNPLINK	Behaves as documented in streamio(7I)). The layered handle *lh* should point to the streams multiplexer. The *arg* parameter is the multiplexor ID number returned by `I_PLINK` when the streams were linked.

Return Values The `ldi_ioctl()` function returns `0` upon success. If a failure occurs before the request is passed on to the device, possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

Context These functions can be called from user or kernel context.

See Also `streamio`(7I), `ddi_model_convert_from`(9F)

Name ldi_open_by_dev, ldi_open_by_name, ldi_open_by_devid, ldi_close – open and close devices

Synopsis

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

int ldi_open_by_dev(dev_t *devp, int otyp, int flag, cred_t  *cr, ldi_handle_t *lhp,
     ldi_ident_t li);

int ldi_open_by_name(char  *pathname, int flag, cred_t *cr, ldi_handle_t *lhp,
     ldi_ident_t li);

int ldi_open_by_devid(ddi_devid_t devid, char  *minor_name, int flag,
     cred_t *cr, ldi_handle_t *lhp, ldi_ident_t li);

int ldi_close(ldi_handle_t lh, int flag, cred_ t *cr);
```

Parameters

lh	Layered handle
lhp	Pointer to a layered handle that is returned upon a successful open.
li	LDI identifier.
cr	Pointer to the credential structure used to open a device.
devp	Pointer to a device number.
pathname	Pathname to a device.
devid	Device ID.
minor_name	Minor device node name.
otyp	Flag passed to the driver indicating which interface is open. Valid settings are: OTYP_BLK — Open the device block interface. OTYP_CHR — Open the device character interface. Only one OTYP flag can be specified. To open streams devices, specify `OTYP_CHR`.
flag	Bit field that instructs the driver on how to open the device. Valid settings are: FEXCL — Open the device with exclusive access; fail all other attempts to open the device. FNDELAY — Open the device and return immediately. Do not block the open even if something is wrong. FREAD — Open the device with read-only permission. (If ORed with `FWRITE`, allow both read and write access).

FWRITE	Open a device with write-only permission (if ORed with `FREAD`, then allow both read and write access).
FNOCTTY	Open the device. If the device is a tty, do not attempt to open it as a session-controlling tty.

Description The `ldi_open_by_dev()`, `ldi_open_by_name()` and `ldi_open_by_devid()` functions allow a caller to open a block, character, or streams device. Upon a successful open, a layered handle to the device is returned via the layered handle pointed to by *lhp*. The ldi identifier passed to these functions is previously allocated with `ldi_ident_from_stream`(9F), `ldi_ident_from_dev`(9F), and `ldi_ident_from_dip`(9F).

The `ldi_open_by_dev()` function opens a device specified by the dev_t pointed to by devp. Upon successful open, the caller should check the value of the dev_t to see if it has changed. (Cloning devices will change this value during opens.) When opening a streams device, otyp must be OTYP_CHR.

The `ldi_open_by_devid()` function opens a device by devid. The caller must specify the minor node name to open.

The `ldi_open_by_name()` function opens a device by pathname. Pathname is a null terminated string in the kernel address space. Pathname must be an absolute path, meaning that it must begin with '/'. The format of the pathname supplied to this function is either a `/devices` path or any other filesystem path to a device node. Opens utilizing `/devices` paths are supported before root is mounted. Opens utilizing other filesystem paths to device nodes are supported only if root is already mounted.

The `ldi_close()` function closes a layered handle that was obtained with either `ldi_open_by_dev()`, `ldi_open_by_name()`, or `ldi_open_by_devid()`. After `ldi_close()` returns the layered handle, the *lh* that was previously passed in is no longer valid.

Return Values The `ldi_close()` function returns `0` for success. `EINVAL` is returned for invalid input parameters. Otherwise, any other error number may be returned by the device.

The `ldi_open_by_dev()` and `ldi_open_by_devid()` functions return `0` upon success. If a failure occurs before the device is open, possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENODEV	Requested device does not exist.
ENXIO	Unsupported device operation or access mode.

The `ldi_open_by_name()` function returns `0` upon success. If a failure occurs before the device is open, possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENODEV	Requested device path does not exist.
EACCES	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *cr* are denied.
ENXIO	Unsupported device operation or access mode.

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

These functions should not be called from a device's attach, detach, or power entry point. This could result in a system crash or deadlock.

See Also `scsi_vhci`(7D), `ldi_ident_from_dev`(9F), `ldi_ident_from_dip`(9F), `ldi_ident_from_stream`(9F)

Notes Use only OTYP_CHR or OTYP_BLK options when you use the `ldi_open_by_dev()` and `ldi_open_by_devid()` functions to open a device. Other flags, including OTYP_LYR, have been deprecated and should not be used with these interfaces.

The caller should be aware of cases when multiple paths to a single device may exist. (This can occur for scsi disk devices if `scsi_vhci`(7D)) is disabled or a disk is connected to multiple controllers not supported by scsi_vhci(7D).

In these cases, `ldi_open_by_devid()` returns a device handle that corresponds to a particular path to a target device. This path may not be the same across multiple calls to `ldi_open_by_devid()`. Device handles associated with the same device but different access paths should have different filesystem device paths and dev_t values.

In the cases where multiple paths to a device exist and access to the device has not been virtualized via MPXIO (as with scsi disk devices not accessed via `scsi_vhci`(7D)), the LDI does not provide any path fail-over capabilities. If the caller wishes to do their own path management and failover they should open all available paths to a device via `ldi_open_by_name()`.

Name ldi_poll – Poll a device

Synopsis

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



int ldi_poll(ldi_handle_t lh, short events, int anyyet, short *reventsp,
     struct pollhead **phpp);
```

Parameters

lh	Layered handle.
events	Potential events. Valid events are:

POLLIN	Data other than high priority data may be read without blocking.
POLLOUT	Normal data may be written without blocking.
POLLPRI	High priority data may be received without blocking.
POLLHUP	Device hangup has occurred.
POLLERR	An error has occurred on the device.
POLLRDNORM	Normal data (priority band = 0) may be read without blocking.
POLLRDBAND	Data from a non-zero priority band may be read without blocking.
POLLWRNORM	Data other than high priority data may be read without blocking.
POLLWRBAND	Priority data (priority band > 0) may be written.

anyyet	A flag that is non-zero if any other file descriptors in the pollfd array have events pending. The `poll(2)` system call takes a pointer to an array of pollfd structures as one of its arguments. See `poll(2)` for more details.
reventsp	Pointer to a bitmask of the returned events satisfied.
phpp	Pointer to a pointer to a pollhead structure.

Description The `ldi_poll()` function passes a poll request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.

Return Values The `ldi_poll()` function returns `0` upon success. If a failure occurs before the request is passed on to the device, possible return values are:

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

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

Name ldi_prop_exists – Check for the existence of a property

Synopsis

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

int ldi_prop_exists(ldi_handle_t lh, uint_t flags, char *name);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

lh	Layered handle.
flags	Possible flag values are some combination of:

LDI_DEV_T_ANY	Match the lookup request independent of the actual dev_t value that was used when the property was created. The flag indicates any dev_t value (including DDI_DEV_T_NONE) associated with a possible property match satisfies the matching criteria.
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.

Description `ldi_prop_exists()` checks for the existence of a property associated with a device represented by the layered driver handle, regardless of the property value data type.

Properties are searched for based on the dip and dev_t values associated with the layered handle, and the property name. This association is handled by the layered driver infrastructure on behalf of the consumers of `ldi_prop_exists()`.

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 of the device represented by the layered handle.
6. Return `0` if not found and `1` if found.

Typically, the specific dev_t value associated with the device represented by the layered handle (ldi_handle_t) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the ldi property look up functions.

However, if the LDI_DEV_T_ANY flag is used, the ldi property lookup functions will match the request regardless of the dev_t value associated with the property at the time of its creation. If a property was created with a dev_t set to DDI_DEV_T_NONE, the only way to look up this property is with the LDI_DEV_T_ANY flag. PROM properties are always created with a dev_t set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.

Return Values `ldi_prop_exists()` returns `1` if the property exists and `0` otherwise.

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

Example `The following example demonstrates the use of ldi_prop_exists().`

```
       /* Determine the existence of the "interrupts" property */
         ldi_prop_exists(lh, LDI_DEV_T_ANY|DDI_PROP_NOTPROM, "interrupts");
```

See Also `ddi_prop_exists(9F)`

Writing Device Drivers

Name

ldi_prop_get_int, ldi_prop_get_int64 – Lookup integer property

Synopsis

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

int ldi_prop_get_int(ldi_handle_t lh, uint_t flags, char *name,
     int defvalue);

int64_t ldi_prop_get_int64(ldi_handle_t lh, uint_t flags, char *name,
     int64_t defvalue);
```

Parameters

lh	Layered handle.
flags	Possible flag values are some combination of:

LDI_DEV_T_ANY	Match the lookup request independent of the actual dev_t value that was used when the property was created. Indicates any dev_t value (including DDI_DEV_T_NONE) associated with a possible property match satisfies the matching criteria.
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 property name.
defvalue	Integer value that is returned if the property is not found.

Interface Level

Solaris DDI specific (Solaris DDI)

Description

The `ldi_prop_get_int()` and `ldi_prop_get_int64()` functions search for an integer property associated with a device represented by the layered driver handle. If the integer property is found, the functions return the property value.

Properties are searched for based on the dip and dev_t values associated with the layered handle, the property name, and 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 of the device represented by the layered handle.

6. Return defvalue.

Typically, the specific dev_t value associated with the device represented by the layered handle (ldi_handle_t) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the ldi property look up functions.

However, if the LDI_DEV_T_ANY flag is used, the ldi property lookup functions match the request regardless of the dev_t value associated with the property at the time of its creation. If a property was created with a dev_t set to DDI_DEV_T_NONE, the only way to look up this property is with the LDI_DEV_T_ANY flag. PROM properties are always created with a dev_t 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 property. If the property is not found, the argument defvalue is returned as the property value.

`ldi_prop_get_int64()` does not search the PROM for 64-bit property values.

Return Values `ldi_prop_get_int()` and `ldi_prop_get_int64()` return the property value. 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_t, DDI_PROP_NOT_FOUND is returned.

Context `ldi_prop_get_int()` and `ldi_prop_get_int64()` can be called from user or kernel context.

Examples Using `ldi_prop_get_int64()`.

`The following example demonstrates the use of ldi_prop_get_int64().`

```
/*
 * Get the value of the integer "timeout" property, using
 * our own default if no such property exists
 */

int64_t timeout, defval;

timeout = ldi_prop_get_int64(lh, LDI_DEV_T_ANY|DDI_PROP_DONTPASS,
    propname, defval);
```

See Also `ddi_prop_get_int(9F)`, `ddi_prop_get_int64(9F)`, `ldi_prop_exists(9F)`.

Writing Device Drivers

Name ldi_prop_lookup_int_array, ldi_prop_lookup_int64_array, ldi_prop_lookup_string_array, ldi_prop_lookup_string, ldi_prop_lookup_byte_array – Lookup property information

Synopsis

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

int ldi_prop_lookup_int_array(ldi_handle_t lh, uint_t  flags, char *name,
     int **datap, uint_t *nelementsp);

int ldi_prop_lookup_int64_array(ldi_handle_t lh, uint_t  flags, char *name,
     int64_t  **datap, uint_t *nelementsp);

int ldi_prop_lookup_string_array(ldi_handle_t lh, uint_t  flags, char *name,
     char ***datap, uint_t *nelementsp);

int ldi_prop_lookup_string(ldi_handle_t lh, uint_t  flags, char *name,
     char  **datap);

int ldi_prop_lookup_byte_array(ldi_handle_t lh, uint_t  flags, char *name,
     uchar_t **datap, uint_t *nelements);
```

Parameters

lh	Layered handle.
flags	Possible flag values are some combination of:

LDI_DEV_T_ANY	Match the lookup request independent of the actual `dev_t` value that was used when the property was created. The flag indicates any `dev_t` value (including DDI_DEV_T_NONE) associated with a possible property match will satisfy the matching criteria.
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 property name.
nelements	The address of an unsigned integer which, upon successful return, contains the number of elements accounted for in the memory pointed at by datap. Depending on the interface you use, the elements are either integers, strings or bytes.

datap

ldi_prop_lookup_int_array()	Pointer address to an array of integers which, upon successful return, point to memory containing the integer array property value.
ldi_prop_lookup_int64_array()	Pointer address to an array of 64-bit integers which, upon successful return, point to memory containing the integer array property value.

ldi_prop_lookup_string_array()	Pointer address to an array of strings which, upon successful return, point to memory containing the array of strings. The string array is formatted as an array of pointers to NULL terminated strings, much like the argv argument to `execve(2)`.
ldi_prop_lookup_string()	Pointer address to a string which, upon successful return, points to memory containing the NULL terminated string value of the property.
ldi_prop_lookup_byte_array()	Pointer address to an array of bytes which, upon successful return, point to memory containing the property byte array value.

Interface Level Solaris DDI specific (Solaris DDI).

Description The property look up functions search for and, if found, return the value of a given property. Properties are searched for based on the dip and dev_t values associated with the layered handle, the property name, and 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 of the device represented by the layered handle.
6. Return `DDI_PROP_NOT_FOUND`.

Typically, the specific dev_t value associated with the device represented by the layered handle (ldi_handle_t) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the ldi property look up functions.

However, if the LDI_DEV_T_ANY flag is used, the ldi property lookup functions match the request regardless of the dev_t value associated with the property at the time of its creation. If a property was created with a dev_t set to DDI_DEV_T_NONE, then the only way to look up this property is with the LDI_DEV_T_ANY flag. PROM properties are always created with a dev_t set to DDI_DEV_T_NONE.

name must always be set to the name of the property being looked up.

For the `ldi_prop_lookup_int_array()`, `ldi_prop_lookup_int64_array()`, `ldi_prop_lookup_string_array()`, `ldi_prop_lookup_string()`, and `ldi_prop_lookup_byte_array()` functions, datap is the address of a pointer which, upon successful return, points 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 functions below for details on the different return values. nelementsp is the address of an unsigned integer which, upon successful return, contains the number of integer, string or byte elements accounted for in the memory pointed at by *datap.

All of the property look up functions may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up function and is finished with that property, it must be freed by call `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if you call `ldi_prop_lookup_int_array()` with datap set to the address of a pointer to an integer, &my-int-ptr, the companion free call is ddi_prop_free(my-int-ptr).

Property look up functions are described below:

`ldi_prop_lookup_int_array()`	This function 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.
`ldi_prop_lookup_int64_array()`	This function searches for and returns an array of integer property values. An array of integers is defined to *nelementsp number of 8 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 This function does not search the PROM for 64-bit property values.
`ldi_prop_lookup_string_array()`	This function searches for and returns a property that is an array of strings. datap should be set to an 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)`.

`ldi_prop_lookup_string()`	This function searches for and returns a property that is a null-terminated string. datap should be set to the address of a pointer to a string which, upon successful return, points to memory containing the string value of the property.
`ldi_prop_lookup_byte_array()`	This function 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, points to memory containing the byte array value of the property.
`ddi_prop_free()`	Frees the resources associated with a property previously allocated using `ldi_prop_lookup_int_array()`, `ldi_prop_lookup_int64_array()`, `ldi_prop_lookup_string_array()`, `ldi_prop_lookup_string()`, and `ldi_prop_lookup_byte_array()`.

Return Values The functions `ldi_prop_lookup_int_array()`, `ldi_prop_lookup_int64_array()`, `ldi_prop_lookup_string_array()`, `ldi_prop_lookup_string()`, and `ldi_prop_lookup_byte_array()` return the following values:

DDI_PROP_SUCCESS	Property found and returned.
DDI_PROP_INVAL_ARG	If an attempt is made to look up a property with a NULL ldi handle, name is NULL or name is a the null string.
DDI_PROP_NOT_FOUND	Property not found.
DDI_PROP_UNDEFINED	Prop explicitly undefined (see `ddi_prop_undefine(9F)`).
DDI_PROP_CANNOT_DECODE	Property value cannot be decoded.

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

Example `Using ldi_prop_lookup_int64_array().`

```
       The following example demonstrates the use of
       ldi_prop_lookup_int64_array().


       int64_t *options;
       uint_t  noptions;

       /*
```

```
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */

if  (ldi_prop_lookup_int64_array(lh,
    LDI_DEV_T_ANY|DDI_PROP_NOTPROM, "options",
    &options, &noptions) == DDI_PROP_SUCCESS) {
       /*
          * Process the options data from the property
         * we just received. Let’s do "our thing" with data.
         */
        xx_process_options(options, noptions);

        /*
         * Free the memory allocated for the property data
         */
        ddi_prop_free(options);
}
```

See Also execve(2), ddi_prop_free(9F), ddi_prop_lookup(9F), ldi_prop_exists(9F) .

Writing Device Drivers

Name

ldi_putmsg, ldi_getmsg – Read/write message blocks from/to a stream

Synopsis

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

int ldi_putmsg(ldi_handle_t lh, mblk_t *smp);

int  ldi_getmsg(ldi_handle_t lh, mblk_t **rmp, timestruc_t *timeo);
```

Parameters

lh	Layered handle.
smp	Message block to send.
rmp	Message block to receive.
timeo	Optional timeout for data reception.

Description

The `ldi_putmsg` function allows a caller to send a message block to a streams device specified by the layered handle *lh*. Once the message (smp) has been passed to `ldi_putmsg()`, the caller must not free the message even if an error occurs.

The `ldi_getmsg()` function allows a caller to receive a message block from a streams device specified by the layered handle *lh*. Callers must free the message received with `freemsg(9F)`.

If a NULL timeout value is specified when the caller receives a message, the caller sleeps until a message is received.

Return Values

The `ldi_putmsg()` and `ldi_getmsg()` functions return `0` upon success. If a failure occurs before the request is passed to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

The `ldi_getmsg()` function may also return:

ETIME	Returned if the timeout `timeo` expires with no messages received.

Context

These functions may be called from user or kernel context.

Name ldi_read, ldi_write – Read and write from a device

Synopsis

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

int ldi_read(ldi_handle_t lh, struct uio *uiop, cred_t *cr);

int  ldi_write(ldi_handle_t lh, struct uio *uiop, cred_t *cr);
```

Parameters

lh	Layered handle.
cr	Pointer to a credential structure used to open a device.
uiop	Pointer to the `uio(9S)` structure. `uio(9S)` specifies the location of the read or write data. (Either userland or kernel.)

Description The `ldi_read()` function passes a read request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.

The `ldi_write()` function passes a write request to the device entry point for a device specified by the layered handle. This operation is supported for block, character, and streams devices.

Return Values The `ldi_read()` and `ldi_write()` functions return `0` upon success. If a failure occurs before the request is passed to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

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

Name ldi_remove_event_handler – remove an NDI event service callback

Synopsis

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

int ldi_remove_event_handler(ldi_handle_t lh,  ldi_callback_id_t id);
```

Interface Level Solaris DDI Specific (Solaris DDI)

Parameters

ldi_handle_t lh	Layered handle representing the device for which the event notification is requested.
ldi_callback_id_t id	Unique system-wide registration ID returned by `ldi_add_event_handler`(9F) upon successful registration.

Description The `ldi_remove_event_handler()` function removes the callback handler specified by the registration ID (*ldi_callback_id_t*). Upon successful removal, the callback handler is removed from the system and is not invoked at the event occurance.

Return Values

`DDI_SUCCESS`	Callback handler removed successfully.
`DDI_FAILURE`	Failed to remove callback handler.

Context This function can be called from user and kernel contexts only.

See Also `ldi_add_event_handler`(9F), `ldi_get_eventcookie`(9F)

Writing Device Drivers

Name ldi_strategy – Device strategy request

Synopsis

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

int ldi_strategy(ldi_handle_t lh,  struct buf *bp);
```

Parameters

lh	Layered handle.
bp	Pointer to the `buf(9S)` structure.

Description The `ldi_strategy()` function passes a strategy request to the device entry point for the device specified by the layered handle. This operation is supported for block devices.

Return Values The `ldi_strategy()` function returns `0` if the strategy request has been passed on to the target device. Other possible return values are:

EINVAL	Invalid input parameters.
ENOTSUP	Operation is not supported for this device.

Once the request has been passed on to the target devices strategy entry point, any further errors will be reported by `bioerror`(9F) and `biodone`(9F). See the `strategy`(9E) entry point for more information.

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

Name

linkb – concatenate two message blocks

Synopsis

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



void linkb(mblk_t *mp1, mblk_t *mp2);
```

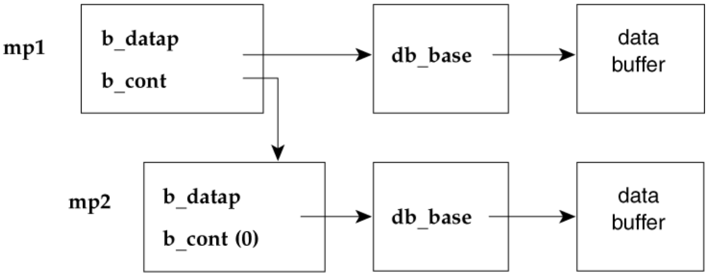
Interface Level

Architecture independent level 1 (DDI/DKI).

Description

The `linkb()` function 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`:



linkb(mp1, mp2);

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

The `linkb()` function can be called from user, interrupt, or kernel 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

Name

mac, mac_alloc, mac_free, mac_register, mac_unregister, mac_tx_update, mac_link_update, mac_rx, mac_init_ops, mac_fini_ops – MAC driver service routines

Synopsis

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

mac_register_t *mac_alloc(uint_t version);

void mac_free(mac_register_t *mregp);

int mac_register(mac_register_t *mregp, mac_handle_t *mhp);

int mac_unregister(mac_handle_t mh);

void mac_tx_update(mac_handle_t mh);

void mac_link_update(mac_handle_t mh, link_state_t new_state);

void mac_rx(mac_handle_t mh, void *reserved, mblk_t *mp_chain);

void mac_init_ops(struct dev_ops *ops, const char *name);

void mac_fini_ops(struct dev_ops *ops);
```

Parameters

version	MAC version
mh	MAC handle
mhp	pointer to a MAC handle
mregp	pointer to a mac_register_t(9S) structure
reserved	reserved argument
mp_chain	chain of message blocks containing a received packet
new_state	media link state
ops	device operations structure
name	device driver name

Interface Level

Solaris architecture specific (Solaris DDI)

Description

The `mac_alloc()` function allocates a new `mac_register`(9S) structure and returns a pointer to it. The allocated structure may contain some MAC-private elements. These private elements are initialized by the MAC layer before `mac_alloc()` returns, and the other elements of the structure are initialized to 0. The device driver must initialize the structure members as described by `mac_register` before passing a pointer to the structure to `mac_register`. The version argument should be set to `MAC_VERSION_V1`.

The `mac_free()` function frees a `mac_register` structure previously allocated by `mac_alloc()`.

The `mac_register()` function is called from the device driver's `attach`(9E) entry point, and is used to register the MAC-based device driver with the MAC layer. The `mac_register()` entry point is passed an instance of the `mac_register`structure previously allocated by `mac_alloc()`.

On success, `mac_register()` returns 0 and sets *mhp* to point to a new MAC handle corresponding to the new MAC instance. This MAC handle is subsequently passed by the driver to the framework as an argument to other MAC routines such as the ones described here. The `attach()` entry point of the driver should return `DDI_SUCCESS` in this case. On failure, `mac_register()` returns a non-zero error as described by `Intro`(2). The `attach()` entry point of the driver should return `DDI_FAILURE` in this case.

The `mac_unregister()` function is called by the driver from its `detach`(9E) entry point to unregister the instance from the MAC layer. It should pass the MAC handle which was previously obtained from `mac_register()`. `mac_unregister()` returns 0 on success, in which case the driver's `detach()` entry point should return `DDI_SUCCESS`. `mac_unregister()` returns a non-zero error as described by `Intro`(2) on failure. In this case the driver's `detach()` entry point should return `DDI_FAILURE`.

The `mac_tx_update()` function should be called by the driver to reschedule stalled outbound packets. Whenever the driver's `mc_tx`(9E) has returned a non-empty chain of packets, it must later `mac_tx_update()` to inform the MAC layer that it should retry the packets that previously could not be sent. `mac_tx_update()` should be called as soon as possible after resources are again available, to ensure that MAC resumes passing outbound packets to the driver's `mc_tx()` entry point.

The `mac_link_update()` function is called by the device driver to notify the MAC layer of changes in the media link state. The *new_state* argument must be set to one of the following:

`LINK_STATE_UP`	The media link is up.
`LINK_STATE_DOWN`	The media link is down.
`LINK_STATE_UNKNOWN`	The media link is unknown.

The `mac_rx()` function is called by the driver's interrupt handler to pass a chain of one or more packets to the MAC layer. Packets of a chain are linked with the *b_next* pointer. The driver should avoid holding mutex or other locks during the call to `mac_rx()`. In particular, locks that could be taken by a transmit thread may not be held during a call to `mac_rx()`.

The `mac_init_ops()` function must be invoked from the `_init`(9E) entry point of the device driver before a call to `mod_install`(9F). It is passed a pointer to the device driver's operations structure, and the name of the device driver.

The `mac_fini_ops()` function must be called from `_fini`(9E) before the driver is unloaded after invoking `mod_remove`(9F), or before returning from `_init()` in the case of an error returned by `mod_install()`.

Return Values The `mac_alloc()` function returns a pointer to a new `mac_register(9S)` structure.

The `mac_register()` and `mac_unregister()` functions return a non-zero error, as defined by `Intro(2)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also `Intro(2)`, `attributes(5)`, `attach(9E)`, `detach(9E)`, `_fini(9E)`, `_init(9E)`, `mac(9E)`, `mc_tx(9E)`, `mod_install(9F)`, `mod_remove(9F)`, `dev_ops(9S)`, `mac_register(9S)`

Name

mac_hcksum_get, mac_hcksum_set – hardware checksumming offload routines

Synopsis

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

void mac_hcksum_get(mblk_t *mp, uint32_t *start, uint32_t *stuff,
    uint32_t *end, uint32_t *value, uint32_t *flags);

void mac_hcksum_set(mblk_t *mp, uint32_t start, uint32_t stuff,
    uint32_t end, uint32_t value, uint32_t *flags);
```

Parameters

mp	pointer to a message block
start	offset, in bytes, from the start of the IP header to the start of the checksum span
end	offset, in bytes, from the start of the IP header to the end of the checksum span
stuff	offset, in bytes, from the start of the IP header to the checksum field in the protocol header
value	hardware computed checksum value
flags	per-packet flags indicating the hardware checksumming to be performed on outbound packets, or the hardware checksumming performed on inbound packet

Interface Level

Solaris architecture specific (Solaris DDI)

Description

Hardware checksumming allows the checksum computation to be offloaded to the network device hardware for lower CPU utilization. Hardware checksumming capabilities are advertised from the driver's `mc_getcapab(9E)` entry point. The description of `mc_getcapab()` also includes more information about the expected behavior of drivers for full and partial checksumming offload.

For received traffic, the hardware can enable hardware checksumming, and the network stack will know how to handle packets for which checksum computation or verification has been performed. The `mac_hcksum_set()` function can be used by a device driver to associate information related to the hardware checksumming performed on the packet.

The *flags* argument can be a combination of the following:

`HCK_FULLCKSUM`	The full checksum was computed, and is passed through the *value* argument.
`HCK_FULLCKSUM_OK`	The full checksum was verified in hardware and is correct.
`HCK_PARTIALCKSUM`	Partial checksum computed and passed through the *value* argument. The *start* and *end* arguments specify the checksum span.
`HCK_IPV4_HDRCKSUM_OK`	IP header checksum was verified in hardware and is correct.

`HCK_PARTIALCKSUM` is mutually exclusive with the `HCK_FULLCKSUM` and `HCK_FULLCKSUM` flags.

For outbound packets, hardware checksumming capabilities are queried via the `mc_getcapab()` entry point. Hardware checksumming is enabled by the network stack based on the `MAC_CAPAB_HCKSUM` capability. A device driver that advertised support for this capability can subsequently receive outbound packets that may not have a fully computed checksum. It is the responsibility of the driver to invoke `mac_hcksum_get()` to retrieve the per-packet hardware checksumming metadata.

`HCK_FULLCKSUM`	Compute full checksum for this packet.
`HCK_PARTIALCKSUM`	Compute partial 1's complement checksum based on the *start*, *stuff*, and offset.
`HCK_IPV4_HDRCKSUM`	Compute the IP header checksum.

`HCK_PARTIALCKSUM` is mutually exclusive with `HCK_FULLCKSUM`.

The flags `HCK_FULLCKSUM`, `HCK_FULLCKSUM_OK`, and `HCK_PARTIALCKSUM` are used for both IPv4 and IPv6 packets. The driver advertises support for IPv4 and/or IPv6 full checksumming during capability negotiation. See `mc_getcapab`(9E).

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also

`attributes`(5), `mac`(9E)

Name mac_lso_get – LSO routine

Synopsis `#include <sys/mac_provider.h>`

```
void mac_lso_get(mblk_t *mp, uint32_t *mss, uint32_t *flags);
```

Description LSO (Large Segment Offload, or Large Send Offload) allows the network stack to send larger buffers to a device driver. These large buffers can then segmented in hardware, allowing for reduced CPU utilization, PCI overhead, and reduced buffer management costs.

LSO is enabled only for device driver instances that advertise support for the `MAC_CAPAB_LSO` capability through the `mc_getcapab`(9E) entry point.

Once a device driver advertises the LSO capability, it must use the `mac_lso_get()` entry point to query whether LSO must be performed on the packet. The following values for the *flags* argument are supported:

`HW_LSO`	When set, this flag indicates that LSO is enabled for that packet. The maximum segment size (MSS) to be used during segmentation of the large segment is returned through the location pointed to by *mss*.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also `attributes`(5), `mac`(9E)

Name mac_prop_info_set_perm, mac_prop_info_set_default_uint8, mac_prop_info_set_default_str, mac_prop_info_set_default_link_flowctrl, mac_prop_info_set_range_uint32 – set property information

Synopsis

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

void mac_prop_info_set_perm(mac_prop_info_handle_t ph,
    uint8_t perm);

void mac_prop_info_set_default_uint8(mac_prop_info_handle_t ph,
    uint8_t val);

void mac_prop_info_set_default_str(mac_prop_info_handle_t ph,
    const char *str);

void mac_prop_info_set_default_link_flowctrl(
    mac_prop_info_handle_t ph, link_flowctrl_t val);

void mac_prop_info_set_range_uint32(mac_prop_info_handle_t ph,
    uint32_t min, uint32_t max);
```

Description The entry points described here are invoked from a device driver's `mc_getcapab`(9E) entry point to associate information such as default values, permissions, or allowed value ranges.

Each one of these functions takes as first argument the property information handle which is passed to `mc_propinfo()` as argument.

The `mac_prop_info_set_perm()` function specifies the property of the property. The permission is passed through the *perm* argument and can be set to one of the following values.

`MAC_PROP_PERM_READ`	The property is read-only.
`MAC_PROP_PERM_WRITE`	The property is write-only.
`MAC_PROP_PERM_RW`	The property can be read and written.

The driver is not required to call `mac_prop_info_set_perm()` for every property. If the driver does not call that function for a specific property, the framework will assume that the property has read and write permissions, corresponding to `MAC_PROP_PERM_RW`.

The `mac_prop_info_set_default_uint8()`, `mac_prop_info_set_default_str()`, and `mac_prop_info_set_default_link_flowctrl()` functions are used to associate a default value with a specific property.

The `mac_prop_info_set_range_uint32()` function is used by a driver to associate an allowed range of values for a specific property. The range is defined by the *min* and *max* arguments passed by the device driver.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also `attributes`(5), `mac`(9E)

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);
```

Interface Level

These interfaces are obsolete. `scsi_setup_cdb`(9F) should be used instead.

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

The `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, interrupt, or kernel 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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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 The `makedevice()` function 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 The `makedevice()` function can be called from user, interrupt, or kernel context.

See Also `getmajor`(9F), `getminor`(9F)

Name max – return the larger of two integers

Synopsis

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



int max(int int1, int int2);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

int1	The first integer.
int2	The second integer.

Description The `max()` function compares two signed integers and returns the larger of the two.

Return Values The larger of the two numbers.

Context The `max()` function can be called from user, interrupt, or kernel context.

See Also `min`(9F)

Writing Device Drivers

Name MBLKHEAD, MBLKIN, MBLKL, MBLKSIZE, MBLKTAIL – Message block utility macros

Synopsis

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



int MBLKHEAD(mblk_t *mp);

int MBLKTAIL(mblk_t *mp);

int MBLKSIZE(mblk_t *mp);

int MBLKL(mblk_t *mp);

int MBLKIN(mblk_t *mp, int offset, int len);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	Message to be examined.
offset	Offset from *mp->b_rptr* from which to start examining.
len	Number of bytes to examine.

Description The `MBLKHEAD()` macro calculates the number of bytes between the first byte and the first unread byte of the message block, that is: *mp->b_rptr - mp->b_datap->db_base*.

The `MBLKTAIL()` macro calculates the number of bytes between the first unwritten byte and the last byte of the message block, that is: *mp->b_datap->db_lim - mp->b_wptr*.

The `MBLKSIZE()` macros calculates the total size of the message block, that is: *mp->b_datap->db_lim - mp->b_datap->db_base*.

The `MBLKL()` macro calculates the length of the message block, that is: *mp->b_wptr - mp->b_rptr*.

The `MBLKIN()` macro checks whether the byte range specified by *offset* and *len* resides entirely within the message block.

Return Values The `MBLKHEAD()`, `MBLKTAIL()`, `MBLKL()` and `MBLKSIZE()` functions all return the appropriate byte count, as specified above. `MBLKIN()` returns non-zero if the check succeeds, or zero if it fails.

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

Notes These macros may evaluate any of their arguments more than once. This precludes passing arguments with side effects.

These macros assume the message itself is well formed, that is: *mp->b_datap->db_base <= mp->b_rptr <= mp->b_wptr <= mp->b_datap->db_lim*.

See Also `msgb(9S)`

STREAMS Programming Guide

Name mcopyin – Convert an M_IOCTL or M_IOCDATA message to an M_COPYIN

Synopsis

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



void mcopyin(mblk_t *mp, void *private, size_t size,
     void * useraddr);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	M_IOCTL or M_IOCDATA message.
private	Value to which the *cq_private* field of `copyreq`(9S) is set.
size	Value to which the *cq_size* field of `copyreq`(9S) is set.
useraddr	Optionally, the value to which the *cq_addr* field of `copyreq`(9S) is set.

Description The `mcopyin()` function converts an M_IOCTL or M_IOCDATA message into an M_COPYIN message using the supplied arguments.

To convert the message, `mcopyin()` changes the message type to M_COPYIN, and its payload from a `iocblk`(9S) to a `copyreq`(9S). Since the `iocblk`(9S) and `copyreq`(9S) are designed to overlay one another, the only fields which must be updated are *cq_private*, *cq_size*, and *cq_addr*, which are set to the supplied values. If *useraddr* is passed as NULL, *mp* must be a transparent M_IOCTL, and *cq_addr* is assigned the pointer-sized quantity found at *mp->b_cont->b_rptr*.

Any trailing message blocks associated with *mp* are freed.

Return Values None.

Context This function can be called from user, kernel or interrupt context.

See Also `mcopyout`(9F), `copyreq`(9S)

STREAMS Programming Guide

Name mcopymsg – Copy message contents into a buffer and free message

Synopsis

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



void mcopymsg(mblk_t *mp, void *buf);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	Message to be copied.
buf	Buffer in which to copy.

Description The `mcopymsg()` function copies the contents of the specified message into the specified buffer. If the message consists of more than a single message block, the contents of each message block are placed consecutively into the buffer. After copying the message contents to *buf*, `mcopymsg()` frees the message *mp*.

The provided buffer must be large enough to accommodate the message. If the buffer is not large enough, the results are unspecified. The `msgsize`(9F) function can be used to calculate the total size of the message beforehand.

Return Values None.

Context This function can be called from user, kernel or interrupt context.

See Also `freemsg`(9F), `msgsize`(9F)

STREAMS Programming Guide

Name mcopyout – Convert an M_IOCTL or M_IOCDATA message to an M_COPYOUT

Synopsis

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



void mcopyout(mblk_t *mp, void *private, size_t size, void *useraddr, mblk_t *dp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	M_IOCTL or M_IOCDATA message.
private	Value to set the cq_private field of the `copyreq`(9S) to.
size	Value to set the cq_size field of the `copyreq`(9S) to.
useraddr	Optionally, the value to set the cq_addr field of the `copyreq`(9S) to.
dp	Optionally, the payload to copy out.

Description The `mcopyout()` function converts an M_IOCTL or M_IOCDATA message into an M_COPYOUT message using the supplied arguments.

To convert the message, `mcopyout()` changes the message type to M_COPYOUT, and its payload from a `iocblk`(9S) to a `copyreq`(9S). Since the `iocblk`(9S) and `copyreq`(9S) are designed to overlay one another, the only fields which must be updated are *cq_private*, *cq_size*, and *cq_addr*, which are set to the supplied values. If *useraddr* is passed as NULL, the M_IOCTL must be transparent and *cq_addr* is assigned the pointer-sized quantity found at *mp->b_cont->b_rptr*.

If *dp* is not NULL, any trailing message blocks associated with *mp* are freed, *mp->b_cont* is reset to *dp* and *dp->b_wptr* is set to *dp->b_rptr* + *size*. Otherwise, any trailing message blocks are unaffected.

Return Values None.

Context This function can be called from user, kernel or interrupt context.

See Also `mcopyin`(9F), `copyreq`(9S), `iocblk`(9S)

STREAMS Programming Guide

Name membar_ops, membar_enter, membar_exit, membar_producer, membar_consumer – memory access synchronization barrier operations

Synopsis

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

void membar_enter(void);

void membar_exit(void);

void membar_producer(void);

void membar_consumer(void);
```

Description The `membar_enter()` function is a generic memory barrier used during lock entry. It is placed after the memory operation that acquires the lock to guarantee that the lock protects its data. No stores from after the memory barrier will reach visibility and no loads from after the barrier will be resolved before the lock acquisition reaches global visibility.

The `membar_exit()` function is a generic memory barrier used during lock exit. It is placed before the memory operation that releases the lock to guarantee that the lock protects its data. All loads and stores issued before the barrier will be resolved before the subsequent lock update reaches visibility.

The `membar_enter()` and `membar_exit()` functions are used together to allow regions of code to be in relaxed store order and then ensure that the load or store order is maintained at a higher level. They are useful in the implementation of mutex exclusion locks.

The `membar_producer()` function arranges for all stores issued before this point in the code to reach global visibility before any stores that follow. This is useful in producer modules that update a data item, then set a flag that it is available. The memory barrier guarantees that the available flag is not visible earlier than the updated data, thereby imposing store ordering.

The `membar_consumer()` function arranges for all loads issued before this point in the code to be completed before any subsequent loads. This is useful in consumer modules that check if data is available and read the data. The memory barrier guarantees that the data is not sampled until after the available flag has been seen, thereby imposing load ordering.

Return Values No values are returned.

Errors No errors are defined.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `atomic_add`(9F), `atomic_and`(9F), `atomic_bits`(9F), `atomic_cas`(9F), `atomic_dec`(9F), `atomic_inc`(9F), `atomic_ops`(9F), `atomic_or`(9F), `atomic_swap`(9F), `attributes`(5), `atomic_ops`(3C)

Notes Atomic instructions (see `atomic_ops`(9F)) ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used.

Name memchr, memcmp, memcpy, memmove, memset – Memory operations

Synopsis

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



void *memchr(const void *s, int c, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);

void *memcpy(void *restrict s1, const void *restrict s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);

void *memset(void *s, int c, size_t n);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

dst	Pointers to character strings.
n	Count of characters to be copied.
s1, *s2*	Pointers to character strings.

Description These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The `memchr()` function returns a pointer to the first occurrence of *c* (converted to an `unsigned char`) in the first *n* bytes (each interpreted as an `unsigned char`) of memory area *s*, or a null pointer if *c* does not occur.

The `memcmp()` function compares its arguments, looking at the first *n* bytes (each interpreted as an `unsigned char`), and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2* when taken to be unsigned characters.

The `memcpy()` function copies *n* bytes from memory area *s2* to *s1*. It returns *s1*. If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function copies *n* bytes from memory area *s2* to memory area *s1*. Copying between objects that overlap will take place correctly. It returns *s1*.

The `memset()` function sets the first *n* bytes in memory area *s* to the value of *c* (converted to an `unsigned char`). It returns *s*.

Usage Using `memcpy()` might be faster than using `memmove()` if the application knows that the objects being copied do not overlap.

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

See Also `bcopy`(9F), `ddi_copyin`(9F), `strcpy`(9F)

Writing Device Drivers

Name merror – Send an M_ERROR message upstream

Synopsis

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



void merror(queue_t *wq, mblk_t *mp, int error);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

wq	Write queue associated with the read queue to send the M_ERROR on.
mp	Optionally, a STREAMS message to convert to an M_ERROR.
error	Error code to include in the M_ERROR message.

Description The `merror()` function constructs an M_ERROR message, and sends the resulting message upstream.

If *mp* is NULL, `merror()` allocates a one-byte M_ERROR message. If *mp* is non-NULL, `merror()` attempts to convert the passed-in message to an M_ERROR. However, if the passed-in message has more than one reference (see `dupmsg`(9F)), or if it is of zero length, it is freed and a new message is allocated.

If the allocation or conversion fails, `merror()` silently fails. Otherwise, the resulting one-byte data block is assigned the specified error code and sent upstream.

Return Values None.

Context This function can be called from user, kernel or interrupt context.

Notes Callers must not hold any locks across an `merror()` that can be acquired as part of `put`(9E) processing.

See Also `put`(9E), `dupmsg`(9F)

STREAMS Programming Guide

Name mexchange – Exchange one message for another

Synopsis

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



mblk_t *mexchange(queue_t *wq, mblk_t *mp, size_t size,
     uchar_t type, int32_t primtype);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

wq	Optionally, write queue associated with the read queue to be used on failure (see below).
mp	Optionally, the message to exchange.
size	Size of the returned message.
type	Type of the returned message.
primtype	Optionally, a 4 byte value to store at the beginning of the returned message.

Description The `mexchange()` function exchanges the passed in message for another message of the specified *size* and *type*.

If *mp* is not NULL, is of at least *size* bytes, and has only one reference (see `dupmsg`(9F)), *mp* is converted to be of the specified *size* and *type*. Otherwise, a new message of the specified *size* and *type* is allocated. If allocation fails, and *wq* is not NULL, `merror`(9F) attempts to send an error to the stream head.

Finally, if *primtype* is not -1 and *size* is at least 4 bytes, the first 4 bytes are assigned to be *primtype*. This is chiefly useful for STREAMS-based protocols such as DLPI and TPI which store the protocol message type in the first 4 bytes of each message.

Return Values A pointer to the requested message is returned on success. NULL is returned on failure.

Context This function can be called from user, kernel or interrupt context.

See Also `dupmsg`(9F), `merror`(9F)

STREAMS Programming Guide

Name min – return the lesser of two integers

Synopsis

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



int min(int int1, int int2);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

int1	The first integer.
int2	The second integer.

Description The `min()` function compares two signed integers and returns the lesser of the two.

Return Values The lesser of the two integers.

Context The `min()` function can be called from user, interrupt, or kernel context.

See Also `max`(9F)

Writing Device Drivers

Name

mioc2ack – Convert an M_IOCTL message to an M_IOCACK message

Synopsis

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



void mioc2ack(mblk_t *mp, mblk_t *dp, size_t count, int rval);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

mp	M_IOCTL message.
dp	Payload to associate with M_IOCACK message.
count	Value to set the ioc_count of the `iocblk`(9S) to.
rval	Value to set the ioc_rval of the `iocblk`(9S) to.

Description

The `mioc2ack()` function converts an M_IOCTL message into an M_IOCACK message using the supplied arguments.

To convert the message, `mioc2ack()` changes the message type to M_IOCACK, sets the *ioc_count* and *ioc_rval* members of the `iocblk`(9S) associated with *mp* to the passed-in values, and clears the *ioc_error* field. Further, it frees any message blocks chained off of *mp->b_cont* and resets *mp->b_cont* to *dp*. Finally, if *dp* is not NULL, `mioc2ack()` resets *dp->b_wptr* to be *dp->b_rptr* + *count* (that is, it sets *dp* to be exactly *count* bytes in length).

Return Values

None.

Context

This function can be called from user, kernel or interrupt context.

See Also

`miocack`(9F), `miocnak`(9F), `iocblk`(9S)

STREAMS Programming Guide

Name miocack – Positively acknowledge an M_IOCTL message

Synopsis

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



void miocack(queue_t *wq, mblk_t *mp, intcount, int rval);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

wq	Write queue associated with the read queue to send the M_IOCACK on.
mp	M_IOCTL message.
count	Value to set the ioc_count of the `iocblk`(9S) to.
rval	Value to set the ioc_rval of the `iocblk`(9S) to.

Description The `miocack()` function converts an M_IOCTL message into a M_IOCACK message and sends the resulting message upstream.

To convert the message, `miocack()` changes the message type to M_IOCACK, sets the 'ioc_count' and 'ioc_rval' members of the `iocblk`(9S) associated with *mp* to the passed-in values, and clears the 'ioc_error' field. If the caller specifies a non-zero value for count, it is expected that the caller has already set 'mp->b_cont' field to point to a message block with a length of at least *count* bytes.

Callers that only need to perform the message conversion, or need to perform additional steps between the conversion and the sending of the M_IOCACK should use `mioc2ack`(9F).

Return Values None.

Context This function can be called from user, kernel or interrupt context.

Notes Callers must not hold any locks across a `miocack()` that can be acquired as part of `put`(9E) processing.

See Also `mioc2ack`(9F), `put`(9E), `iocblk`(9S)

STREAMS Programming Guide

Name miocnak – Negatively acknowledge an M_IOCTL message

Synopsis

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



void miocnak(queue_t *wq, mblk_t *mp, int count, int error);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

wq	Write queue associated with the read queue to send the M_IOCNAK on.
mp	M_IOCTL message.
count	Value to set the ioc_count of the `iocblk`(9S) to.
error	Value to set the ioc_error of the `iocblk`(9S) to.

Description The `miocnak()` function converts an M_IOCTL message into an M_IOCNAK message and sends the resulting message upstream.

To convert the message, `miocnak()` changes the message type to M_IOCNAK, sets the *ioc_count* and *ioc_error* members of the `iocblk`(9S) associated with *mp* to the passed-in values, and clears the *ioc_rval* field. Since payloads cannot currently be associated with M_IOCNAK messages, *count* must always be zero. If *error* is passed as zero, `EINVAL` is assumed.

Return Values None.

Context This function can be called from user, kernel or interrupt context.

Notes Callers must not hold any locks across a `miocnak()` that can be acquired as part of `put`(9E) processing.

See Also `mioc2ack`(9F), `miocack`(9F), `put`(9E), `iocblk`(9S)

STREAMS Programming Guide

Name miocpullup – Prepare the payload of an M_IOCTL message for access

Synopsis

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



int miocpullup(mblk_t *mp, size_t size);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

mp	M_IOCTL message.
size	Number of bytes to prepare.

Description The `miocpullup()` function prepares the payload of the specified M_IOCTL message for access by ensuring that it consists of at least *size* bytes of data.

If the M_IOCTL message is transparent, or its total payload is less than *size* bytes, an error is returned. Otherwise, the payload is concatenated as necessary to provide contiguous access to at least *size* bytes of data. As a special case, if *size* is zero, `miocpullup()` returns successfully, even if no payload exists.

Return Values Zero is returned on success. Otherwise an errno value is returned indicating the problem.

Context This function can be called from user, kernel or interrupt context.

See Also *STREAMS Programming Guide*

Name mkiocb – allocates a STREAMS ioctl block for M_IOCTL messages in the kernel.

Synopsis

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



mblk_t *mkiocb(uint_t command);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

command	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, interrupt, or kernel context.

Examples

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.

```
 1  test_function(queue_t *q, test_info_t *testinfo)
 2  {
 3   mblk_t *mp;
 4
 5   if ((mp = mkiocb(TEST_CMD)) == NULL)
 6       return (0);
 7
 8       /* save off ioctl ID value */
 9       testinfo->xx_iocid = ((struct iocblk *)mp->b_rptr)->ioc_id;
10
11       putnext(q, mp);       /* send message downstream */
12       return (1);
13  }
```

EXAMPLE 2 The ioctl ID Value

During the read service routine, the ioctl ID value for `M_IOCACK` or `M_IOCNAK` should equal the ioctl that was previously sent by this module before processing.

```
test_lrsrv(queue_t *q)
{
    ...

    switch (DB_TYPE(mp)) {
    case M_IOCACK:
    case M_IOCNAK:
        /* Does this match the ioctl that this module sent */
        ioc = (struct iocblk*)mp->b_rptr;
        if (ioc->ioc_id == testinfo->xx_iocid) {
            /* matches, so process the message */
            ...
            freemsg(mp);
        }
        break;
    }
    ...
}
```

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.

```
test_open(queue_t *q, dev_t devp, int oflag, int sflag, cred_t *credp)
{
     while ((mp = mkiocb(TEST_IOCTL)) == NULL) {
          int id;

          id = qbufcall(q, sizeof (union ioctypes), BPRI_HI,
              dummy_callback, 0);
          /* Handle interrupts */
          if (!qwait_sig(q)) {
              qunbufcall(q, id);
              return (EINTR);
          }
     }
     putnext(q, mp);
}
```

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_IOCACK` or `M_IOCNAK` is received.

Name

mod_install, mod_remove, mod_info – add, remove or query a loadable module

Synopsis

```
#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);
```

Interface Level

Solaris DDI specific (Solaris DDI).

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.

When `_init`(9E) is executing, its call to `mod_install()` enables other threads to call `attach`(9E) even prior to `mod_install()` returning and `_init`(9E) completion. From a programming standpoint this means that all `_init`(9E) initialization must occur prior to `_init`(9E) calling `mod_install()`. If `mod_install()` fails (non-zero return value), any initialization must be undone.

When `_fini`(9E) is executing, another thread may call `attach`(9E) prior to `_fini`(9E) calling `mod_remove()`. If this occurs, the `mod_remove()` fails (non-zero return). From a programming standpoint, this means that `_init`(9E) initilizations should only be undone after a successful return from `mod_remove()`.

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

Name msgdsize – return the number of bytes in a message

Synopsis

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



size_t msgdsize(mblk_t *mp);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

mp	Message to be evaluated.

Description The `msgdsize()` function 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.

Return Values The number of data bytes in a message, expressed as an integer.

Context The `msgdsize()` function can be called from user, interrupt, or kernel context.

Examples See `bufcall`(9F) for an example that uses `msgdsize()`.

See Also `bufcall`(9F)

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

`Non-null`	Successful completion. A pointer to the new message is returned.
`NULL`	An error occurred.

Context The `msgpullup()` function can be called from user, interrupt, or kernel context.

See Also `srv`(9E), `allocb`(9F), `pullupmsg`(9F), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

Notes The `msgpullup()` function is a DKI-compliant replacement for the older `pullupmsg`(9F) routine. Users are strongly encouraged to use `msgpullup()` instead of `pullupmsg`(9F).

Name msgsize – Return the total number of bytes in a message

Synopsis

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



size_t msgsize(mblk_t *mp);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

mp	Message to be evaluated.

Description The `msgsize()` function counts the number of bytes in a message, regardless of the data type of the underlying data blocks.

Return Values Number of bytes in the message.

Context This function can be called from user, kernel or interrupt context.

See Also `msgdsize`(9F)

STREAMS Programming Guide

Name mt-streams – STREAMS multithreading

Synopsis `#include <sys/conf.h>`

Interface Level Solaris DDI specific (Solaris DDI).

Description 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:

`D_MTPUTSHARED`	This flag modifies the default behavior when `put`(9E) procedure are invoked so that the inner perimeter is entered shared instead of exclusively.
`D_MTOCEXCL`	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. 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

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 Level Solaris DDI specific (Solaris DDI).

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, meaning that 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 interrupt priority returned from `ddi_intr_get_pri`(9F) or `ddi_intr_get_softint_pri`(9F). Note that *arg* should be the value of the interrupt priority cast by calling the `DDI_INTR_PRI` macro. 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 ensues.

`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 be sure that no other thread attempts 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.

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_intr_get_pri(hdlp, &pri);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER, DDI_INTR_PRI(pri));
ddi_intr_add_handler(hdlp, xxintr, (caddr_t)un, NULL);
```

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)
{
          ASSERT(mutex_owned(&un->un_lock));
...
```

See Also

`lockstat`(1M), `Intro`(9F), `condvar`(9F), `ddi_intr_alloc`(9F), `ddi_intr_add_handler`(9F), `ddi_intr_get_pri`(9F), `ddi_intr_get_softint_pri`(9F), `rwlock`(9F), `semaphore`(9F)

Writing Device Drivers

Notes Compiling with `_LOCKTEST` or `_MPSTATS` defined has no effect. To gather lock statistics, see `lockstat`(1M).

The address of a `kmutex_t` lock must be aligned on an 8-byte boundary for 64-bit kernels, or a 4-byte boundary for 32-bit kernels. Violation of this requirement will result in undefined behavior, including, but not limited to, failure of mutual exclusion or a system panic.

To write scalable, responsive drivers that do not hang, panic or deadlock the system, follow these guidelines:

Never return from a driver entry point with a mutex held.
Never hold a mutex when calling a service that may block, for example `kmem_alloc`(9F) with KM_SLEEP or `delay`(9F).
Always acquire mutexes in a consistent order. If a critical section acquires mutex A followed by B, and elsewhere in the driver mutex B is acquired before A, the driver can deadlock with one thread holding A and waiting for B and another thread holding B while waiting for A.
Always use a mutex to enforce exclusive access to data, not instruction paths.
Acquiring a lock in user context that is also acquired in interrupt context means that, as long as that lock is held, the driver instance holding the lock is subject to all the rules and limitations of interrupt context.
In most cases, a mutex can and should be acquired and released within the same function.
Liberal use of debugging aids like ASSERT(mutex_owned(&mutex)) can help find callers of a function which should be holding a mutex but are not. This means you need to test your driver compiled with DEBUG.
Do not use a mutex to set driver state. However, you should use a mutex to protect driver state data.
Use per-instance and automatic data where possible to reduce the amount of shared data.
Per-instance data can be protected by a per-instance lock to improve scalability and reduce contention with multiple hardware instances.
Avoid global data and global mutexes whenever possible.

Name

net_event_notify_register, net_event_notify_unregister – add/delete a function to be called for changes to a event

Synopsis

```
#include <sys/hook.h>
#include <sys/neti.h>

int net_event_notify_register(net_handle_t family, char
     *event, hook_notify_fn_t *callback, void *arg);

int net_event_notify_unregister(net_handle_t family, char
     *event, hook_notify_fn_t *callback);

typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,
     void *arg, const char *name1, const char *name2, const char
     *name3);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

family	value returned from a successful call to `net_protocol_lookup()`.
callback	function to call when a change occurs.
event	name of the event for which notification of change is desired.
arg	pointer to pass into the `callback()` function when a change occurs.

Description

The `net_event_notify_register()` function registers a function represented by the pointer *callback* to be called when there is a change to the event represented by *family*. The types of changes for which notifications are available for is currently limited to the addition and removal of hooks.

The `net_event_notify_unregister()` function indicates that there is no longer any desire to receive notification of changes to the event through function calls to the specified *callback*.

The name of a hook should be considered a private interface unless otherwise specified. The hook names used by `IPFilter` in Solaris are a public, but uncommitted, interface.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called when the *event* occurs must not block any other events.

The arguments passed through to the `callback` are as follows (the command is either `HN_REGISTER` or `HN_UNREGISTER`):

name1	is the name of the protocol.
name2	is the name of the event

name3 is the name of the hook being added/removed

Return Values If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST the given callback function is already registered.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `attributes(5)`, `net_hook_register(9F)`, `net_hook_unregister(9F)`, `net_protocol_lookup(9F)`

Name net_getifname – determine the name given to a network interface

Synopsis

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

int net_getifname(const net_data_t net, const phy_if_t ifp,
     char *buffer, size_t buflen);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup(9F)`.
ifp	value returned from a successful call to `net_phylookup(9F)` or `net_phygetnext(9F)`.
buffer	pointer to the buffer in which to write the interface name.
buflen	size of the buffer in which to store the interface name.

Description The `net_getifname()` function copies the name of the network interface into the buffer provided. The name will always be null-terminated. If the buffer is too small to fit both the interface name and the null-terminated name, the name in the buffer is truncated to fit. See `net_phygetnext(9F)` for an example on how to use this function.

Return Values The `net_getifname()` function returns:

−1	The network protocol does not support this function.
0	Successful completion.
1	Unsuccessful.

Context The `net_getifname()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_phygetnext(9F)`, `net_phylookup(9F)`, `net_protocol_lookup(9F)`

Name

net_getlifaddr – determine a network address for a given interface

Synopsis

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

int net_getlifaddr(const net_data_t net, const phy_if_t ifp,
     const net_if_t lif, int const type,
     struct sockaddr* storage);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
ifp	value returned from a successful call to `net_phylookup`(9F) or `net_phygetnext`(9F), indicating which network interface the information should be returned from.
lif	indicates the logical interface from which to fetch the address.
type	indicates what type of address should be returned. See below for more details on this field.
storage	pointer to an area of memory to store the address data.

Description

The `net_getlifaddr()` function retrieves the address information for each logical interface. Each call to `net_getlifaddr()` requires that the caller pass a pointer to an array of address information types to retrieve, and an accompanying pointer to an array of pointers to `struct sockaddr` structures to which to copy the address information. See `net_lifgetnext`(9F) for an example on how to use this function.

Each member of the address `type` array should be one of the values listed here.

`NA_ADDRESS`	Return the network address associated with the logical interface (`lif`) that belongs to the network interface (`ifp`).
`NA_PEER`	Return the address assigned to the remote host for point to point network interfaces for the given network/logical interface.
`NA_BROADCAST`	Return the broadcast address assigned to the given network/logical interface for network interfaces that support broadcast packets.
`NA_NETMASK`	Return the netmask associated with the given network/logical interface for network interfaces that support broadcast packets.

Return Values

The `net_getlifaddr()` function returns:

−1	The network protocol does not support this function.
0	Successful completion.
1	Unsuccessful.

Context The `net_getifaddr()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_lifgetnext(9F)`, `net_phylookup(9F)`, `net_phygetnext(9F)net_protocol_lookup(9F)`

Name net_getmtu – determine the MTU of a given network interface

Synopsis

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

int net_getmtu(const net_data_t net, const phy_if_t ifp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
ifp	value returned from a successful call to `net_phylookup`(9F) or `net_phygetnext`(9F).

Description The `net_getmtu()` function receives information about the current MTU of a network interface. The value returned from this function call should not be cached as the MTU of a network interface since it is not guaranteed to be constant.

Return Values The `net_getmtu()` function returns -1 if the network protocol does not support this feature and otherwise returns the current MTU of the network interface.

Context The `net_getmtu()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_phygetnext`(9F), `net_phylookup`(9F), `net_protocol_lookup`(9F)

Name net_getnetid – returns the instance identifier

Synopsis

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

netid_t net_getnetid(const net_data_t net);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).

Description The `net_getnetid()` function returns the instance identifier for the protocol instance returned via a call to `net_protocol_lookup`(9F).

Return Values The `net_getnetid()` function returns the value of the instance identifier.

Context The `net_getnetid()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_protocol_lookup`(9F)

Name net_getpmtuenabled – determine if path MTU discovery is enabled for a network protocol

Synopsis

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

int net_getpmtuenabled(const net_data_t net);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).

Description The `net_getpmtuenabled()` function returns a value to indicate whether or not path MTU (PMTU) discovery is enabled for this network protocol.

Return Values The `net_getpmtuenabled()` function returns:

−1	The network protocol does not support this function.
0	PATH MTU discovery is disabled.
1	PATH MTU discovery is enabled.

Context The `net_getpmtuenabled()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_getmtu`(9F), `net_protocol_lookup`(9F)

Name

net_hook_register – add a hook to be called in event processing

Synopsis

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

net_hook_t net_hook_register(const net_data_t net, hook_t *hook);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_register()`.
hook	pointer to a `hook_t` structure.

Description

The `net_hook_register()` function uses hooks that allow callbacks to be registered with events that belong to a network protocol. A successful call to `net_hook_register()` requires that a valid handle for a network protocol be provided (the *net* parameter), along with a hook description that includes a reference to an available event.

While it is possible to use the same `hook_t` structure with multiple calls to `net_hook_register()`, it is not encouraged.

The `hook_t` structure passed in with this function is described by `hook_t`(9S). The following describes how this structure is used.

`h_func`	Must be non-NULL and represent a function that fits the specified interface.
`h_name`	Gives the hook a name that represents its owner. No duplication of `h_name` among the hooks present for an event is allowed.
`h_flags`	Currently unused and must be set to 0.
`h_hint, h_hintvalue`	Specify a hint to `net_hook_register()` on how to insert this hook. If the hint cannot be specified, then an error is returned.
`h_arg;`	May take any value that the consumer wishes to have passed back when the hook is activated.

Return Values

If the `net_hook_register()` function succeeds, 0 is returned. Otherwise, one of the following errors is returned:

ENOMEM	The system cannot allocate any more memory to support registering this hook.
ENXIO	A hook cannot be found among the given family of events.
EEXIST	A hook with the given `h_name` already exists on that event.
ESRCH	A before or after dependency cannot be satisfied due to the hook with
EBUSY	The `h_hint` field specifies a hint that cannot currently be satisfied because it conflicts with another hook. An example of this might be specifying `HH_FIRST` or `HH_LAST` when another hook has already been registered with this value.

Context The `net_hook_register()` function may be called from user or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_hook_unregister`(9F), `hook_t`(9S)

Name net_hook_unregister – disable a hook that was called in event processing

Synopsis

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

int net_hook_unregister(const net_data_t net, nethook_t hook);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_register()`.
hook	value returned from a successful call to `net_hook_register`(9F).

Description The `net_hook_unregister()` function disables the callback hooks that were registered with the `net_hook_register()` function.

Return Values If the `net_hook_unregister()` function succeeds, 0 is returned. Otherwise, an error indicating the problem encountered.

Context The `net_hook_unregister()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_hook_register`(9F)

Name netinfo – interface to network data/functionality

Description The `net_getnetid()` interface is designed to provide the framework for accessing functionality and data within an implementation of a network layer protocol (OSI layer 3.) A protocol may or may not provide full coverage for each of the functions that is described within this interface. Where it does not, it must return an appropriate error condition for that call. Documentation pertaining to the network protocol, as found in man page section 7pP, must list which functions provided by this interface are and are not supported.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes`(5), `hook_alloc`(9F), `hook_free`(9F), `hook_nic_event`(9S), `hook_pkt_event`(9S), `hook_t`(9S), `net_getifname`(9F), `net_getlifaddr`(9F), `net_getmtu`(9F), `net_getnetid`(9F), `net_getpmtuenabled`(9F), `net_hook_register`(9F), `net_hook_unregister`(9F), `net_inject`(9F), `net_inject_alloc`(9F), `net_inject_free`(9F), `net_inject_t`(9S), `net_instance_alloc`(9F), `net_instance_free`(9F), `net_instance_register`(9F), `net_instance_register`(9F), `net_inject_t`(9S), `net_ispartialchecksum`(9F), `net_isvalidchecksum`(9F), `net_kstat_create`(9F), `net_lifgetnext`(9F), `net_phygetnext`(9F), `net_phylookup`(9F), `net_protocol_lookup`(9F), `net_protocol_release`(9F), `net_protocol_walk`(9F), `net_routeto`(9F), `net_zoneidtonetid`(9F)

Name net_inject – determine if a network interface name exists for a network protocol

Synopsis

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

int net_inject(const net_data_t net, inject_t style,
      net_inject_t *packet);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
style	method that determines how this packet is to be injected into the network or kernel.
packet	details about the packet to be injected.

Description The `net_inject()` function provides an interface to allow delivery of network layer (layer 3) packets either into the kernel or onto the network. The method of delivery is determined by style.

If `NI_QUEUE_IN` is specified, the packet is scheduled for delivery up into the kernel, imitating its reception by a network interface. In this mode, `packet->ni_addr` is ignored and `packet->ni_physical` specifies the interface for which the packet is made to appear as if it arrived on.

If `NI_QUEUE_OUT` is specified, the packet is scheduled for delivery out of the kernel, as if it were being sent by a raw socket. In this mode, `packet->ni_addr` and `packet->ni_physical` are both ignored.

Neither `NI_QUEUE_IN` or `NI_QUEUE_OUT` cause the packet to be immediately processed by the kernel. Instead, the packet is added to a list and a timeout is scheduled (if there are none already pending) to deliver the packet. The call to `net_inject()` returns once the setup has been completed, and not after the packet has been processed. The packet processing is completed on a different thread and in a different context to that of the original packet. Thus, a packet queued up using `net_inject()` for either `NI_QUEUE_IN` or `NI_QUEUE_OUT` is presented to the packet event again. A packet received by a hook from `NH_PHYSICAL_IN` and then queued up with `NI_QUEUE_IN` is seen by the hook as another `NH_PHYSICAL_IN` packet. This also applies to both `NH_PHYSICAL_OUT` and `NI_QUEUE_OUT` packets.

If `NI_DIRECT_OUT` is specified, an attempt is made to send the packet out to a network interface immediately. No processing on the packet, aside from prepending any required layer 2 information, is made. In this instance, `packet->ni_addr` may be used to specify the next hop (for the purpose of link layer address resolution) and `packet->ni_physical` determines which interface the packet should be sent out.

For all three packets, `packet->ni_packet` must point to an `mblk` structure with the packet to be delivered.

See `net_inject_t`(9S) for more details on the structure `net_inject_t`.

Return Values The `net_inject()` function returns:

−1	The network protocol does not support this function.
0	The packet is successfully queued or sent.
1	The packet could not be queued up or sent out immediately.

Context The `net_inject()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_protocol_lookup`(9F), `netinfo`(9F), `net_inject_t`(9S)

Name net_inject_alloc – allocate a net_inject_t structure

Synopsis

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

net_inject_t *net_inject_alloc(const int version);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *version* must always be the symbol `NETI_VERSION`.

Description The `net_inject_alloc()` function allocates a `net_inject_t` structure, returning a pointer for the caller to use.

Return Values Upon success, `net_inject_alloc()` returns a pointer to the allocated *net_inject_t* structure. On failure, `hook_alloc()` returns a `NULL` pointer.

Context The `net_inject_alloc()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_inject_free`(9F), `net_inject_t`(9S)

Name net_inject_free – free a net_inject_t structure

Synopsis

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

void net_inject_free(net_inject_t *inject);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *inject* pointer returned by `net_inject_alloc`(9F).

Description The `net_inject_free()` function frees a `net_inject_t` structure that was originally allocated by `net_inject_alloc`(9F).

Context The `net_inject_free()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_inject_alloc`(9F), `attributes`(5), `net_inject_t`(9S)

Name net_instance_alloc – allocate a net_instance_t structure

Synopsis

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

net_instance_t *net_instance_alloc(const int version);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *version* must always be the symbol `NETI_VERSION`.

Description The `net_instance_alloc()` function allocates a `net_instance_t` structure, returning a pointer for the caller to use.

Return Values Upon success, `net_instance_alloc()` returns a pointer to the allocated *net_instance_t* structure. On failure, it returns a `NULL` pointer.

Context The `net_instance_alloc()` function may be called from user or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_instance_free`(9F), `attributes`(5), `net_inject_t`(9S)

Name net_instance_free – free a net_instance_t structure

Synopsis

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

void net_instance_free(net_instance_t *net_instance);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net_instance	pointer returned by `net_instance_alloc`(9F).

Description The `net_instance_free()` function frees a `net_instance_t` structure that was originally allocated by `net_instance_alloc`(9F).

Context The `net_instance_free()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_instance_alloc`(9F), `attributes`(5), `net_instance_t`(9S)

Name

net_instance_notify_register, net_instance_notify_unregister – add/delete a function to be called for changes to an instance

Synopsis

```
#include <sys/hook.h>
#include <sys/neti.h>

int net_instance_notify_register(net id_t net_id,
     hook_notify_fn_t *callback, void *arg);

int net_instance_notify_unregister(net id_t net_id,
     hook_notify_fn_t *callback);

typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,
     void *arg, const char *name1, const char *name2, const char
     *name3);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

netid	value from either *callback* registered with `net_instance_register()` or `net_zoneidtonetid()`.
callback	function to call when a change occurs.
arg	pointer to pass into the `callback()` function when a change occurs.

Description

The `net_instance_notify_register()` function registers a function represented by the pointer *callback* to be called when there is a new instance added or removed from the given network instance (represented by *netid*.)

The `net_instance_notify_unregister()` function indicates that there is no longer any desire to receive notification of changes to the instance through function calls to the specified *callback*.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called must not block any other events.

The arguments passed through to the `callback` are as follows (the command is either `HN_REGISTER` or `HN_UNREGISTER`):

name1	is the *netid* represented as a string.
name2	is `NULL`.
name3	is the name of the instance being added/removed

Return Values If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST	the given callback function is already registered.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `attributes(5)`, `net_instance_register(9F)`, `net_instance_unregister(9F)`, `net_zoneidtonetid(9F)`

Name net_instance_register – register a set of instances to occur with IP instance events

Synopsis

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

int net_instance_register(net_instance_t *instances);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

instances	must be a pointer returned by `net_instance_alloc`(9F).

Description The `net_instance_register()` function attempts to record the set of functions passed by instances that are to be called when an event related to IP instance maintenance occurs.

Return Values If the `net_instance_register()` function succeeds, `DDI_SUCCESS` is returned. Otherwise, `DDI_FAILURE` is returned to indicate failure due to the name in the instance already being present.

Context The `net_instance_register()` function may be called from user or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_instance_alloc`(9F), `net_instance_unregister`(9F), `attributes`(5), `net_instance_t`(9S)

Name net_instance_unregister – disable a set of instances

Synopsis

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

void net_instance_unregister(net_instance_t *instances);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *instances* must be a pointer returned by `net_instance_alloc`(9F).

Description The `net_instance_unregister()` function removes the set of instances that were previously registered with the `net_instance_register()` function.

Return Values If the `net_instance_unregister()` function succeeds, 0 is returned. Otherwise, an error indicating the problem encountered.

Context The `net_instance_unregister()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_instance_alloc`(9F), `net_instance_register`(9F), `attributes`(5), `net_instance_t`(9S)

Name net_ispartialchecksum – indicate if a packet is being scheduled for hardware checksum calculation

Synopsis

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

int net_ispartialchecksum(const net_data_t net, mblk_t *mb);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
mb	the `mblk` structure holding a packet that is the subject of this query.

Description The `net_ispartialchecksum()` function looks at the fields within the `mblk` structure to determine if the packet contained inside contains headers with only partial checksum values. Partial checksum values are stored inside headers when the calculation of the complete checksum is being handled by the hardware.

Return Values The `net_ispartialchecksum()` function returns:

–1	The network protocol does not support this function.
0	The packet does not contain partial checksums.

If a packet is marked for hardware checksum'ing, the following values are returned:

`NET_HCK_L3_FULL`	Complete layer 3 checksum calculated
`NET_HCK_L3_PART`	Partial layer 3 checksum calculated
`NET_HCK_L4_FULL`	Complete layer 4 checksum calculated
`NET_HCK_L4_PART`	Partial layer 4 checksum calculated

Context The `net_ispartialchecksum()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_isvalidchecksum`(9F), `net_protocol_lookup`(9F), `attributes`(5)

Name net_isvalidchecksum – verify layer 3 and layer 4 checksums

Synopsis

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

int net_isvalidchecksum(const net_data_t net, mblk_t *mb);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup(9F)`.
mb	the `mblk` structure holding a packet that is the subject of this query.

Description The `net_isvalidchecksum()` function verifies the layer 3 checksum (and, in some case, the layer 4 checksum) in the packet. If possible, fields that are used by hardware checksum'ing are examined rather than manually verifying that the checksums are present for packets received from a network interface.

For both IPv4 and IPv6, TCP, UDP and ICMP (including ICMPV6 for IPv6) layer 4 checksums are currently validated.

Return Values The `net_isvalidchecksum()` function returns:

−1	The network protocol does not support this function.
0	The packet does not contain partial checksums.
1	The packet does contain partial checksums.

Context The `net_isvalidchecksum()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_ispartialchecksum(9F)`, `net_protocol_lookup(9F)`, `attributes(5)`

Name net_kstat_create – create and initialize a new kstat for a specific instance of IP

Synopsis

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/neti.h>

kstat_t *net_kstat_create(netid_t netid, char *module,
     int instance, char *name, char *class, uchar_type type,
     ulong_t ndata, uchar_t ks_flag);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

netid IP instance identifier.

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.

ks_flag A bit-field of various flags for this `kstat`. *ks_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 no mounted partitions.) Note: Persistent `kstats` cannot be virtual, since `ks_data` points to garbage as soon as the driver goes away.

Description

The `net_kstat_create()` function allocates and initializes a `kstat`(9S) structure. See `kstat_create`(9F) for a complete discussion of this function.

Return Values

If successful, `net_kstat_create()` returns a pointer to the allocated `kstat`. `NULL` is returned upon failure.

Context

The `net_kstat_create()` function may be called from user or kernel context.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also

`ddi_get_instance`(9F), `kstat_create`(9F), `kstat_delete`(9F), `hook_t`(9S), `kstat_named`(9S)

Name net_lifgetnext – search through a list of logical network interfaces

Synopsis

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

net_if_t net_lifgetnext(const net_data_t net, const phy_if_t ifp,
     net_if_t lif);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
ifp	value returned from a successful call to `net_phylookup`(9F) or `net_phygetnext`(9F).
lif	value returned from a successful call to this function.

Description The `net_lifgetnext()` function is used to search through all of the logical interfaces that are associated with a physical network interface. To start searching through this list, a value of 0 should be passed through as the value of `lif`. When 0 is returned by this function, the last of the interfaces owned by this protocol has been reached.

When called successfully, the value returned represents a logical interface that exists, at the time of the call, within the scope of the network interface and its assigned network protocol. This value is only guaranteed to be unique for a name within the scope of the network interface and its assigned protocol.

Examples

```
   net_data_t net;
    phy_if_t ifp;
    net_if_t lif;
    char buffer[32];
    net_ifaddr_t atype[1];
    struct sockaddr_in sin[1];

    net = net_protocol_lookup("inet");

    if (net != NULL) {
         atype[0] = NA_ADDRESS;
         ifp = net_phylookup(net, "hme0");
         for (lif = net_lifgetnext(net, 0); lif != 0;
               lif = net_lifgetnext(net, lif)) {
               /* Do something with lif */
               if (net_getlifaddr(net, ifp, lif, 1, atype, sin) == 0)
                     printf("hme0:%d %x0, lif,
                         ntohl(sin[0].sin_addr.s_addr));
          }
      }
```

Return Values The `net_lifgetnext()` function returns a value of -1 if it is not supported by the network protocol and a value of 0 if an attempt to go beyond the last network interface is made. Otherwise, it returns a value representing a network interface.

Context The `net_lifgetnext()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_phygetnext(9F)`, `net_phylookup(9F)`

Name

net_phygetnext – search through the current list of network interfaces

Synopsis

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

phy_if_t net_phygetnext(const net_data_t net, const phy_if_t ifp);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup(9F)`.
ifp	value returned from a successful call to this function or `net_phylookup(9F)`.

Description

The `net_phygetnext()` function searches through all of the network interfaces that a network protocol "owns". To start searching through all of the interfaces owned by a protocol, a value of 0 should be passed through as the value of `ifp`. When 0 is returned by this function, the last of the interfaces owned by this protocol has been reached.

When called successfully, the value returned represents a network interface that exists, at the time of the call, within the scope of the network interface. This value is only guaranteed to be unique for a name within the scope of the network protocol.

Examples

```
net_data_t net;
 phy_if_t ifp;
 char buffer[32];

 net = net_protocol_lookup("inet");

 if (net != NULL) {
      for (ifp = net_phygetnext(net, 0); ifp != 0;
           ifp = net_phygetnext(net, ifp)) {
           /* Do something with ifp */
           if (net_getifname(net, ifp, buffer,
                sizeof(buffer) >= 0)
                 printf("Interface %s0, buffer);
       }
   }
```

Return Values

The `net_phygetnext()` function returns -1 if it is not supported by the network protocol or 0 if an attempt to go beyond the last network interface is made. Otherwise, it returns a value representing a network interface.

Context

The `net_phygetnext()` function may be called from user, kernel, or interrupt context.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `net_phylookup`(9F), `net_protocol_lookup`(9F), `attributes`(5)

Name net_phylookup – determine if a network interface name exists for a network protocol

Synopsis

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

phy_if_t net_phylookup(const net_data_t net, const char *name);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
name	name of the network interface to find.

Description The `net_phylookup()` function attempts to resolve the interface name passed in with the network protocol.

When called successfully, the value returned represents a network interface that exists, at the time of the call, within the scope of the network interface. This value is only guaranteed to be unique for a name within the scope of the network protocol.

Return Values The `net_phylookup()` function returns -1 if it is not supported by the network protocol, and 0 if the named network interface does not exist (or is otherwise unknown). Otherwise, it returns a value greater than 0 representing a network interface that currently exists within the scope of this network protocol.

Context The `net_phylookup()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_getifname`(9F), `net_phygetnext`(9F), `net_protocol_lookup`(9F), `attributes`(5)

Name net_protocol_lookup – locate an implementation of a network layer protocol

Synopsis

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

net_data_t net_protocol_lookup(netid_t id, const char *protocol);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

id	network instance identifier.
family	name of the network protocol to find.

Description The `net_protocol_lookup()` function attempts to locate a data structure that defines what capabilities it is exporting through this interface. The value returned by this call is guaranteed to be valid until it is passed into a call to `net_protocol_release`(9F), after which it should no longer be treated as valid.

The protocol must be a registered name of a network protocol that has been registered. The symbols NHF_INET and NHF_INET6 should be passed to `net_protocol_lookup()` as the protocol name to gain access to either IPv4 or IPv6 respectively.

Return Values The `net_protocol_lookup()` function returns NULL if it does not find any knowledge about the network protocol referenced. Otherwise, it returns a value that can be used with other calls in this framework.

Context The `net_protocol_lookup()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_protocol_release`(9F), `attributes`(5)

Name

net_protocol_notify_register, net_instance_protocol_unregister – add/delete a function to be called for changes to a protocol

Synopsis

```
#include <sys/hook.h>
#include <sys/neti.h>

int net_protocol_notify_register(net_handle_t family,
     hook_notify_fn_t *callback, void *arg);

int net_protocol_notify_unregister(net_handle_t family,
     hook_notify_fn_t *callback);

typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,
     void *arg, const char *name1, const char *name2, const char
     *name3);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

family	value returned from `net_protocol_lookup()`.
callback	function to call when a change occurs.
arg	pointer to pass into the `callback()` function when a change occurs.

Description

The `net_protocol_notify_register()` function registers a function represented by the pointer *callback* to be called when there is a change to the protocol represented by *family*. The types of changes for which notifications are available for is currently limited to the addition and removal of protocols.

The `net_protocol_notify_unregister()` function removes the function specified by the pointer *callback* from the list of functions to call. This call may fail if the specified function cannot be found.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called must not block any other protocols.

The arguments passed through to the `callback` are as follows (the command is either `HN_REGISTER` or `HN_UNREGISTER`):

name1	is the name of the protocol
name2	is `NULL`.
name3	is the name of the protocol being added/removed

Return Values If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST	the given callback function is already registered.

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `attributes(5)`, `net_protocol_lookup(9F)`

Name net_protocol_release – indicate that a reference to a network protocol is no longer required

Synopsis

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

int net_protocol_release(net_data_t *net);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).

Description The `net_protocol_release()` function indicates to the network information framework that the caller is no longer interested in any knowledge about the network protocol to which the parameter being passed through applies.

Return Values The `net_protocol_release()` function returns:

−1	The value passed in is unknown to this framework.
0	Successful completion.
1	Unsuccessful because this function has been called too many times.

Context The `net_protocol_release()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_protocol_lookup`(9F), `net_protocol_walk`(9F), `attributes`(5)

Name net_protocol_walk – step through the list of registered network protocols

Synopsis

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

net_data_t *net_protocol_walk(net_data_t net);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).

Description The `net_protocol_walk()` function walks through all of the network protocols that have been registered with this interface. The initial call to `net_protocol_walk()` should be made by passing in NULL as the value for *net*. When this function returns NULL, the end of the list has been reached.

A caller of `net_protocol_walk()` is required to walk through the entire list of network protocols, until NULL is returned or, when finished with using the value returned, pass it into a call to `net_protocol_release`(9F).

Return Values The `net_protocol_walk()` function returns NULL when the end of the list is returned. Otherwise, it returns a non-NULL value as a token for being passed into other function calls within this interface.

Context The `net_protocol_walk()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_protocol_lookup`(9F), `net_protocol_release`(9F), `attributes`(5)

Name net_routeto – indicate which network interface packets are sent

Synopsis

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

phy_if_t net_routeto(const net_data_t *net, struct sockaddr *address,
     struct sockaddr *nexthop);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

net	value returned from a successful call to `net_protocol_lookup`(9F).
address	network address to find a path out of the machine for.
nexthop	pointer to the `sockaddr` structure in which to store the address of the next hop. If this information is not required, the value NULL may be passed instead.

Description The `net_routeto()` function indicates which network interface packets destined for a particular address would be sent out of, according to the systems network routing tables. If `next` is supplied as a non-NULL pointer, the IP address of the `nexthop` router to be used is returned in it.

Return Values The `net_routeto()` function returns:

−1	The network protocol does not support this function.
0	This function cannot find a route for the address given.
>0	Indicates which network interface can be used to reach the given address.

Context The `net_routeto()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `net_phygetnext`(9F), `net_phylookup`(9F), `net_protocol_lookup`(9F), `attributes`(5)

Name net_zoneidtonetid – map a zoneid_t struture identifier to a netid_t structure

Synopsis

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

netid_t net_zoneidtonetid(const zoneid_t zone);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters *zone* valid `zoneid_t` structure that refers to a running zone.

Description The `net_zoneidtonetid()` function maps the given `zoneid_t` structure (used to represent a zone that is currently running) into a `netid_t` structure that is associated with the IP instance supporting network functions for that zone.

Return Values The `net_zoneidtonetid()` function returns -1 if no mapping took place. Otherwise, it returns the `netid_t` structure currently used by the `zoneid_t` structure. For zones that are using a shared IP instance, the `netid_t` structure for the instance owned by the global zone is returned.

Context The `net_zoneidtonetid()` function may be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu
Interface Stability	Committed

See Also `attributes(5)`

Name nochpoll – error return function for non-pollable devices

Synopsis

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



int nochpoll(dev_t dev, short events, int anyyet, short *reventsp, struct pollhead **pollhdrp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

dev	Device number.
events	Event flags.
anyyet	Check current events only.
reventsp	Event flag pointer.
pollhdrp	Poll head pointer.

Description The `nochpoll()` function 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 The `nochpoll()` function returns `ENXIO`.

Context The `nochpoll()` function can be called from user, interrupt, or kernel context.

See Also `poll`(2), `chpoll`(9E), `cb_ops`(9S)

Writing Device Drivers

Name nodev – error return function

Synopsis

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



int nodev( );
```

Interface Level Architecture independent level 1 (DDI/DKI).

Description `nodev()` 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.

Return Values `nodev()` returns `ENXIO`.

Context `nodev()` can be only called from user context.

See Also `nulldev`(9F), `cb_ops`(9S)

Writing Device Drivers

Name noenable – prevent a queue from being scheduled

Synopsis

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



void noenable(queue_t *q);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters *q* Pointer to the queue.

Description The `noenable()` function prevents the *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 The `noenable()` function can be called from user, interrupt, or kernel context.

See Also `enableok`(9F), `insq`(9F), `putbq`(9F), `putq`(9F), `qenable`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name nulldev – zero return function

Synopsis

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



int nulldev();
```

Interface Level Architecture independent level 1 (DDI/DKI).

Description `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.

Return Values `nulldev()` returns a `0`.

Context `nulldev()` can be called from any context.

See Also `nodev`(9F), `cb_ops`(9S)

Writing Device Drivers

Name nvlist_add_boolean, nvlist_add_boolean_value, nvlist_add_byte, nvlist_add_int8, nvlist_add_uint8, nvlist_add_int16, nvlist_add_uint16, nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64, nvlist_add_string, nvlist_add_nvlist, nvlist_add_nvpair, nvlist_add_boolean_array, nvlist_add_int8_array, nvlist_add_uint8_array, nvlist_add_nvlist_array, 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, nvlist_t – value pair functions

Synopsis

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

int nvlist_add_boolean(nvlist_t *nvl, const char *name);

int nvlist_add_boolean_value(nvlist_t *nvl, const char *name, boolean_t val);

int nvlist_add_byte(nvlist_t *nvl, const char *name, uchar_t val);

int nvlist_add_int8(nvlist_t *nvl, const char *name, int8_t val);

int nvlist_add_uint8(nvlist_t *nvl, const char *name, uint8_t val);

int nvlist_add_int16(nvlist_t *nvl, const char *name, int16_t val);

int nvlist_add_uint16(nvlist_t *nvl, const char *name, uint16_t val);

int nvlist_add_int32(nvlist_t *nvl, const char *name, int32_t val);

int nvlist_add_uint32(nvlist_t *nvl, const char *name, uint32_t val);

int nvlist_add_int64(nvlist_t *nvl, const char *name, int64_t val);

int nvlist_add_uint64(nvlist_t *nvl, const char *name, uint64_t val);

int nvlist_add_string(nvlist_t *nvl, const char *name, char *val);

int nvlist_add_nvlist(nvlist_t *nvl, const char *name, nvlist_t *val);

int nvlist_add_nvpair(nvlist_t *nvl, nvpair_t *nvp);

int nvlist_add_boolean_array(nvlist_t *nvl, const char *name, boolean_t *val,
     uint_t nelem);

int nvlist_add_byte_array(nvlist_t *nvl, const char *name, uchar_t *val,
     uint_t nelem);

int nvlist_add_int8_array(nvlist_t *nvl, const char *name, int8_t *val,
     uint_t nelem);

int nvlist_add_uint8_array(nvlist_t *nvl, const char *name, uint8_t *val uint_t nelem);

int nvlist_add_int16_array(nvlist_t *nvl, const char *name, int16_t *val,
     uint_t nelem);

int nvlist_add_uint16_array(nvlist_t *nvl, const char *name, uint16_t *val,
     uint_t nelem);
```

```
int nvlist_add_int32_array(nvlist_t *nvl, const char *name, int32_t *val,
     uint_t nelem);

int nvlist_add_uint32_array(nvlist_t *nvl, const char *name, uint32_t *val,
     uint_t nelem);

int nvlist_add_int64_array(nvlist_t *nvl, const char *name, int64_t *val,
     uint_t nelem);

int nvlist_add_uint64_array(nvlist_t *nvl, const char *name, uint64_t *val,
     uint_t nelem);

int nvlist_add_string_array(nvlist_t *nvl, const char *name, const *char *val,
     uint_t nelem);

int nvlist_add_nvlist_array(nvlist_t *nvl, const char *name, nvlist_t **val,
     uint_t nelem);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

nvl	The `nvlist_t` to be processed.
nvp	The `nvpair_t` (name-value pair) 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 add a new name-value pair to an `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 follows 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.

Multiple threads can simultaneously read the same nvlist_t, but only one thread may actively change a given nvlist_t at a time. The caller is responsible for the synchronization.

The `nvlist_add_boolean()` function is deprecated and the `nvlist_add_boolean_value()` function is used instead.

Return Values

`0`	success
`EINVAL`	invalid argument
`ENOMEM`	insufficient memory

Context

These functions can be called from interrupt context only if (1) the default allocator is used and the `KM_NOSLEEP` flag is set, or (2) the specified allocator did not sleep for free memory (for example, if it uses a pre-allocated buffer for memory allocations).

See `nvlist_alloc`(9F) for a description of pluggable allocators and KM_NOSLEEP. These functions can be called from user or kernel context in all cases.

Name nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup, nv_alloc_init, nv_alloc_fini, nvlist_xalloc, nvlist_xpack, nvlist_xunpack, nvlist_xdup, nvlist_merge – Manage a name-value pair list

Synopsis

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

List Manipulation:

int nvlist_alloc(nvlist_t **nvlp, uint_t nvflag,
     int kmflag);

int nvlist_xalloc(nvlist_t **nvlp, uint_t nvflag, nv_alloc_t *nva);

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 flag);

int nvlist_xpack(nvlist_t *nvl, char **bufp, size_t *buflen, int encoding,
     nv_alloc_t *nva);

int nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvlp, int flag);

int nvlist_xunpack(char *buf, size_t buflen, nvlist_t **nvlp, nv_alloc_t *nva);

int nvlist_dup(nvlist_t *nvl, nvlist_t **nvlp, int flag);

int nvlist_xdup(nvlist_t *nvl, nvlist_t **nvlp, nv_alloc_t *nva);

int nvlist_merge(nvlist_t *dst, nvlist_t *nvl, int flag);

Pluggable Allocator Configuration:

nv_alloc_t *nvlist_lookup_nv_alloc(nvlist_t *);

int nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t * nvo,/* args */ ...);

void  nv_alloc_reset(nv_alloc_t  *nva);

void  nv_alloc_fini(nv_alloc_t *nva);

Pluggable Allocation Initialization with Fixed Allocator:

int nv_alloc_init(nv_alloc_t *nva,
     nv_fixed_ops, void * bufptr,  size_t sz);
```

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`	`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	`nvlist_t` to be processed.
dst	Destination `nvlist_t`.
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.
nvo	Pluggable allocator operations pointer (nv_alloc_ops_t).
nva	Points to a nv_alloc_t structure to be used for the specified `nvlist_t`.

Description List Manipulation:

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.

If `NV_UNIQUE_NAME` is specified for nvflag, existing nvpairs with matching names are removed before the new nvpair is added. If `NV_UNIQUE_NAME_TYPE` is specified for nvflag, existing nvpairs with matching names and data types are removed before the new nvpair is added. See `nvlist_add_byte`(9F) for more details.

The `nvlist_xalloc()` function differs from `nvlist_alloc()` in that `nvlist_xalloc()` can use a different allocator, as described in the Pluggable Allocators section.

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 allocates memory and updates ***bufp* to point to the memory and updates ***buflen* to contain the size of the allocated memory. The value of *kmflag* indicates the memory allocation policy

The `nvlist_xpack()` function differs from `nvlist_pack()` in that `nvlist_xpack()` can use a different allocator.

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_xunpack()` function differs from `nvlist_unpack()` in that `nvlist_xunpack()` can use a different allocator.

The `nvlist_dup()` function makes a copy of *nvl* and updates *nvlp* to point to the copy.

The `nvlist_xdup()` function differs from `nvlist_dup()` in that `nvlist_xdup()` can use a different allocator.

The `nvlist_merge()` function adds copies of all name-value pairs from `nvlist_t` *nvl* to `nvlist_t dst`. Name-value pairs in dst are replaced with name-value pairs from `nvl` which have identical names (if dst has the type `NV_UNIQUE_NAME`), or identical names and types (if dst has the type `NV_UNIQUE_NAME_TYPE`).

The `nvlist_lookup_nv_alloc()` function retrieves the pointer to the allocator used when manipulating a name-value pair list.

PLUGGABLE ALLOCATORS

Using Pluggable Allocators:

The `nv_alloc_init()`, `nv_alloc_reset()` and `nv_alloc_fini()` functions provide an interface that specifies the allocator to be used when manipulating a name-value pair list.

The `nv_alloc_init()` determines allocator properties and puts them into the *nva* argument. You need to specify the *nv_arg* argument, the *nvo* argument and an optional variable argument list. The optional arguments are passed to the (`*nv_ao_init()`) function.

The *nva* argument must be passed to `nvlist_xalloc()`, `nvlist_xpack()`, `nvlist_xunpack()` and `nvlist_xdup()`.

The `nv_alloc_reset()` function resets the allocator properties to the data specified by `nv_alloc_init()`. When no (`*nv_ao_reset()`) function is specified, `nv_alloc_reset()` is without effect.

The `nv_alloc_fini()` destroys the allocator properties determined by `nv_alloc_init()`. When a (`*nv_ao_fini()`) routine is specified, it is called from `nv_alloc_fini()`.

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

The ‘nv_alloc_sleep' and ‘nv_alloc_nosleep' nv_alloc_t pointers may be used with nvlist_xalloc to mimic the behavior of nvlist_alloc with KM_SLEEP and KM_NOSLEEP, respectively.

```
o  nv_alloc_nosleep
o  nv_alloc_sleep
```

The nvpair framework provides a fixed-buffer allocator, accessible via nv_fixed_ops.

```
o  nv_fixed_ops
```

Given a buffer size and address, the fixed-buffer allocator allows for the creation of nvlists in contexts where malloc or kmem_alloc services may not be available. The fixed-buffer allocator is designed primarily to support the creation of nvlists.

Memory freed using `nvlist_free()`, pair-removal, or similar routines is not reclaimed.

When used to initialize the fixed-buffer allocator, nv_alloc_init should be called as follows:

```
int nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo,
    void *bufptr, size_t sz);
```

When invoked on a fixed-buffer, the `nv_alloc_reset()` function resets the fixed buffer and prepares it for re-use. The framework consumer is responsible for freeing the buffer passed to `nv_alloc_init()`.

CREATING PLUGGABLE ALLOCATORS

Any producer of name-value pairs may possibily specify his own allocator routines. You must provide the following pluggable allocator operations in the allocator implementation.

```
int (*nv_ao_init)(nv_alloc_t *nva, va_list nv_valist);
void (*nv_ao_fini)(nv_alloc_t *nva);
void *(*nv_ao_alloc)(nv_alloc_t *nva, size_t sz);
void (*nv_ao_reset)(nv_alloc_t *nva);
void (*nv_ao_free)(nv_alloc_t *nva, void *buf, size_t sz);
```

The *nva* argument of the allocator implementation is always the first argument.

The optional (*`nv_ao_init()`) function is responsible for filling the data specified by `nv_alloc_init()` into the `nva_arg()` argument. The (*`nv_ao_init()`) function is called only when `nv_alloc_init()` is executed.

The optional (*`nv_ao_fini()`) function is responsible for the cleanup of the allocator implementation. It is called by `nv_alloc_fini()`.

The required (*`nv_ao_alloc()`) function is used in the nvpair allocation framework for memory allocation. The sz argument specifies the size of the requested buffer.

The optional (*`nv_ao_reset()`) function is responsible for resetting the nva_arg argument to the data specified by `nv_alloc_init()`.

The required (*`nv_ao_free()`) function is used in the nvpair allocator framework for memory de-allocation. The argument buf is a pointer to a block previously allocated by (*`nv_ao_alloc()`) function. The size argument sz must exactly match the original allocation.

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

Return Values

For `nvlist_alloc()`, `nvlist_dup()`, `nvlist_xalloc()`, and `nvlist_xdup()`:

`0`	success
`EINVAL`	invalid argument
`ENOMEM`	insufficient memory

For `nvlist_pack()`, `nvlist_unpack()`, `nvlist_xpack()`, and `nvlist_xunpack()`:

`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

For `nvlist_lookup_nv_alloc()`:

pointer to the allocator

Usage

The fixed-buffer allocator is very simple allocator. It uses a pre-allocated buffer for memory allocations and it can be used in interrupt context. You are responsible for allocation and de-allocation for the pre-allocated buffer.

Examples

```
/*
 * using the fixed-buffer allocator.
 */
 #include <sys/nvpair.h>

/* initialize the nvpair allocator framework */
static nv_alloc_t *
init(char *buf, size_t size)
```

```
{
     nv_alloc_t *nvap;

     if ((nvap = kmem_alloc(sizeof(nv_alloc_t), KM_SLEEP)) == NULL)
         return (NULL);

     if (nv_alloc_init(nvap, nv_fixed_ops, buf, size) == 0)
         return (nvap);

     return (NULL);
 }

 static void
 fini(nv_alloc_t *nvap)
 {
       nv_alloc_fini(nvap);
       kmem_free(nvap, sizeof(nv_alloc_t));
 }
  static int
  interrupt_context(nv_alloc_t *nva)
  {
     nvlist_t *nvl;
     int error;

     if ((error = nvlist_xalloc(&nvl, NV_UNIQUE_NAME, nva)) != 0)
          return (-1);

     if ((error = nvlist_add_int32(nvl, "name", 1234)) == 0)
          error = send_nvl(nvl);

     nvlist_free(nvl);
     return (error);
    }
```

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.

The `nvlist_xalloc()`, `nvlist_xpack()`, `nvlist_xunpack()`, and `nvlist_xdup()` functions can be called from interrupt context only if (1) the default allocator is used and the `KM_NOSLEEP` flag is set or (2) the specified allocator did not sleep for free memory (for example, it uses a pre-allocated buffer for memory allocations).

These functions can be called from user or kernel context with any valid flag.

Name nvlist_lookup_boolean, nvlist_lookup_boolean_value, nvlist_lookup_byte, nvlist_lookup_int8, nvlist_lookup_int16, nvlist_lookup_int32, nvlist_lookup_int64, nvlist_lookup_uint8, nvlist_lookup_uint16, nvlist_lookup_uint32, nvlist_lookup_uint64, nvlist_lookup_string, nvlist_lookup_nvlist, nvlist_lookup_boolean_array, nvlist_lookup_byte_array, nvlist_lookup_int8_array, nvlist_lookup_int16_array, nvlist_lookup_int32_array, nvlist_lookup_int64_array, nvlist_lookup_uint8_array, nvlist_lookup_uint16_array, nvlist_lookup_uint32_array, nvlist_lookup_uint64_array, nvlist_lookup_string_array, nvlist_lookup_nvlist_array, nvlist_lookup_pairs – 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, const char *name);

int nvlist_lookup_boolean_value(nvlist_t *nvl, const char *name,
     boolean_t *val);

int nvlist_lookup_byte(nvlist_t *nvl, const char *name,
     uchar_t *val);

int nvlist_lookup_int8(nvlist_t *nvl, const char *name,
     int8_t *val);

int nvlist_lookup_uint8(nvlist_t *nvl, const char *name,
     uint8_t *val);

int nvlist_lookup_int16(nvlist_t *nvl, const char *name,
     int16_t *val);

int nvlist_lookup_uint16(nvlist_t *nvl, const char *name,
     uint16_t *val);

int nvlist_lookup_int32(nvlist_t *nvl, const char *name,
     int32_t *val);

int nvlist_lookup_uint32(nvlist_t *nvl, const char *name,
     uint32_t *val);

int nvlist_lookup_int64(nvlist_t *nvl, const char *name,
     int64_t *val);

int nvlist_lookup_uint64(nvlist_t *nvl, const char *name,
     uint64_t *val);

int nvlist_lookup_string(nvlist_t *nvl, const char *name,
     char **val);

int nvlist_lookup_nvlist(nvlist_t *nvl, const char *name,
     nvlist_t **val);

int nvlist_lookup_boolean_array(nvlist_t *nvl, const char *name,
     boolean_t **val,
     uint_t *nelem);
```

```
int nvlist_lookup_byte_array(nvlist_t *nvl, const char *name,
     uchar_t **val,
     uint_t *nelem);

int nvlist_lookup_int8_array(nvlist_t *nvl, const char *name,
     int8_t **val, uint_t *nelem);

int nvlist_lookup_uint8_array(nvlist_t *nvl, const char *name,
     uint8_t **val,
     uint_t *nelem);

int nvlist_lookup_int16_array(nvlist_t *nvl, const char *name,
     int16_t **val,
     uint_t *nelem);

int nvlist_lookup_uint16_array(nvlist_t *nvl, const char *name,
     uint16_t **val,
     uint_t *nelem);

int nvlist_lookup_int32_array(nvlist_t *nvl, const char *name,
     int32_t **val,
     uint_t *nelem);

int nvlist_lookup_uint32_array(nvlist_t *nvl, const char *name,
     uint32_t **val,
     uint_t *nelem);

int nvlist_lookup_int64_array(nvlist_t *nvl, const char *name,
     int64_t **val,
     uint_t *nelem);

int nvlist_lookup_uint64_array(nvlist_t *nvl, const char *name,
     uint64_t **val,
     uint_t *nelem);

int nvlist_lookup_string_array(nvlist_t *nvl, const char *name,
     char  ***val,
     uint_t *nelem);

int nvlist_lookup_nvlist_array(nvlist_t *nvl, const char *name,
     nvlist_t ***val,
     uint_t *nelem);

int nvlist_lookup_pairs(nvlist_t *nvl, int flag, ...);
```

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.
nelem	Address to store the number of elements in value.
val	Address to store the value or starting address of the array value.

flag	Specify bit fields defining lookup behavior:

NV_FLAG_NOENTOK	The retrival function will not fail if no matching name-value pair is found.

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.

Multiple threads can simultaneously read the same `nvlist_t` but only one thread should actively change a given `nvlist_t` at a time. The caller is responsible for the synchronization.

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*.

The `nvlist_lookup_pairs()` function retrieves a set of `nvpairs`. The arguments are a null-terminated list of pairs (data type `DATA_TYPE_BOOLEAN`), triples (non-array data types) or quads (array data types). As shown below, the interpretation of the arguments depends on the value of *type*. See `nvpair_type`(9F).

name	Name of the name-value pair to search.
type	Data type.
val	Address to store the starting address of the value. When using data type `DATA_TYPE_BOOLEAN`, the *val* argument is ignored.
nelem	Address to store the number of elements in value. Non-array data types have only one argument and *nelem* is ignored.

The argument order is *name*, *type*, [*val*], [*nelem*].

When using `NV_FLAG_NOENTOK` and no matching name-value pair is found, the memory pointed to by *val* and *nelem* is not touched.

These functions return `0` on success and an error value on failure.

Errors These functions fail under the following conditions.

`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, interrupt, or kernel context.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `nvlist_alloc`(9F), `nvpair_type`(9F)

Writing Device Drivers

STREAMS Programming Guide

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:

```
DATA_TYPE_BOOLEAN
DATA_TYPE_BOOLEAN_VALUE
DATA_TYPE_BYTE
DATA_TYPE_INT8
DATA_TYPE_UINT8
DATA_TYPE_INT16
DATA_TYPE_UINT16
DATA_TYPE_INT32
DATA_TYPE_UINT32
DATA_TYPE_INT64
DATA_TYPE_UINT64
DATA_TYPE_STRING
DATA_TYPE_NVLIST
DATA_TYPE_BOOLEAN_ARRAY
DATA_TYPE_BYTE_ARRAY
DATA_TYPE_INT8_ARRAY
DATA_TYPE_UINT8_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
DATA_TYPE_NVLIST_ARRAY
```

After nvpairs is removed from or replaced in an nvlist, it cannot be manipulated. This includes `nvlist_next_nvpair()`, `nvpair_name()` and `nvpair_type()`. Replacement can happen during pair addition on nvlists created with NV_UNIQUE_NAME_TYPE and NV_UNIQUE_NAME. See `nvlist_alloc`(9F) for more details.

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

Name nvlist_remove, nvlist_remove_all – remove name-value pairs

Synopsis

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

int nvlist_remove(nvlist_t *nvl, const char *name, data_type_t type);

int  nvlist_remove_all(nvlist_t *nvl, const char *name);
```

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 be removed.
type	Data type of the `nvpair` to be removed.

Description 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.

Multiple threads can simultaneously read the same `nvlist_t` but only one thread may actively change a given `nvlist_t` at a time. The caller is responsible for the synchronization.

Return Values These functions return `0` on success and an error value on failure.

Context The `nvlist_remove()` and `nvlist_remove_all()` functions can be called from user, interrupt, or kernel context.

Errors

EINVAL	There is an invalid argument.
ENOENT	No name-value pairs were found to match the criteria specified by name and type.

Name nvpair_value_byte, nvpair_value_nvlist, nvpair_value_int8, nvpair_value_int16, nvpair_value_int32, nvpair_value_int64, nvpair_value_uint8, nvpair_value_uint16, nvpair_value_uint32, nvpair_value_uint64, nvpair_value_string, nvpair_value_boolean_array, nvpair_value_byte_array, nvpair_value_nvlist_array, nvpair_value_int8_array, nvpair_value_int16_array, nvpair_value_int32_array, nvpair_value_int64_array, nvpair_value_uint8_array, nvpair_value_uint16_array, nvpair_value_uint32_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair

Synopsis

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

int nvpair_value_boolean_value(nvpair_t *nvpair, boolean_t *val);

int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);

int nvpair_value_int8(nvpair_t *nvpair, int8_t *val);

int nvpair_value_uint8(nvpair_t *nvpair, uint8_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_nvlist(nvpair_t *nvpair, nvlist_t **val);

int nvpair_value_boolean_array(nvpair_t *nvpair, boolean_t **val,
     uint_t *nelem);

int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val,
     uint_t *nelem);

int nvpair_value_int8_array(nvpair_t *nvpair, int8_t **val,
     uint_t *nelem);

int nvpair_value_uint8_array(nvpair_t *nvpair, uint8_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);

int nvpair_value_nvlist_array(nvpair_t *nvpair, nvlist_t ***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.

Description 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)`

The value of an nvpair may not be retrieved after the nvpair having been removed from or replaced in an nvlist. Replacement can happen during pair addition on nvlists created with NV_UNIQUE_NAME_TYPE and NV_UNIQUE_NAME. See `nvlist_alloc(9F)` for more details.

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, interrupt, or kernel context.

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 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

The `OTHERQ()` function returns a pointer to a queue's partner.

Context

The `OTHERQ()` function can be called from user, interrupt, or kernel 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.

```
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

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_put8(9F)`, and `ddi_put8(9F)` instead. For `repoutsb()`, `repoutsw()`, and`repoutsl()`, use, respectively, `ddi_rep_put8(9F)`, `ddi_rep_put8(9F)`, and `ddi_rep_put8(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 to the I/O port.

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, interrupt, or kernel context.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	x86
Stability Level	Obsolete

See Also `isa`(4), `attributes`(5), `ddi_put8`(9F), `ddi_put8`(9F), `ddi_put8`(9F), `ddi_rep_put8`(9F), `ddi_rep_put8`(9F), `ddi_rep_put8`(9F), `inb`(9F)

Writing Device Drivers

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.

Context These routines can be called from user, kernel, or interrupt context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus

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:

Previous Name	New Name
`pci_config_getb`	`pci_config_get8`
`pci_config_getw`	`pci_config_get16`
`pci_config_getl`	`pci_config_get32`
`pci_config_getll`	`pci_config_get64`
`pci_config_putb`	`pci_config_put8`
`pci_config_putw`	`pci_config_put16`
`pci_config_putl`	`pci_config_put32`
`pci_config_putll`	`pci_config_put64`

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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus

See Also `attributes(5)`

IEEE 1275 PCI Bus Binding

Name

pci_ereport_setup, pci_ereport_teardown, pci_ereport_post – post error reports for the generic PCI errors logged in the PCI Configuration Status register.

Synopsis

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

void pci_ereport_setup(dev_info_t *dip, int);

void pci_ereport_teardown(dev_info_t *dip);

void pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *dep,
     uin16_t *status);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the `dev_info` structure of the devices
dep	Pointer to DDI error status
status	Pointer to status bit storage location

Description

The `pci_ereport_setup()` function initializes support for error report generation and sets up the resources for subsequent access to PCI, `PCI/X` or `PCI Express Configuration` space. The caller must have established a fault management capability level of at least `DDI_FM_EREPORT_CAPABLE` with a previous call to `ddi_fm_init()` for *dip*.

The `pci_ereport_teardown()` function releases any resources allocated and set up by `pci_ereport_setup()` and associated with *dip*.

The `pci_ereport_post()` function is called to scan for and post any PCI, `PCI/X` or `PCI Express Bus` errors. On a PCI bus, for example, the errors detected include:

- Detected Parity Error
- Master Data Parity Error
- Target Abort
- Master Abort
- System Error
- Discard Timeout

The `pci_ereport_post()` function must be called only from a driver's error handler callback function. See `ddi_fm_handler_register(9F)`. The *error_status* argument to the error handler callback function should be passed through as the *dep* argument to `pci_ereport_post()` as it may contain bus specific information that might be useful for handling any errors that are discovered.

The `fme_flag` in the `error_status` argument to the error handler callback function will contain one of the following:

`DDI_FM_ERR_UNEXPECTED()`	Any errors discovered are unexpected.

`DDI_FM_ERR_EXPECTED()`	Errors discovered were the result of a `DDI_ACC_CAUTIOUS` operation.
`DDI_FM_ERR_POKE()`	Errors discovered are the result of a `ddi_poke`(9F) operation.
`DDI_FM_ERR_PEEK()`	Errors discovered are the result of a `ddi_peek`(9F) operation.

Error report events are generated automatically if `fme_flag` is set to `DDI_FM_ERR_UNEXPECTED` and the corresponding error bits are set in the various PCI, `PCI/X` or `PCI Express Bus` error registers of the device associated with *dip*. The generated error report events are posted to the Solaris Fault Manager, `fmd`(1M), for diagnosis.

If the status argument is non-null, `pci_ereport_post()` saves the contents of the `PCI Configuration Status Register` to `*status`. If it is not possible to read the `PCI Configuration Status Register`, `-1` is returned in `*status` instead.

On return from the call to `pci_ereport_post()`, the `ddi_fm_error_t` structure pointed at by *dep* will have been updated, and the `fme_status` field contains one of the following values:

`DDI_FM_OK`	No errors were detected which might affect this device instance.
`DDI_FM_FATAL`	An error which is considered fatal to the operational state of the system was detected.
`DDI_FM_NONFATAL`	An error which is not considered fatal to the operational state of the system was detected. The `fme_acc_handle` or `fme_dma_handle` fields in the returned `ddi_fm_error_t` structure will typically reference a handle that belongs to the device instance that has been affected.
`DDI_FM_UNKNOWN`	An error was detected, but the call was unable to determine the impact of the error on the operational state of the system. This is treated the same way as `DDI_FM_FATAL` unless some other device is able to evaluate the fault to be `DDI_FM_NONFATAL`.

Context

The `pci_ereport_setup()` and `pci_ereport_teardown()` functions must be called from user or kernel context.

The `pci_ereport_post()` function can be called in any context.

Examples

```
int xxx_fmcap = DDI_FM_EREPORT_CAPABLE | DDI_FM_ERRCB_CAPABLE;

xxx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd) {

      ddi_fm_init(dip, &xxx_fmcap, &xxx_ibc);
          if (xxx_fmcap & DDI_FM_ERRCB_CAPABLE)
      ddi_fm_handler_register(dip, xxx_err_cb);
          if (xxx_fmcap & DDI_FM_EREPORT_CAPABLE)
              pci_ereport_setup(dip);
```

```
}

xxx_err_cb(dev_info_t *dip, ddi_fm_error_t *errp) {
uint16_t status;

     pci_ereport_post(dip, errp, &status);
return (errp->fme_status);
}

xxx_detach(dev_info_t *dip, ddi_attach_cmd_t cmd) {

     if (xxx_fmcap & DDI_FM_EREPORT_CAPABLE)
         pci_ereport_teardown(dip);
     if (xxx_fmcap & DDI_FM_ERRCB_CAPABLE)
         ddi_fm_handler_unregister(dip);
ddi_fm_fini(dip);

}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`fmd(1M)`, `attributes(5)`, `ddi_fm_handler_register(9F)`, `ddi_fm_init(9F)`, `ddi_peek(9F)`, `ddi_poke(9F)`, `ddi_fm_error(9S)`

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

Solaris DDI specific (Solaris DDI)

Parameters

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)`:

```
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",
                    ddi_driver_name(dip), ddi_get_instance(dip));
```

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:

```
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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

Writing Device Drivers

PCI Bus Power Management Interface Specification Version 1.1

PCI Bus Specification Revision 2.1

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	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);
```

EXAMPLE 1 Invoking the save and restore functions *(Continued)*

```
        if (pci_restore_config_regs(dip) == DDI_FAILURE) {
            /*
             * appropriate error path handling here
             */
            ...
            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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `attributes(5)`, `power(9E)`

Writing Device Drivers

PCI Bus Power Management Interface Specification Version 1.1

PCI Bus Specification Revision 2.1

Name physio, minphys – perform physical I/O

Synopsis

```
#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 Solaris DDI specific (Solaris DDI).

Parameters

`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.
`non-zero`	Upon failure.

Context `physio()` can be called from user context only.

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.

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)

Parameters

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.

If these functions are called as a result of entry into the driver's `attach`(9E), `detach`(9E) or `power`(9E) entry point, these functions must be called from the same thread which entered `attach`(9E), `detach`(9E) or `power`(9E).

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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also `power.conf`(4), `pm`(7D), `attach`(9E), `detach`(9E), `power`(9E), `pm_raise_power`(9F), `pm`(9P), `pm-components`(9P)

Writing Device Drivers

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 Solaris DDI specific (Solaris DDI)

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 inquirying 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()`.

Note – If this function is called as a result of entry into the driver's `attach`(9E), `detach`(9E) or `power`(9E) entry point, this function must be called from the same thread which entered `attach`(9E), `detach`(9E) or `power`(9E).

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)
{

    ...


    /*
     * 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.
     */
        ASSERT(mutex_owned(&xsp->lock));
    if (level_informed == *(xsp->level_reg_addr)) {
        if (pm_power_has_changed(xsp->dip, XXUSB_COMPONENT,
            level_informed) != DDI_SUCCESS) {
            mutex_exit( &xsp->lock);
            return(DDI_INTR_UNCLAIMED);
        }
        }

    ....

}


xxdisk_power(dev_info *dip, int comp, int level)
{
    mutex_enter( xsp->lock);

    ...

    ...

}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability level	Evolving

See Also `power.conf`(4), `pm`(7D), `attach`(9E), `detach`(9E), `power`(9E), `pm_busy_component`(9F), `pm_idle_component`(9F), `pm_raise_power`(9F), `pm_lower_power`(9F), `pm`(9P), `pm-components`(9P)

Writing Device Drivers

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)

Parameters

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).

Note – If these functions are called as a result of entry into the driver's `attach`(9E), `detach`(9E) or `power`(9E) entry point, these functions must be called from the same thread which entered `attach`(9E), `detach`(9E) or `power`(9E).

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):

```
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_busycnt < 1) {
        /*
            * Component is not already marked busy
            */
          if (pm_busy_component(xsp->dip,
                 XXDISK_COMPONENT) != DDI_SUCCESS) {
                    bioerror(bp,EIO);
                    biodone(bp);
                    return (0);
                    }
                    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) {
                if (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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also `power.conf`(4), `pm`(7D), `attach`(9E), `detach`(9E), `power`(9E), `pm_busy_component`(9F), `pm_idle_component`(9F), `pm`(9P), `pm-components`(9P)

Writing Device Drivers

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
`0`	Power cycle is not advised
`-1`	Error due to invalid argument.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

`power.conf`(4), `attributes`(5), `power`(9E)

Writing Device Drivers

Using Power Management

Name pollwakeup – inform a process that an event has occurred

Synopsis

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



void pollwakeup(struct pollhead *php, short event);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

php	Pointer to a `pollhead` structure.
event	Event to notify the process about.

Description The `pollwakeup()` function 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 The `pollwakeup()` function can be called from user, interrupt, or kernel context.

See Also `poll(2)`, `chpoll(9E)`

Writing Device Drivers

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

Name priv_getbyname – map a privilege name to a number

Synopsis

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



int priv_getbyname(const char  *priv, int flags);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

priv	name of the privilege
flags	flags, must be zero or `PRIV_ALLOC`

Description The `priv_getbyname()` function maps a privilege name to a privilege number for use with the `priv_*()` kernel interfaces.

If `PRIV_ALLOC` is passed as a flag parameter, an attempt is made to allocate a privilege if it is not yet defined. The newly allocated privilege number is returned.

Privilege names can be specified with an optional `priv_` prefix, which is stripped.

Privilege names are case insensitive but allocated privileges preserve case.

Allocated privileges can be at most `{PRIVNAME_MAX}` characters long and can contain only alphanumeric characters and the underscore character.

Return Values This function returns the privilege number, which is greater than or equal to 0, if it succeeds. It returns a negative error number if an error occurs.

Errors

`EINVAL` This might be caused by any of the following

- The *flags* parameter is invalid.
- The specified privilege does not exist.
- The *priv* parameter contains invalid characters.

`ENOMEM` There is no room to allocate another privilege.

`ENAMETOOLONG` An attempt was made to allocate a privilege that was longer than `{PRIVNAME_MAX}` characters.

Context This functions can be called from user and kernel contexts.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Interface Stability	Evolving

See Also `attributes`(5), `privileges`(5)

Writing Device Drivers

Name priv_policy, priv_policy_only, priv_policy_choice – check, report, and audit privileges

Synopsis

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



int priv_policy(const cred_t  *cr, int priv, int err,
     const char  *msg);

int priv_policy_only(const cred_t  *cr, int priv);

int priv_policy_choice(const cred_t  *cr, int priv);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

cr	The credential to be checked.
priv	The integer value of the privilege to test.
err	The error code to return.
msg	String that is added to the privilege debugging message if one is generated. NULL if no additional information is needed. Because the function name is included in the output, NULL is usually the best value to pass as a parameter.

Description These functions aid in privilege checking and privilege debugging.

The `priv_policy()`, `priv_policy_only()`, and `priv_policy_choice()` functions all check whether *priv* is asserted in the effective set of the credential. The special value `PRIV_ALL` tests for all privileges.

The `priv_policy()` function updates the ASU accounting flag and records the privilege used on success in the audit trail if the required privilege was not a basic privilege.

The `priv_policy_only()` function checks whether a privilege is asserted and has no side effects.

The `priv_policy_choice()` function behaves like `priv_policy_only()` but records the successfully used non-basic privileges in the audit trail.

Return Values On success, `priv_policy()` return 0. On failure it returns its parameter *err*.

On success, `priv_policy_choice()` and `priv_policy_only()` return 1, on failure both return 0.

Errors

`EINVAL`	This might be caused by any of the following: ■ The *flags* parameter is invalid. ■ The specified privilege does not exist. ■ The *priv* parameter contains invalid characters.
`ENOMEM`	There is no room to allocate another privilege.

`ENAMETOOLONG`	An attempt was made to allocate a privilege that was longer than `{PRIVNAME_MAX}` characters.

Context This functions can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also `acct(3HEAD)`, `attributes(5)`, `privileges(5)`

Writing Device Drivers

Name

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)
```

```
voidproc_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.h`(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

The `proc_ref()` returns the following:

pref	An opaque handle used to refer to the current process.

The `proc_signal()` returns the following:

`0`	The process existed before the signal was sent.
`-1`	The process no longer exists; no signal was sent.

Context The `proc_unref()` and `proc_signal()` functions can be called from user, interrupt, or kernel context. The `proc_ref()` function should be called only from user context.

See Also `signal.h`(3HEAD), `putnextctl1`(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 The `ptob()` function can be called from user, interrupt, or kernel context.

See Also `btop`(9F), `btopr`(9F), `ddi_ptob`(9F)

Writing Device Drivers

Name

pullupmsg – concatenate bytes in a message

Synopsis

```
#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

The `pullupmsg()` function 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

The `pullupmsg()` function can be called from user, interrupt, or kernel context.

Examples

EXAMPLE 1 Using `pullupmsg()`

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.

```
 1 xxxwsrv(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
 9        while ((mp = getq(q)) != NULL) {
10                switch (mp->b_datap->db_type) {
11             case M_DATA:
12                       if (pullupmsg(mp, -1)) {
13                            dma_addr = vtop(mp->b_rptr);
14                               dma_len = mp->b_wptr - mp->b_rptr;
15                               xxx_do_dma(dma_addr, dma_len);
```

EXAMPLE 1 Using `pullupmsg()` *(Continued)*

```
                                freemsg(mp);
                                break;
                        }
                        for (tmp = mp; tmp; tmp = tmp->b_cont) {
                             dma_addr = vtop(tmp->b_rptr);
                             dma_len = tmp->b_wptr - tmp->b_rptr;
                             xxx_do_dma(dma_addr, dma_len);
                     }
                        freemsg(mp);
                        break;
          . . .
                }
        }
}
```

See Also

`srv`(9E), `allocb`(9F), `msgpullup`(9F), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

Notes

The `pullupmsg()` function 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 Architecture independent level 1 (DDI/DKI).

Parameters

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)`, `putctl1(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

```
#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 The `putbq()` function 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 The `putbq()` function 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 The `putbq()` function can be called from user, interrupt, or kernel 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

Name putctl1 – send a control message with a one-byte parameter to a queue

Synopsis

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



int putctl1(queue_t *q, int type, int p);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

q	Queue to which the message is to be sent.
type	Type of message.
p	One-byte parameter.

Description The `putctl1()` function, 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.

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 The `putctl1()` function can be called from user, interrupt, or kernel context.

Examples See the `putctl(9F)` function page for an example of `putctl1()`.

See Also `put(9E)`, `allocb(9F)`, `datamsg(9F)`, `putctl(9F)`, `putnextctl1(9F)`

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 Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Queue to which the message is to be sent.
type	Message type (must be control, not data type).

Description

The `putctl()` function 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

The `putctl()` function can be called from user, interrupt, or kernel 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). `putctl1`(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).

```
 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
 9       switch (type) {
10          case M_BREAK:
11             if (!putctl(wrq->q_next, M_BREAK))
12                      num_alloc_fail++;
13                break;
14
15       case M_DELAY:
16             if (!putctl1(wrq->q_next, M_DELAY, parm))
17                      num_alloc_fail++;
```

EXAMPLE 1 Using `putctl()` *(Continued)*

```
           break;

     default:
           cmn_err(CE_PANIC, "send_ctl: bad message type passed");
              break;
        }
}
```

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

```
#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 The `putnext()` function is used to pass a message to the `put`(9E) routine of the next queue in the stream.

Return Values None.

Context The `putnext()` function can be called from user, interrupt, or kernel context.

Examples See `allocb`(9F) for an example of using `putnext()`.

See Also `put`(9E), `allocb`(9F), `put`(9F), `qprocson`(9F)

Writing Device Drivers

STREAMS Programming Guide

Notes The `put()` and `putnext()` functions should be called only after `qprocson()` is finished.

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 Level Architecture independent level 1 (DDI/DKI).

Parameters

q	Queue to which the message is to be sent.
type	Type of message.
p	One-byte parameter.

Description The `putnextctl1()` function, 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.

The `putnextctl1()` function 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 The `putnextctl1()` function can be called from user, interrupt, or kernel context.

Examples See the `putnextctl(9F)` function page for an example of `putnextctl1()`.

See Also `put(9E)`, `allocb(9F)`, `datamsg(9F)`, `putctl1(9F)`, `putnextctl(9F)`

Writing Device Drivers

STREAMS Programming Guide

Name

putnextctl – send a control message to a queue

Synopsis

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



int putnextctl(queue_t *q, int type);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Queue to which the message is to be sent.
type	Message type (must be control, not data type).

Description

The `putnextctl()` function 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.

The `putnextctl()` function should always be used in preference to `putctl`(9F)

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

The `putnextctl()` function can be user, interrupt, or kernel context.

Examples

The `send_ctl` routine is used to pass control messages downstream. `M_BREAK` messages are handled with `putnextctl( )` (line 8). `putnextctl1`(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).

```
 1  void
 2  send_ctl(queue_t *wrq, uchar_t type, uchar_t parm)
 3  {
 4                extern int num_alloc_fail;
 5
 6                switch (type) {
 7             case M_BREAK:
 8                    if (!putnextctl(wrq, M_BREAK))
 9                                num_alloc_fail++;
10                       break;
11
```

```
              case M_DELAY:
                  if (!putnextctl1(wrq, M_DELAY, parm))
                              num_alloc_fail++;
                  break;

           default:
                  cmn_err(CE_PANIC, "send_ctl: bad message type passed");
                     break;
              }
}
```

See Also put(9E), cmn_err(9F), datamsg(9F), putctl(9F), putnextctl1(9F)

Writing Device Drivers

STREAMS Programming Guide

Name putq – put a message on a queue

Synopsis

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



int putq(queue_t *q, mblk_t *bp);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

q	Pointer to the queue to which the message is to be added.
bp	Message to be put on the queue.

Description The `putq()` function 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.

Return Values The `putq()` function returns `1` on success and `0` on failure.

Note – Upon failure, the caller should call `freemsg`(9F) to free the pointer to the message block.

Context The `putq()` function can be called from user, interrupt, or kernel context.

Examples See the `datamsg`(9F) function page for an example of `putq()`.

See Also `datamsg`(9F), `putbq`(9F), `qenable`(9F), `rmvq`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name qassociate – associate STREAMS queue with driver instance

Synopsis

```
#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:

```
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:

```
(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)

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 Level Solaris DDI specific (Solaris DDI).

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 The `qbufcall()` function 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, the `qbufcall()` function 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 The `qbufcall()` function can be called from user, interrupt, or kernel 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 Architecture independent level 1 (DDI/DKI).

Parameters *q* Pointer to the queue to be enabled.

Description The `qenable()` function 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 The `qenable()` function can be called from user, interrupt, or kernel context.

Examples See the `dupb`(9F) function page for an example of the `qenable()`.

See Also `dupb`(9F)

Writing Device Drivers

STREAMS Programming Guide

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

q	Pointer to the RD side of a STREAMS queue pair.

Description The `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.

The `qprocson()` function 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.

The `qprocson()` function must be called before calling `put`(9F), `putnext`(9F), `qbufcall`(9F), `qtimeout`(9F), `qwait`(9F), or `qwait_sig`(9F).

The `qprocsoff()` function 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, interrupt, or kernel 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 Level

Architecture independent level 1 (DDI/DKI).

Parameters

q	Pointer to the queue.
mp	Pointer to the message to be sent in the opposite direction.

Description

The `qreply()` function sends messages in the reverse direction of normal flow. That is, `qreply(`*q*`,` *mp*`)` is equivalent to `putnext(OTHERQ(`*q*`),` *mp*`)`.

Context

The `qreply()` function can be called from user, interrupt, or kernel 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()` 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     }
```

EXAMPLE 1 Canonical Flushing Code for STREAMS Drivers. *(Continued)*

```
}
```

See Also

`put`(9E), `srv`(9E), `flushq`(9F), `OTHERQ`(9F), `putnext`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name qsize – find the number of messages on a queue

Synopsis

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



int qsize(queue_t *q);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

q	Queue to be evaluated.

Description The `qsize()` function evaluates the queue *q* and returns the number of messages it contains.

Return Values If there are no message on the queue, `qsize()` returns `0`. Otherwise, it returns the integer representing the number of messages on the queue.

Context The `qsize()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

STREAMS Programming Guide

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 *),
     void *arg, clock_t ticks);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

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).

The `qprocson`(9F) function must be called before calling `qtimeout()`.

Return Values The `qtimeout()` function 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 The `qtimeout()` function can be called from user, interrupt, or kernel context.

See Also `drv_usectohz`(9F), `mt-streams`(9F), `qbufcall`(9F), `qprocson`(9F), `qunbufcall`(9F), `quntimeout`(9F)

Writing Device Drivers

STREAMS Programming Guide

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 The `qunbufcall()` function 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 The `qunbufcall()` function can be called from user, interrupt, or kernel context.

See Also `mt-streams`(9F), `qbufcall`(9F), `qtimeout`(9F), `quntimeout`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name quntimeout – cancel previous qtimeout function call

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 The `quntimeout()` function 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 The `quntimeout()` function returns `-1` if the `id` is not found. Otherwise, `quntimeout()` returns a `0` or positive value.

Context The `quntimeout()` function can be called from user, interrupt, or kernel context.

See Also `mt-streams(9F)`, `qbufcall(9F)`, `qtimeout(9F)`, `qunbufcall(9F)`

Writing Device Drivers

STREAMS Programming Guide

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 Level Solaris DDI specific (Solaris DDI).

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.

Note – The thread that calls `close()` does not necessarily have the ability to receive signals, particularly when called by `exit`(2). In this case, `qwait_sig()` behaves exactly as `qwait()`. Driver writers may use `ddi_can_receive_sig`(9F) to determine when this is the case, and, if so, arrange some means to avoid blocking indefinitely (for example, by using `qtimeout`(9F).

`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`.

```
xxopen(qp, . . .)
       queue_t *qp;
{
          struct xxdata *xx;
           /* Allocate xxdata structure */
       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), `ddi_can_receive_sig`(9F), `mt-streams`(9F), `qbufcall`(9F), `qprocson`(9F), `qtimeout`(9F)

STREAMS Programming Guide

Writing Device Drivers

Name qwriter – asynchronous STREAMS perimeter upgrade

Synopsis

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



void qwriter(queue_t *qp, mblk_t *mp, void (*func)(), int perimeter);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

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:

```
(*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 an `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

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 The `RD()` function can be called from user, interrupt, or kernel 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	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 The `rmalloc()` function 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.

The `rmalloc()` function 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 The `rmalloc()` function can be called from user, interrupt, or kernel 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).

EXAMPLE 1 Illustrating the principles of map management *(Continued)*

- 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 */
    . . .
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
13          cmn_err(CE_PANIC, "xxstart: kmem_alloc failed before %d buffer"
14                  "allocation", XX_BUFSIZE);
15      }
16
17      /*
18       * Initialize the resource map with number
19       * of slots in map.
20       */
21      xx_mp = rmallocmap(XX_MAPSIZE);
22
24      /*
25       * Initialize space management map with total
26       * buffer area it is to manage.
27       */
28      rmfree(xx_mp, XX_BUFSIZE, bp);
        . . .
```

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).

EXAMPLE 2 Allocating buffers *(Continued)*

- 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.

```
1   #define XX_BUFSIZE  2560
2   #define XX_MAXSIZE  (XX_BUFSIZE / 4)
3
4   static struct map *xx_mp;         /* Private buffer space map */
    ...
5   xxread(dev_t dev, uio_t *uiop, cred_t *credp)
6   {
7
8   register caddr_t addr;
9   register int     size;
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

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

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

mp	Pointer to the resource map from which space is to be allocated.
size	Number of units of space to allocate.

Description

The `rmalloc_wait()` function 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

The `rmalloc_wait()` function returns the base of the allocated space.

Context

This function can be called from user, interrupt, or kernel context. However, in most cases `rmalloc_wait()` should not be called from interrupt context.

See Also

`rmalloc`(9F), `rmallocmap`(9F), `rmfree`(9F), `rmfreemap`(9F)

Writing Device Drivers

Name rmfree – free space back into a resource map

Synopsis

```
#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 The `rmfree()` function 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.

The `rmfree()` function can also be used to initialize a resource map, in which case the size and index should cover the entire resource area.

Context The `rmfree()` function can be called from user, interrupt, or kernel context.

See Also `rmalloc(9F)`, `rmalloc_wait(9F)`, `rmallocmap(9F)`, `rmfreemap(9F)`

Writing Device Drivers

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

Architecture independent level 1 (DDI/DKI).

Parameters

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

The `rmvb()` function 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

The `rmvb()` function can be called from user, interrupt, or kernel 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) &&
                         (tmp->b_rptr == tmp->b_wptr)) {
12                       (void) rmvb(mp, tmp);
13                       freeb(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

The `rmvq()` function 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

The `rmvq()` function can be called from user, interrupt, or kernel 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).

```
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

STREAMS Programming Guide

Warnings Make sure that the message *mp* is linked onto *q* to avoid a possible system panic.

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

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.

The `rw_init()` function 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 interrupt priority returned from `ddi_intr_get_pri`(9F) or `ddi_intr_get_softint_pri`(9F). Note that arg should be the value of the interrupt priority cast by calling the `DDI_INTR_PRI` macro. If the lock is not used by any interrupt handler, the argument should be `NULL.`

The `rw_destroy()` function 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.

The `rw_enter()` function 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.

The `rw_exit()` function releases the lock and may wake up one or more threads waiting on the lock.

The `rw_tryenter()` function 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.

The `rw_tryupgrade()` function 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.

The `rw_read_locked()` function 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, interrupt, or kernel context, except for `rw_init()` and `rw_destroy()`, which can be called from user context only.

See Also

`condvar`(9F), `ddi_intr_alloc`(9F), `ddi_intr_add_handler`(9F), `ddi_intr_get_pri`(9F), `ddi_intr_get_softint_pri`(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).

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 The `SAMESTR()` function 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 The `SAMESTR()` function can be called from user, 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>



intscsi_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 The `scsi_abort()` function 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 The `scsi_abort()` function returns:

`1`	on success.
`0`	on failure.

Context The `scsi_abort()` function can be called from user, interrupt, or kernel 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

Name	scsi_alloc_consistent_buf – allocate an I/O buffer for SCSI DMA

Synopsis

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



struct buf *scsi_alloc_consistent_buf(structscsi_address*ap,
     struct buf *bp, size_t datalen, uint_t bflags,
     int (*callback)(caddr_t), caddr_t arg);
```

Interface Level Solaris DDI specific (Solaris DDI).

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 The `scsi_alloc_consistent_buf()` function returns 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.

Return Values The `scsi_alloc_consistent_buf()` function 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 fromuser, interrupt, or kernel context. 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);
 rqpkt = 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 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 The `scsi_cname()` function 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.

The `scsi_dname()` function decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

The `scsi_mname()` function decodes SCSI messages.

The `scsi_rname()` function decodes packet completion reasons.

The `scsi_sname()` function 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, interrupt, or kernel context.

Examples **EXAMPLE 1** Decoding SCSI tape commands.

The `scsi_cname()` function decodes SCSI tape commands as follows:

EXAMPLE 1 Decoding SCSI tape commands. *(Continued)*

```
static char *st_cmds[] = {
            "\000test unit ready",
            "\001rewind",
            "\003request sense",
            "\010read",
            "\012write",
            "\020write file mark",
            "\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_destroy_pkt – free an allocated SCSI packet and its DMA resource

Synopsis

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



void scsi_destroy_pkt(struct scsi_pkt *pktp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

pktp	Pointer to a `scsi_pkt`(9S) structure.

Description The `scsi_destroy_pkt()` function 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 *pktp* is freed.

Context The `scsi_destroy_pkt()` function may be called from user, interrupt, or kernel 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_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

The `scsi_dmaget()` function allocates DMA resources for an already allocated SCSI packet. *pkt* is a pointer to the previously allocated SCSI packet (see `scsi_pktalloc`(9F)).

The *dmatoken* parameter 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.

The *callback* parameter 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.

The `scsi_dmafree()` function frees the DMA resources associated with the SCSI packet. The packet itself remains allocated.

Return Values

The `scsi_dmaget()` function 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 or kernel context. Otherwise, it may be called from user, kernel, or interrupt context. The *callback* function may not block or call routines that block.

The `scsi_dmafree()` function can be called from user, interrupt, or kernel context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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_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 *cm
     struct scsi_extended_sense *sensep);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

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 The `scsi_errmsg()` function 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:

Severity Value:	String:
`SCSI_ERR_ALL`	`All`
`SCSI_ERR_UNKNOWN`	`Unknown`
`SCSI_ERR_INFO`	`Informational`
`SCSI_ERR_RECOVERE`	`Recovered`
`SCSI_ERR_RETRYABL`	`Retryable`

Severity Value:	String:
SCSI_ERR_FATAL	Fatal

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:

```
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
};
```

Context

The `scsi_errmsg()` function may be called from user, interrupt, or kernel context.

Examples

EXAMPLE 1 Generating error information.

This entry:

```
scsi_errmsg(devp, pkt, "sd", SCSI_ERR_INFO, bp->b_blkno,
      err_blkno, sd_cmds, rqsense);
```

EXAMPLE 1 Generating error information. *(Continued)*

Generates:

```
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: QUANTUM    Serial Number: 123456
    Sense Key:  Unit Attention
    ASC: 0x29 (reset), ASCQ: 0x0, FRU: 0x0
```

See Also `cmn_err`(9F), `scsi_log`(9F), `scsi_device`(9S), `scsi_extended_sense`(9S), `scsi_pkt`(9S)

Writing Device Drivers

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 The `scsi_free_consistent_buf()` function frees a buffer header and consistent data buffer that was previously allocated using `scsi_alloc_consistent_buf`(9F).

Context The `scsi_free_consistent_buf()` function can be called from user, interrupt, or kernel context.

See Also `freerbuf`(9F), `scsi_alloc_consistent_buf`(9F), `buf`(9S)

Writing Device Drivers

Warning The `scsi_free_consistent_buf()` function 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.

Name scsi_get_device_type_scsi_options – look up per-device-type scsi-options property

Synopsis

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



int scsi_get_device_type_scsi_options(dev_info_t  *dip, struct scsi_device  *devp,
     int default_scsi_options);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

dip	Pointer to the device info node for this HBA driver.
devp	Pointer to a `scsi_device(9S)` structure of the target.
default_scsi_options	Value returned if no match is found.

Description 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 DFHSS4S",  "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*

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_PARITY`
If not set, the HBA should not operate in parity mode.

`SCSI_OPTIONS_QAS`
If not set, the HBA should not make use of the Quick Arbitration Select feature. Consult your Sun hardware documentation to determine whether your machine supports QAS.

`SCSI_OPTIONS_FAST`
If not set, the HBA should not operate the bus in FAST SCSImode.

`SCSI_OPTIONS_FAST20`
If not set, the HBA should not operate the bus in FAST20 SCSI mode.

`SCSI_OPTIONS_FAST40`
If not set, the HBA should not operate the bus in FAST40 SCSI mode.

`SCSI_OPTIONS_FAST80`
If not set, the HBA should not operate the bus in FAST80 SCSI mode.

`SCSI_OPTIONS_FAST160`
If not set, the HBA should not operate the bus in FAST160 SCSI mode.

`SCSI_OPTIONS_FAST320`
If not set, the HBA should not operate the bus in FAST320 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.

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.

Description

`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.

Name scsi_hba_lookup_capstr – return index matching capability string

Synopsis

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

int scsi_hba_lookup_capstr(char *capstr);
```

Interface Level Solaris architecture specific (Solaris DDI).

Parameters

capstr	Pointer to a string

Description The `scsi_hba_lookup_capstr()` function attempts to match *capstr* against a known set of capability strings. If found, the defined index for the matched capability is returned.

The following indices are defined for the capability strings listed below.

`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`”
`SCSI_CAP_RESET_NOTIFICATION`	“`reset-notification`”
`SCSI_CAP_QFULL_RETRIES`	“`qfull-retries`”
`SCSI_CAP_QFULL_RETRY_INTERVAL`	“`qfull-retry-interval`”
`SCSI_CAP_LUN_RESET`	“`lun-reset`”
`SCSI_CAP_CDB_LEN`	“`max-cdb-length`”
`SCSI_CAP_TRAN_LAYER_RETRIES`	“tran-layer-retries”

Return Values The `scsi_hba_lookup_capstr()` function returns a non-negative index value that corresponds to the capability string. If the string does not match a known capability, `−1` is returned.

Context The `scsi_hba_lookup_capstr()` function can be called from user, interrupt, or kernel context.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `tran_getcap`(9E), `tran_setcap`(9E), `scsi_ifgetcap`(9F), `scsi_ifsetcap`(9F), `scsi_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()`:

The `scsi_hba_pkt_alloc()` function allocates space for a `scsi_pkt` structure. HBA drivers must 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.

The `scsi_hba_pkt_alloc()` function copies the `scsi_address`(9S) structure pointed to by *ap* to the `pkt_address` field in the `scsi_pkt`(9S).

The `scsi_hba_pkt_alloc()` function 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()`:

The `scsi_hba_pkt_free()` function frees the space allocated for the `scsi_pkt`(9S) structure.

Return Values The `scsi_hba_pkt_alloc()` function returns a pointer to the `scsi_pkt` structure, or `NULL` if no space is available.

Context The `scsi_hba_pkt_alloc()` function can be called from user, interrupt, or kernel context. Drivers must not allow `scsi_hba_pkt_alloc()` to sleep if called from an interrupt routine.

The `scsi_hba_pkt_free()` function can be called from user, interrupt, or kernel context.

See Also `tran_init_pkt`(9E), `scsi_address`(9S), `scsi_pkt`(9S)

Writing Device Drivers

Name scsi_hba_probe – default SCSI HBA probe function

Synopsis

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



int scsi_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 `scsi_hba_probe()` is a function providing the semantics of `scsi_probe`(9F). An HBA driver may call `scsi_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 `scsi_hba_probe` directly.

Return Values See `scsi_probe`(9F) for the return values from `scsi_hba_probe()`.

Context `scsi_hba_probe()` should only be called from the HBA's `tran_tgt_probe`(9E) entry point.

See Also `tran_tgt_probe`(9E), `scsi_probe`(9F), `scsi_device`(9S)

Writing Device Drivers

Name scsi_hba_tran_alloc, scsi_hba_tran_free – allocate and free transport structures

Synopsis

```
#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);
```

Interface Level Solaris architecture specific (Solaris DDI).

Parameters

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 `scsi_hba_tran_alloc()`:

The `scsi_hba_tran_alloc()` function 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()`:

The `scsi_hba_tran_free()` function is used to free the `scsi_hba_tran`(9S) structure allocated by `scsi_hba_tran_alloc()`.

Return Values The `scsi_hba_tran_alloc()` function returns a pointer to the allocated transport structure, or NULL if no space is available.

Context The `scsi_hba_tran_alloc()` function can be called from user, interrupt, or kernel context. Drivers must not allow `scsi_hba_tran_alloc()` to sleep if called from an interrupt routine.

The `scsi_hba_tran_free()` function can be called from user, interrupt, or kernel context context.

See Also `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 Solaris DDI specific (Solaris DDI).

Parameters

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 `scsi_ifsetcap()` function is used by target drivers to set the capabilities of the host adapter driver. The *cap* pointer is a name-value pair identified by a null-terminated character string and the integer value of the *cap*. The current value of the capability can be retrieved with the `scsi_ifgetcap()` function. If the *whom* value is `0`, all target drivers are affected. Otherwise, the `scsi_address` structure pointed to by *ap* is the only target that is affected.

The driver should confirm that `scsi_ifsetcap()` and `scsi_ifsetcap()` functions are called with a *cap* that points to a capability which is supported by the device.

The following capabilities have been defined:

`dma-max`	Maximum `dma` transfer size that is supported by the host adapter.
`dma-max-arch`	Maximum `dma` transfer size that is supported by system. Takes the host adapter and system architecture into account. This is useful for target drivers which do not support partial DMAs on systems which do not have an IOMMU. In this case, the DMA can also be limited by the host adapters “scatter/gather” list constraints. The “`dma-max-arch`” capability can not be set. It is implemented with this command and does not rely on a `tran_getcap(9E)` response from the HBA.
`msg-out`	Message out capability that is supported by the host adapter: `0` disables, `1` enables.
`disconnect`	Disconnect capability that is supported by the host adapter: `0` disables, `1` enables.

`synchronous`	Synchronous data transfer capability that is supported by the host adapter: `0` disables, `1` enables.
`wide-xfer`	Wide transfer capability that is supported by the host adapter: `0` disables, `1` enables.
`parity`	Parity checking capability that is supported by host adapter: `0` disables, `1` enables.
`initiator-id`	Host bus address that is returned.
`untagged-qing`	Host adapter capability that supports internal queueing of commands without tagged queueing: `0` disables, `1` enables.
`tagged-qing`	Host adapter capability that supports queuing: `0` disables, `1` enables.
`auto-rqsense`	Host adapter capability that supports auto request sense on check conditions: `0` disables, `1` enables.
`sector-size`	Capability that is set by the target driver to inform the HBA of the granularity, in bytes, of the DMA breakup. The HBA DMA limit structure is set to reflect the byte total of this setting. See `ddi_dma_lim_sparc`(9S) or `ddi_dma_lim_x86`(9S). The `sector-size` should be set to the size of the physical disk sector. The capability defaults to 512 bytes.
`total-sectors`	Capability that is set by the target driver to inform the HBA of the total number of sectors on the device returned by the SCSI `get capacity` command. This capability must be set before the target driver “gets" the `geometry` capability.
`geometry`	Capability that returns the HBA geometry of a target disk. The target driver sets 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 the target disk, `scsi_ifgetcap()` can return `-1` to indicate that the geometry is not defined. For example, if the HBA BIOS supports Logical Block Addressing for the target disk, `scsi_ifgetcap()` returns `-1`. Attempts to retreive the “virtual geometry” from the target driver, such as the `DKIOCG_VIRTGEOM` ioctl, will fail. See `dkio`(7I) for more information about `DKIOCG_VIRTGEOM`.
`reset-notification`	Host adapter capability that supports bus reset notification: `0` disables, `1` enables. See `scsi_reset_notify`(9F).

`linked-cmds`	Host adapter capability that supports linked commands: `0` disables, `1` enables.
`qfull-retries`	Capability that enables or disables `QUEUE FULL` handling. If `0`, the HBA will not retry a command when a `QUEUE FULL` status is returned. If the value is greater than `0`, the HBA driver retries the command a specified number of times at an interval determined by the `qfull-retry-interval`. The range for `qfull-retries` is `0-255`.
`qfull-retry-interval`	Capability that sets the retry interval in milliseconds (`ms`) for commands completed with a `QUEUE FULL` status. The range for `qfull-retry-intervals` is `0-1000` `ms`.
`lun-reset`	Capability that is created with a value of zero by HBA drivers that support the `RESET_LUN` flag in the `tran_reset`(9E) function. If it exists, the `lun-reset` value can be set to `1` by target drivers to allow the 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 `tran_reset()` function of the HBA driver. If `lun-reset` exists and has a value of `1`, the `tran_reset()` function of the HBA driver can be called with the `RESET_LUN` flag.
interconnect-type	Capability held in the `tran_interconnect_type` element of struct `scsi_hba_tran` that indicates the HBA transport interconnect type . The integer value of the interconnect type of the transport is defined in the `services.h` header file.
max-cdb-length	Host adapter capability of the maximum supported CDB (Command Descriptor Block) length. The target driver asks for the capability at attach time. If the HBA driver supports the capability, the maximum length of the CDB is returned in bytes. The target driver can then use that value to determine which CDB is used for the HBA. If the HBA driver does not support the `max-cdb-length` capability, the default value of the target driver is used for the CDB determination.

Return Values

The `scsi_ifsetcap()` function 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.

The `scsi_ifgetcap()` function returns the current value of a capability, or:

−1 If the capability was not defined.

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_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_ifgetcap(&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;
}
```

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

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

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 Level Solaris DDI specific (Solaris DDI).

Parameters

ap	Pointer to a `scsi_address`(9S) structure.
pktp	A pointer to a `scsi_pkt`(9S) structure.
bp	Pointer to a `buf`(9S) 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. Valid values are:

`0`	No status back.
`1`	Return SCSI status byte.
`sizeof(scsi_arq_status)`	Return status information in a `scsi_arq_status` structure. This will include up to 20 bytes of sense data. Please refer to `scsi_arq_status`(9S) for more information.
`EXTCMDS_STATUS_SIZE`	Same as preceding.

privatelen	The required length for the *pkt_private* area.
flags	Flags modifier.
callback	A pointer to a callback function, `NULL_FUNC`, or `SLEEP_FUNC`.
arg	The *callback* function argument.

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.

Return Values

The `scsi_init_pkt()` function 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.

Context If *callback* is `SLEEP_FUNC`, then this routine can be called only from user-level code. Otherwise, it can be called from user, interrupt, or kernel context. The *callback* function may not block or call routines that block.

Examples **EXAMPLE 1** Allocating a Packet Without DMA Resources Attached

To allocate a packet without DMA resources attached, use:

```
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:

```
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:

```
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:

```
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:

```
my_pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
          1, sizeof (struct buf *), PKT_DMA_PARTIAL,
          SLEEP_FUNC, NULL);
```

See Also `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

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 to `EFAULT`.

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 to `EINVAL`.

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 Level Solaris DDI specific (Solaris DDI).

Parameters

dip	Pointer to the `dev_info` structure.
drv_name	String naming the device.
level	Error level.
fmt	Display format.

Description The `scsi_log()` function 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 *format* 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 The `scsi_log()` function may be called from user, interrupt, or kernel context.

See Also `kernel`(1M), `sd`(7D), `cmn_err`(9F), `scsi_errmsg`(9F)

Writing Device Drivers

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 The `scsi_pktalloc()` function 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.

The *cmdlen* parameter 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.

The *dmatoken* parameter 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.

The `scsi_pktfree()` function frees the packet.

The `scsi_resfree()` function 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 can be called only from user or kernel context. Otherwise, it can be called from user, kernel, or interrupt context. The *callback* function may not block or call routines that block. Both deallocation routines can be called from user, kernel, or interrupt context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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_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 The `scsi_poll()` function 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 The `scsi_poll()` function returns:

`0`	command completed successfully.
`–1`	command failed.

Context The `scsi_poll()` function can be called from user, interrupt, or kernel context. 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 Level

Solaris DDI specific (Solaris DDI).

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.

Examples

EXAMPLE 1 Using `scsi_probe()`

```
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.

Name scsi_reset – reset a SCSI bus or target

Synopsis

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



int scsi_reset(struct scsi_address *ap, int level);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

ap	Pointer to the `scsi_address` structure.
level	The level of reset required.

Description The `scsi_reset()` function 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 The `scsi_reset()` function returns:

`1`	Upon success.
`0`	Upon failure.

Context The `scsi_reset()` function can be called from user, interrupt, or kernel 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 Solaris DDI specific (Solaris DDI).

Parameters

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 The `scsi_reset_notify()` function 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 The `scsi_reset_notify()` function can be called from user, interrupt, or kernel context.

See Also `scsi_address`(9S), `scsi_ifgetcap`(9F)

Writing Device Drivers

Name

scsi_setup_cdb – setup SCSI command descriptor block (CDB)

Synopsis

```
int scsi_setup_cdb(union scsi_cdb *cdbp, uchar_t cmd, uint_t addr, uint_t cnt,
     uint_t othr_cdb_data);
```

Interface Level

Solaris DDI specific (Solaris DDI).

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

The `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.

The `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

The `scsi_setup_cdb()` function returns:

`1`	Upon success.
`0`	Upon failure.

Context

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

See Also

`makecom`(9F), `scsi_pkt`(9S)

Writing Device Drivers

American National Standard Small Computer System Interface-2 (SCSI-2)

American National Standard SCSI-3 Primary Commands (SPC)

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:

`SCSIPROBE_NOMEM`	No space available for structures.
`SCSIPROBE_EXISTS`	Device exists and inquiry data is valid.
`SCSIPROBE_NONCCS`	Device exists but inquiry data is not valid.
`SCSIPROBE_FAILURE`	Polled command failure.
`SCSIPROBE_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:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

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)`.

Name scsi_sync_pkt – synchronize CPU and I/O views of memory

Synopsis

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



void scsi_sync_pkt(struct scsi_pkt *pktp);
```

Interface Level Solaris DDI specific (Solaris DDI).

Parameters

pktp	Pointer to a `scsi_pkt`(9S) structure.

Description The `scsi_sync_pkt()` function 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 The `scsi_sync_pkt()` function may be called from user, interrupt, or kernel 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 Level 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 The `scsi_transport()` function 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 The `scsi_transport()` function can be called from user, interrupt, or kernel context.

Examples **EXAMPLE 1** Using `scsi_transport()`

```
if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {
      scsi_log(devp, 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

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 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()`.

Name

scsi_vu_errmsg – display a SCSI request sense message

Synopsis

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



void scsi_vu_errmsg(struct scsi_pkt *pktp, char *drv_name, int severity,
     int err_blkno, struct scsi_key_strings *cmdlist, struct scsi_extended_sense *sensep,
     struct scsi_asq_key_strings *asc_list, char **decode_fru struct scsi_device*, char *, int, char)
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

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.

Description

This function is very similar to `scsi_errmsg`(9F) but allows decoding of vendor-unique ASC/ASCQ and FRU information.

The `scsi_vu_errmsg()` function 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:

```
Severity Value:      String:
SCSI_ERR_ALL         All
SCSI_ERR_UNKNOWN     Unknown
SCSI_ERR_INFO        Information
SCSI_ERR_RECOVERED   Recovered
SCSI_ERR_RETRYABLE   Retryable
SCSI_ERR_FATAL       Fatal
```

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:

```
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
        };
```

Context The `scsi_vu_errmsg()` function may be called from user, interrupt, or kernel context.

Examples **EXAMPLE 1** Using `scsi_vu_errmsg()`

```
struct scsi_asq_key_strings cd_slist[] = {
        0x81, 0, "Logical Unit is inaccessable",
        -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 inaccessable), 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

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.

The `sema_p_sig()` function 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.

The `sema_tryp()` function 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.
`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, interrupt, or kernel context, except for `sema_init()` and `sema_destroy()`, which can be called from user or kernel 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

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

The `sprintf()` function 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>\n` in *buf*.

```
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

The `sprintf()` function returns its first argument, *buf*.

The `snprintf()` function 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

The `sprintf()` and `snprintf()` functions and can be called from user, interrupt, or kernel context.

See Also

Writing Device Drivers

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 Level Solaris DDI specific (Solaris DDI).

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.

Description

`stoi()` The `stoi()` function 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()` The `numtos()` function 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 The `stoi()` function returns the integer value of the string *str*.

Context The `stoi()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

Notes The `stoi()` function handles only positive integers; it does not handle leading minus signs.

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

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

The `strchr()` and `strrchr()` functions return a pointer to a character, or `NULL`, if the search fails.

Context

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

See Also

`strcmp(9F)`

Writing Device Drivers

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 Level Solaris DDI specific (Solaris DDI).

Parameters

s1, *s2*	Pointers to character strings.
n	Count of characters to be compared.

Description

`strcmp()` The `strcmp()` function 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()` The `strncmp()` function 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.

Return Values The `strcmp()` function returns `0` if the strings are the same, or *(*s1 - *s2)* for the last characters compared if they differ.

The `strcasecmp()` and `strncasecmp()` functions return values in the same fashion as `strcmp()` and `strncmp()`, respectively.

The `strncmp()` function 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, interrupt, or kernel context.

See Also *Writing Device Drivers*

Name

strcpy, strlcat, strlcpy, strncat, strncpy, strspn – String operations.

Synopsis

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



char *strcpy(char *dst, const char *src);

size_t strlcat(char *dst, const char *src, size_t dstsize);

size_t  strlcpy(char *dst, const char *src, size_t dstsize);

char *strncat(char *restrict s1, const char *restrict  s2, size_t n);

char *strncpy(char *dst, const char *src, size_t n);

size_t strspn(const char *s1, const char *s2);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

dst, *src*	Pointers to character strings.
s1, *s2*	Pointers to character strings.
n	Count of characters to be copied.

Description

The arguments *dst*, *src*, *s1* and *s2* point to strings. The `strcpy()`, `strlcpy()`, `strncpy()`, `strlcat()` and `strncat()` functions all alter their first argument. These functions do not check for overflow of the array pointed to by the first argument.

`strcpy()` The `strcpy()` function copies characters in the string *src* to *dst*, terminating at the first null character in *src*, and returns *dst* to the caller. No bounds checking is done.

`strncpy()` The `strncpy()` function copies *src* to *dst*, null-padding or truncating at *n* bytes, and returns *dst*. No bounds checking is done.

`strlcpy()` The `strlcpy()` function copies a maximum of *dstsize*–`1` characters (where *dstsize* represents the size of the string buffer *dst*) from *src* to *dst*, truncating *src* if necessary. The result is always null-terminated. The function returns `strlen`(*src*). Buffer overflow can be checked as follows:

```
if (strlcpy(dst, src, dstsize) >= dstsize)
                        return (-1);
```

`strncat()` The `strncat()` function appends a maximum of *n* characters. The initial character of *s2* overrides the null character at the end of *s1*.

`strlcat()` The `strlcat()` function appends a maximum of (*dstsize*-`strlen`(*dst*)-`1`) characters of *src* to *dst* (where *dstsize* represents the size of the string buffer *dst*). If the string pointed to by *dst* contains a null-terminated string that fits into dstsize bytes when `strlcat()` is called, the string pointed to by *dst* is a null-terminated string that fits in dstsize bytes (including the terminating null character) when it completes, and the initial character of *src* overrides the null character at the end of *dst*. If the string pointed to by *dst* is longer than dstsize bytes when `strlcat()` is called, the string pointed to by *dst* is not changed. The function returns min{*dstsize*, `strlen`(*dst*)}+`strlen`(*src*). Buffer overflow can be checked as follows:

```
if (strlcat(dst, src, dstsize) >= dstsize)
                      return -1;
```

`strspn()` The `strspn()` function returns the length of the initial segment of string `s1` that consists entirely of characters from string *s2*.

Return Values The `strcpy()`, `strncat()` and `strncpy()` functions return *dst*.

For `strlcat()`, `strlcpy()` and `strspn()` functions, see the Description section.

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

See Also `strlen`(9F), `strcmp`(9F), `bcopy`(9F), `ddi_copyin`(9F)

Writing Device Drivers

Notes If copying takes place between objects that overlap, the behavior of `strcpy()`, `strlcat()`, `strlcpy()`, `strncat()`, `strncpy()`, `strspn()` is undefined.

Name	strlen – determine the number of non-null bytes in a string
Synopsis	`#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	The `strlen()` function returns the number of non-null bytes in the string argument *s*.
Return Values	The `strlen()` function returns the number of non-null bytes in *s*.
Context	The `strlen()` function can be called from user, interrupt, or kernel context.
See Also	*Writing Device Drivers*

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 Level Architecture independent level 1 (DDI/DKI).

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 The `strlog()` function 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 The `strlog()` function returns `0` if it fails to submit the message to the `log`(7D) driver and `1` otherwise.

Context The `strlog()` function can be called from user, interrupt, or kernel 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

Name	strqget – get information about a queue or band of the queue

Synopsis

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



int strqget(queue_t *q, qfields_t what, unsigned char pri, void *valp);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

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 The `strqget()` function 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 The `strqget()` function can be called from user, interrupt, or kernel 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

```
#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 The `strqset()` function 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 The `strqset()` function can be called from user, interrupt, or kernel 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`.

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 that appears *after* the `C` keyword struct of the native form.
umodel	A bit field that contains either the `ILP32` model bit (`DATAMODEL_ILP32`), or the `LP64` model bit (`DATAMODEL_LP64`). In an `ioctl`(9E), these bits are present in the flag parameter. In a `devmap`(9E), the bits are present in the model parameter `mmap`(9E). The `ddi_mmap_get_model`(9F) can be called 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 that can contain substructures. If the structures contain substructures, unions, or arrays, the *field* can be whatever complex expression would naturally follow 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 described in the following sections.

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, 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 it 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 a `devmap`(9E) routine, *umodel* is the model parameter. In a `mmap`(9E) routine, *umodel* is the return value of `ddi_mmap_get_model`(9F). This macro is intended only for handles created with `STRUCT_DECL()`.

STRUCT_HANDLE(structname, handle)

Declares a structure handle *handle* but, unlike `STRUCT_DECL()`, it does not allocate an instance of "struct".

void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)

Initializes handle 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, *umodel* is the model parameter. In an `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*, depending on the data model associated with *handle*. If the data model stored by `STRUCT_INIT()` or

	`STRUCT_SET_HANDLE()` is `DATAMODEL_ILP32`, the size of the `ILP32` form is returned. Otherwise, the size of the native form is returned.
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 casts caddr32_t to a (void *) when it is accessed. Failure to use this macro for a pointer leads 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 another expression, but only as a statement.
STRUCT_FSETP(handle, field, val)	This is the equivalent of STRUCT_FGETP() for STRUCT_FGET(), with the same exceptions. Like STRUCT_FSET, STRUCT_FSETP should not be used within another expression, but only as a statement.
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:

```
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.

```
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:

```
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)
{
        struct copyresp *cp = (struct copyresp *)msgp->b_rptr;
        STRUCT_HANDLE(strbuf, sb);

        if (msgp->b_cont->b_cont != NULL) {
                msgp->b_cont = msgpullup(msgp->b_cont, -1);
                if (msgp->b_cont == NULL) {
                        miocnak(q, msgp, 0, ENOSR);
                        return;
                }
        }
        STRUCT_SET_HANDLE(sb, cp->cp_flag, (void *)msgp->b_cont->b_rptr);
        if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
                miocnak(q, msgp, 0, ENOSR);
                return;
        }
        ...
        miocack(q, msgp, 0, 0);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also

`devmap(9E)`, `ioctl(9E)`, `mmap(9E)`,`ddi_mmap_get_model(9F)`

Writing Device Drivers

STREAMS Programming Guide

Name swab – swap bytes in 16-bit halfwords

Synopsis

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



void swab(void *src, void *dst, size_t nbytes);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

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 The `swab()` function 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 The `swab()` function can be called from user, interrupt, or kernel context.

See Also *Writing Device Drivers*

Notes Since `swab()` operates byte-by-byte, it can be used on non-aligned buffers.

Name

taskq, ddi_taskq_create, ddi_taskq_destroy, ddi_taskq_dispatch, ddi_taskq_wait, ddi_taskq_suspend, taskq_suspended, ddi_taskq_resume – Kernel task queue operations

Synopsis

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

ddi_taskq_t *ddi_taskq_create(dev_info_t *dip, const char *name,
     int nthreads, pri_t  pri, uint_t  cflags);

void ddi_taskq_destroy(ddi_taskq_t *tq);

int ddi_taskq_dispatch(ddi_taskq_t *tq, void (* func)(void *),
     void *arg, uint_t dflags);

void ddi_taskq_wait(ddi_taskq_t *tq);

void ddi_taskq_suspend(ddi_taskq_t *tq);

boolean_t ddi_taskq_suspended(ddi_taskq_t *tq);

void ddi_taskq_resume(ddi_taskq_t *tq);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's dev_info structure. May be NULL for kernel modules that do not have an associated dev_info structure.	
name	Descriptive string. Only alphanumeric characters can be used in name and spaces are not allowed. The name should be unique.	
nthreads	Number of threads servicing the task queue. Note that the request ordering is guaranteed (tasks are processed in the order scheduled) if the `taskq` is created with a single servicing thread.	
pri	Priority of threads servicing the task queue. Drivers and modules should specify TASKQ_DEFAULTPRI.	
cflags	Should pass 0 as flags.	
func	Callback function to call.	
arg	Argument to the callback function.	
dflags	Possible *dflags* are:	
	DDI_SLEEP	Allow sleeping (blocking) until memory is available.
	DDI_NOSLEEP	Return DDI_FAILURE immediately if memory is not available.
tq	Pointer to a task queue (ddi_taskq_t *).	
tp	Pointer to a thread structure.	

Description

A kernel task queue is a mechanism for general-purpose asynchronous task scheduling that enables tasks to be performed at a later time by another thread. There are several reasons why you may utilize asynchronous task scheduling:

1. You have a task that isn't time-critical, but a current code path that is.
2. You have a task that may require grabbing locks that a thread already holds.
3. You have a task that needs to block (for example, to wait for memory), but a have a thread that cannot block in its current context.
4. You have a code path that can't complete because of a specific condition, but also can't sleep or fail. In this case, the task is immediately queued and then is executed after the condition disappears.
5. A task queue is just a simple way to launch multiple tasks in parallel.

A task queue consists of a list of tasks, together with one or more threads to service the list. If a task queue has a single service thread, all tasks are guaranteed to execute in the order they were dispatched. Otherwise they can be executed in any order. Note that since tasks are placed on a list, execution of one task and should not depend on the execution of another task or a deadlock may occur. A `taskq` created with a single servicing thread guarantees that all the tasks are serviced in the order in which they are scheduled.

The `ddi_taskq_create()` function creates a task queue instance.

The `ddi_taskq_dispatch()` function places `taskq` on the list for later execution. The *dflag* argument specifies whether it is allowed sleep waiting for memory. DDI_SLEEP dispatches can sleep and are guaranteed to succeed. DDI_NOSLEEP dispatches are guaranteed not to sleep but may fail (return `DDI_FAILURE`) if resources are not available.

The `ddi_taskq_destroy()` function waits for any scheduled tasks to complete, then destroys the `taskq`. The caller should guarantee that no new tasks are scheduled for the closing `taskq`.

The `ddi_taskq_wait()` function waits for all previously scheduled tasks to complete. Note that this function does not stop any new task dispatches.

The `ddi_taskq_suspend()` function suspends all task execution until `ddi_taskq_resume()` is called. Although `ddi_taskq_suspend()` attempts to suspend pending tasks, there are no guarantees that they will be suspended. The only guarantee is that all tasks dispatched after `ddi_taskq_suspend()` will not be executed. Because it will trigger a deadlock, the `ddi_taskq_suspend()` function should never be called by a task executing on a `taskq`.

The `ddi_taskq_suspended()` function returns `B_TRUE` if `taskq` is suspended, and `B_FALSE` otherwise. It is intended to ASSERT that the task queue is suspended.

The `ddi_taskq_resume()` function resumes task queue execution.

Return Values The `ddi_taskq_create()` function creates an opaque handle that is used for all other `taskq` operations. It returns a `taskq` pointer on success and NULL on failure.

The `ddi_taskq_dispatch()` function returns `DDI_FAILURE` if it can't dispatch a task and returns `DDI_SUCCESS` if dispatch succeeded.

The `ddi_taskq_suspended()` function returns `B_TRUE` if `taskq` is suspended. Otherwise `B_FALSE` is returned.

Context All functions may be called from the user or kernel contexts.

Addtionally, the `ddi_taskq_dispatch` function may be called from the interrupt context only if the DDI_NOSLEEP flag is set.

Name

testb – check for an available buffer

Synopsis

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



int testb(size_t size, uint_t pri);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

size	Size of the requested buffer.
pri	Priority of the allocb request.

Description

The `testb()` function 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

The `testb()` function can be called user, interrupt, or kernel 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  xxxsrv(q)
 2      queue_t *q;
 3  {
 4    mblk_t *mp;
 5    mblk_t *nmp;
        . . .
 6    if ((nmp = copymsg(mp)) == NULL) {
 7        putbq(q, mp);
```

EXAMPLE 1 `testb()` example *(Continued)*

```
   timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
   return;
 }
. . .
}

tryagain(q)
  queue_t *q;
{
 register int can_alloc = 0;
 register int num_blks = 0;
 register mblk_t *mp;

 if (!q->q_first)
   return;
 for (mp = q->q_first; mp; mp = mp->b_cont) {
   num_blks++;
   can_alloc += testb((mp->b_datap->db_lim -
     mp->b_datap->db_base), BPRI_MED);
 }
 if (num_blks == can_alloc)
   qenable(q);
 else
   timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
}
```

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.

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 `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

The `timeout()` function returns an opaque non-zero `timeout` identifier that can be passed to `untimeout`(9F) to cancel the request.

Context

The `timeout()` function can be called from user, interrupt, or kernel 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.

```
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);
```

EXAMPLE 1 Using `timeout()` *(Continued)*

```
        xsp->timeout_id = 0;
}
static uint_t
xxintr(caddr_t arg)
{
        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_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

`bufcall`(9F), `cv_timedwait`(9F), `ddi_in_panic`(9F), `delay`(9F), `drv_usectohz`(9F), `untimeout`(9F)

Writing Device Drivers

Name u8_strcmp – UTF-8 string comparison function

Synopsis

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

int u8_strcmp(const char *s1, const char *s2, size_t n,
     int flag, size_t unicode_version, int *errno);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

s1, *s2*	Pointers to null-terminated UTF-8 strings
n	The maximum number of bytes to be compared. If 0, the comparison is performed until either or both of the strings are examined to the string terminating null byte.
flag	The possible comparison options constructed by a bit-wise-inclusive-OR of the following values:

`U8_STRCMP_CS`
Perform case-sensitive string comparison. This is the default.

`U8_STRCMP_CI_UPPER`
Perform case-insensitive string comparison based on Unicode upper case converted results of *s1* and *s2*.

`U8_STRCMP_CI_LOWER`
Perform case-insensitive string comparison based on Unicode lower case converted results of *s1* and *s2*.

`U8_STRCMP_NFD`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form D.

`U8_STRCMP_NFC`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form C.

`U8_STRCMP_NFKD`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form KD.

`U8_STRCMP_NFKC`
Perform string comparison after *s1* and *s2* have been normalized by using Unicode Normalization Form KC.

Only one case-sensitive or case-insensitive option is allowed. Only one Unicode Normalization option is allowed.

unicode_version The version of Unicode data that should be used during comparison. The following values are supported:

`U8_UNICODE_320`
Use Unicode 3.2.0 data during comparison.

`U8_UNICODE_500`
Use Unicode 5.0.0 data during comparison.

`U8_UNICODE_LATEST`
Use the latest Unicode version data available, which is Unicode 5.0.0.

errno A non-zero value indicates that an error has occurred during comparison. The following values are supported:

`EBADF`	The specified option values are conflicting and cannot be supported.
`EILSEQ`	There was an illegal character at *s1*, *s2*, or both.
`EINVAL`	There was an incomplete character at *s1*, *s2*, or both.
`ERANGE`	The specified Unicode version value is not supported.

Description

After proper pre-processing, the `u8_strcmp()` function compares two UTF-8 strings byte-by-byte, according to the machine ordering defined by the corresponding version of the Unicode Standard.

When multiple comparison options are specified, Unicode Normalization is performed after case-sensitive or case-insensitive processing is performed.

Return Values

The `u8_strcmp()` function returns an integer greater than, equal to, or less than 0 if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

When `u8_strcmp()` detects an illegal or incomplete character, such character causes the function to set *errno* to indicate the error. Afterward, the comparison is still performed on the resultant strings and a value based on byte-by-byte comparison is always returned.

Context

The `u8_strcmp()` function can be called from user or interrupt context.

Examples

EXAMPLE 1 Perform simple default string comparison.

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

int
docmp_default(const char *u1, const char *u2) {
    int result;
    int ;

    result = u8_strcmp(u1, u2, 0, 0, U8_UNICODE_LATEST, &errno);
    if (errno == EILSEQ)
        return (-1);
    if (errno == EINVAL)
```

EXAMPLE 1 Perform simple default string comparison. *(Continued)*

```
        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);
```

EXAMPLE 2 Perform upper case based case-insensitive comparison with Unicode 3.2.0 date.

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

int
docmp_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errno;

    result = u8_strcmp(u1, u2, 0, U8_STRCMP_CI_UPPER,
        U8_UNICODE_320, &errno);
    if (errno == EILSEQ)
        return (-1);
    if (errno == EINVAL)
        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);

    return (result);
}
```

EXAMPLE 3 Perform Unicode Normalization Form D.

Perform Unicode Normalization Form D and uppercase-based case-insensitive comparison with Unicode 3.2.0 date.

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

int
docmp_nfd_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errno;

    result = u8_strcmp(u1, u2, 0,
        (U8_STRCMP_NFD|U8_STRCMP_CI_UPPER), U8_UNICODE_320,
        &errno);
    if (errno == EILSEQ)
```

EXAMPLE 3 Perform Unicode Normalization Form D. *(Continued)*

```
        return (-1);
    if (errno == EINVAL)
        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);

    return (result);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`u8_validate`(3C), `u8_textprep_str`(3C), `u8_validate`(3C), `attributes`(5), `u8_textprep_str`(9F), `u8_validate`(9F), `uconv_u16tou32`(9F)

The Unicode Standard (`http://www.unicode.org`)

Name u8_textprep_str – string-based UTF-8 text preparation function

Synopsis

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

size_t u8_textprep_str(char *inarray, size_t *inlen,
     char *outarray, size_t *outlen, int flag,
     size_t unicode_version, int *errno);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

inarray	A pointer to a byte array containing a sequence of UTF-8 character bytes to be prepared.
inlen	As input argument, the number of bytes to be prepared in *inarray*. As output argument, the number of bytes in *inarray* still not consumed.
outarray	A pointer to a byte array where prepared UTF-8 character bytes can be saved.
outlen	As input argument, the number of available bytes at *outarray* where prepared character bytes can be saved. As output argument, after the conversion, the number of bytes still available at *outarray*.
flag	The possible preparation options constructed by a bitwise-inclusive-OR of the following values:

`U8_TEXTPREP_IGNORE_NULL`

Normally `u8_textprep_str()` stops the preparation if it encounters null byte even if the current *inlen* is pointing to a value bigger than zero.

With this option, null byte does not stop the preparation and the preparation continues until *inlen* specified amount of *inarray* bytes are all consumed for preparation or an error happened.

`U8_TEXTPREP_IGNORE_INVALID`

Normally `u8_textprep_str()` stops the preparation if it encounters illegal or incomplete characters with corresponding *errno* values.

When this option is set, `u8_textprep_str()` does not stop the preparation and instead treats such characters as no need to do any preparation.

`U8_TEXTPREP_TOUPPER`

Map lowercase characters to uppercase characters if applicable.

`U8_TEXTPREP_TOLOWER`

Map uppercase characters to lowercase characters if applicable.

`U8_TEXTPREP_NFD`
Apply Unicode Normalization Form D.

`U8_TEXTPREP_NFC`
Apply Unicode Normalization Form C.

`U8_TEXTPREP_NFKD`
Apply Unicode Normalization Form KD.

`U8_TEXTPREP_NFKC`
Apply Unicode Normalization Form KC.

Only one case folding option is allowed. Only one Unicode Normalization option is allowed.

When a case folding option and a Unicode Normalization option are specified together, UTF-8 text preparation is done by doing case folding first and then Unicode Normalization.

If no option is specified, no processing occurs except the simple copying of bytes from input to output.

unicode_version The version of Unicode data that should be used during UTF-8 text preparation. The following values are supported:

`U8_UNICODE_320`
Use Unicode 3.2.0 data during comparison.

`U8_UNICODE_500`
Use Unicode 5.0.0 data during comparison.

`U8_UNICODE_LATEST`
Use the latest Unicode version data available which is Unicode 5.0.0 currently.

errno The error value when preparation is not completed or fails. The following values are supported:

`E2BIG`	Text preparation stopped due to lack of space in the output array.
`EBADF`	Specified option values are conflicting and cannot be supported.
`EILSEQ`	Text preparation stopped due to an input byte that does not belong to UTF-8.
`EINVAL`	Text preparation stopped due to an incomplete UTF-8 character at the end of the input array.
`ERANGE`	The specified Unicode version value is not a supported version.

Description The `u8_textprep_str()` function prepares the sequence of UTF-8 characters in the array specified by *inarray* into a sequence of corresponding UTF-8 characters prepared in the array specified by *outarray*. The *inarray* argument points to a character byte array to the first character in the input array and *inlen* indicates the number of bytes to the end of the array to be converted. The *outarray* argument points to a character byte array to the first available byte in the output array and *outlen* indicates the number of the available bytes to the end of the array. Unless *flag* is `U8_TEXTPREP_IGNORE_NULL`, `u8_textprep_str()` normally stops when it encounters a null byte from the input array regardless of the current *inlen* value.

If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and a sequence of input bytes does not form a valid UTF-8 character, preparation stops after the previous successfully prepared character. If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and the input array ends with an incomplete UTF-8 character, preparation stops after the previous successfully prepared bytes. If the output array is not large enough to hold the entire prepared text, preparation stops just prior to the input bytes that would cause the output array to overflow. The value pointed to by *inlen* is decremented to reflect the number of bytes still not prepared in the input array. The value pointed to by *outlen* is decremented to reflect the number of bytes still available in the output array.

Return Values The `u8_textprep_str()` function updates the values pointed to by *inlen* and *outlen* arguments to reflect the extent of the preparation. When `U8_TEXTPREP_IGNORE_INVALID` is specified, `u8_textprep_str()` returns the number of illegal or incomplete characters found during the text preparation. When `U8_TEXTPREP_IGNORE_INVALID` is not specified and the text preparation is successful, the function returns 0. If the entire string in the input array is prepared, the value pointed to by *inlen* will be 0. If the text preparation is stopped due to any conditions mentioned above, the value pointed to by *inlen* will be non-zero and *errno* is set to indicate the error. If such and any other error occurs, `u8_textprep_str()` returns (`size_t`)-1 and sets *errno* to indicate the error.

Context The `u8_textprep_str()` function can be called from user or interrupt context.

Examples **EXAMPLE 1** Simple UTF-8 text preparation

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
size_t ret;
char ib[MAXPATHLEN];
char ob[MAXPATHLEN];
size_t il, ol;
int err;
.
.
.
```

EXAMPLE 1 Simple UTF-8 text preparation *(Continued)*

```
/*
 * We got a UTF-8 pathname from somewhere.
 *
 * Calculate the length of input string including the terminating
 * NULL byte and prepare other arguments.
 */
(void) strlcpy(ib, pathname, MAXPATHLEN);
il = strlen(ib) + 1;
ol = MAXPATHLEN;

/*
 * Do toupper case folding, apply Unicode Normalization Form D,
 * ignore NULL byte, and ignore any illegal/incomplete characters.
 */
ret = u8_textprep_str(ib, &il, ob, &ol,
    (U8_TEXTPREP_IGNORE_NULL|U8_TEXTPREP_IGNORE_INVALID|
    U8_TEXTPREP_TOUPPER|U8_TEXTPREP_NFD), U8_UNICODE_LATEST, &err);
if (ret == (size_t)-1) {
    if (err == E2BIG)
        return (-1);
    if (err == EBADF)
        return (-2);
    if (err == ERANGE)
        return (-3);
    return (-4);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`u8_strcmp(3C)`, `u8_textprep_str(3C)`, `u8_validate(3C)`, `attributes(5)`, `u8_strcmp(9F)`, `u8_validate(9F)`, `uconv_u16tou32(9F)`

The Unicode Standard (`http://www.unicode.org`)

Name u8_validate – validate UTF-8 characters and calculate the byte length

Synopsis

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

int u8_validate(char *u8str, size_t n, char **list, int flag,
     int *errno);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

u8str	The UTF-8 string to be validated.
n	The maximum number of bytes in *u8str* that can be examined and validated.
list	A list of null-terminated character strings in UTF-8 that must be additionally checked against as invalid characters. The last string in *list* must be null to indicate there is no further string.
flag	Possible validation options constructed by a bitwise-inclusive-OR of the following values:

`U8_VALIDATE_ENTIRE`

By default, `u8_validate()` looks at the first character or up to *n* bytes, whichever is smaller in terms of the number of bytes to be consumed, and returns with the result.

When this option is used, `u8_validate()` will check up to *n* bytes from *u8str* and possibly more than a character before returning the result.

`U8_VALIDATE_CHECK_ADDITIONAL`

By default, `u8_validate()` does not use list supplied.

When this option is supplied with a list of character strings, `u8_validate()` additionally validates *u8str* against the character strings supplied with *list* and returns EBADF in *errno* if *u8str* has any one of the character strings in *list*.

`U8_VALIDATE_UCS2_RANGE`

By default, `u8_validate()` uses the entire Unicode coding space of U+0000 to U+10FFFF.

When this option is specified, the valid Unicode coding space is smaller to U+0000 to U+FFFF.

errno An error occurred during validation. The following values are supported:

`EBADF`	Validation failed because list-specified characters were found in the string pointed to by *u8str*.
`EILSEQ`	Validation failed because an illegal byte was found in the string pointed to by *u8str*.

`EINVAL`	Validation failed because an incomplete byte was found in the string pointed to by *u8str*.
`ERANGE`	Validation failed because character bytes were encountered that are outside the range of the Unicode coding space.

Description

The `u8_validate()` function validates *u8str* in UTF-8 and determines the number of bytes constituting the character(s) pointed to by *u8str*.

Return Values

If *u8str* is a null pointer, `u8_validate()` returns 0. Otherwise, `u8_validate()` returns either the number of bytes that constitute the characters if the next *n* or fewer bytes form valid characters, or -1 if there is an validation failure, in which case it may set *errno* to indicate the error.

Examples

EXAMPLE 1 Determine the length of the first UTF-8 character.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int errno;
.
.
.
len = u8_validate(u8, 4, (char **)NULL, 0, &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

EXAMPLE 2 Check if there are any invalid characters in the entire string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int n;
```

EXAMPLE 2 Check if there are any invalid characters in the entire string. *(Continued)*

```
int errno;
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, (char **)NULL, U8_VALIDATE_ENTIRE, &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

EXAMPLE 3 Check if there is any invalid character, including prohibited characters, in the entire string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int n;
int errno;
char *prohibited[4] = {
    ".", "..", "\\", NULL
};
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, prohibited,
    (U8_VALIDATE_ENTIRE|U8_VALIDATE_CHECK_ADDITIONAL), &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVAL);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
```

EXAMPLE 3 Check if there is any invalid character, including prohibited characters, in the entire string. *(Continued)*

```
                return (MYFS4_ERR_BADCHAR);
            default:
                return (-10);
        }
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

u8_strcmp(3C), u8_textprep_str(3C), u8_validate(3C), attributes(5), u8_strcmp(9F), u8_textprep_str(9F), uconv_u16tou32(9F)

The Unicode Standard (http://www.unicode.org)

Name

uconv_u16tou32, uconv_u16tou8, uconv_u32tou16, uconv_u32tou8, uconv_u8tou16, uconv_u8tou32 – Unicode encoding conversion functions

Synopsis

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

int uconv_u16tou32(const uint16_t *utf16str, size_t *utf16len,
     uint32_t *utf32str, size_t *utf32len, int flag);

int uconv_u16tou8(const uint16_t *utf16str, size_t *utf16len,
     uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u32tou16(const uint32_t *utf32str, size_t *utf32len,
     uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u32tou8(const uint32_t *utf32str, size_t *utf32len,
     uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u8tou16(const uchar_t *utf8str, size_t *utf8len,
     uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u8tou32(const uchar_t *utf8str, size_t *utf8len,
     uint32_t *utf32str, size_t *utf32len, int flag);
```

Interface Level

Solaris DDI specific (Solaris DDI).

Parameters

utf16str	A pointer to a `UTF-16` character string.
utf16len	As an input parameter, the number of 16-bit unsigned integers in *utf16str* as `UTF-16` characters to be converted or saved. As an output parameter, the number of 16-bit unsigned integers in *utf16str* consumed or saved during conversion.
utf32str	A pointer to a `UTF-32` character string.
utf32len	As an input parameter, the number of 32-bit unsigned integers in *utf32str* as `UTF-32` characters to be converted or saved. As an output parameter, the number of 32-bit unsigned integers in *utf32str* consumed or saved during conversion.
utf8str	A pointer to a `UTF-8` character string.
utf8len	As an input parameter, the number of bytes in *utf8str* as `UTF-8` characters to be converted or saved. As an output parameter, the number of bytes in *utf8str* consumed or saved during conversion.
flag	The possible conversion options that are constructed by a bitwise-inclusive-OR of the following values:

`UCONV_IN_BIG_ENDIAN`
The input parameter is in big endian byte ordering.

`UCONV_OUT_BIG_ENDIAN`
The output parameter should be in big endian byte ordering.

`UCONV_IN_SYSTEM_ENDIAN`
The input parameter is in the default byte ordering of the current system.

`UCONV_OUT_SYSTEM_ENDIAN`
The output parameter should be in the default byte ordering of the current system.

`UCONV_IN_LITTLE_ENDIAN`
The input parameter is in little endian byte ordering.

`UCONV_OUT_LITTLE_ENDIAN`
The output parameter should be in little endian byte ordering.

`UCONV_IGNORE_NULL`
The null or `U+0000` character should not stop the conversion.

`UCONV_IN_ACCEPT_BOM`
If the Byte Order Mark (BOM, `U+FEFF`) character exists as the first character of the input parameter, interpret it as the BOM character.

`UCONV_OUT_EMIT_BOM`
Start the output parameter with Byte Order Mark (BOM, `U+FEFF`) character to indicate the byte ordering if the output parameter is in `UTF-16` or `UTF-32`.

Description The `uconv_u16tou32()` function reads the given *utf16str* in `UTF-16` until `U+0000` (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The `UTF-16` characters that are read are converted into `UTF-32` and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as `UTF-32` characters.

The `uconv_u16tou8()` function reads the given *utf16str* in `UTF-16` until `U+0000` (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The `UTF-16` characters that are read are converted into `UTF-8` and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as `UTF-8` characters.

The `uconv_u32tou16()` function reads the given *utf32str* in `UTF-32` until `U+0000` (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified in *utf32len* is read. The `UTF-32` characters that are read are converted into `UTF-16` and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as `UTF-16` characters.

The `uconv_u32tou8()` function reads the given *utf32str* in `UTF-32` until `U+0000` (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified in *utf32len* is read. The `UTF-32` characters that are read are converted into `UTF-8` and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as `UTF-8` characters.

The `uconv_u8tou16()` function reads the given *utf8str* in `UTF-8` until the null ('`\0`') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The `UTF-8` characters that are read are converted into `UTF-16` and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as `UTF-16` characters.

The `uconv_u8tou32()` function reads the given *utf8str* in `UTF-8` until the null ('`\0`') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The `UTF-8` characters that are read are converted into `UTF-32` and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as `UTF-32` characters.

During the conversion, the input and the output parameters are treated with byte orderings specified in the *flag* parameter. When not specified, the default byte ordering of the system is used. The byte ordering *flag* value that is specified for `UTF-8` is ignored.

When `UCONV_IN_ACCEPT_BOM` is specified as the *flag* and the first character of the string pointed to by the input parameter is the BOM character, the value of the BOM character dictates the byte ordering of the subsequent characters in the string pointed to by the input parameter, regardless of the supplied input parameter byte ordering option *flag* values. If the `UCONV_IN_ACCEPT_BOM` is not specified, the BOM as the first character is treated as a regular Unicode character: Zero Width No Break Space (`ZWNBSP`) character.

When `UCONV_IGNORE_NULL` is specified, regardless of whether the input parameter contains `U+0000` or null byte, the conversion continues until the specified number of input parameter elements at *utf16len*, *utf32len*, or *utf8len* are entirely consumed during the conversion.

As output parameters, *utf16len*, *utf32len*, and *utf8len* are not changed if conversion fails for any reason.

Context

The `uconv_u16tou32()`, `uconv_u16tou8()`, `uconv_u32tou16()`, `uconv_u32tou8()`, `uconv_u8tou16()`, and `uconv_u8tou32()` functions can be called from user or interrupt context.

Return Values

Upon successful conversion, the functions return `0`. Upon failure, the functions return one of the following `errno` values:

`EILSEQ`	The conversion detected an illegal or out of bound character value in the input parameter.
`E2BIG`	The conversion cannot finish because the size specified in the output parameter is too small.
`EINVAL`	The conversion stops due to an incomplete character at the end of the input string.

`EBADF`	Conflicting byte-ordering option *flag* values are detected.

Examples

EXAMPLE 1 Convert a `UTF-16` string in little-endian byte ordering into `UTF-8` string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
uint16_t u16s[MAXNAMELEN + 1];
uchar_t u8s[MAXNAMELEN + 1];
size_t u16len, u8len;
int ret;
.
.
.
u16len = u8len = MAXNAMELEN;
ret = uconv_u16tou8(u16s, &u16len, u8s, &u8len,
    UCONV_IN_LITTLE_ENDIAN);
if (ret != 0) {
     /* Conversion error occurred. */
     return (ret);
}
.
.
.
```

EXAMPLE 2 Convert a `UTF-32` string in big endian byte ordering into little endian `UTF-16`.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
/*
  * An UTF-32 character can be mapped to an UTF-16 character with
  * two 16-bit integer entities as a "surrogate pair."
  */
uint32_t u32s[101];
uint16_t u16s[101];
int ret;
size_t u32len, u16len;
.
.
.
u32len = u16len = 100;
```

EXAMPLE 2 Convert a `UTF-32` string in big endian byte ordering into little endian `UTF-16`. *(Continued)*

```
ret = uconv_u32tou16(u32s, &u32len, u16s, &u16len,
    UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
if (ret == 0) {
     return (0);
} else if (ret == E2BIG) {
     /* Use bigger output parameter and try just one more time. */
     uint16_t u16s2[201];

     u16len = 200;
     ret = uconv_u32tou16(u32s, &u32len, u16s2, &u16len,
        UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
     if (ret == 0)
          return (0);
}

/* Otherwise, return -1 to indicate an error condition. */
return (-1);
```

EXAMPLE 3 Convert a `UTF-8` string into `UTF-16` in little-endian byte ordering.

Convert a `UTF-8` string into `UTF-16` in little-endian byte ordering with a Byte Order Mark (BOM) character at the beginning of the output parameter.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
uchar_t u8s[MAXNAMELEN + 1];
uint16_t u16s[MAXNAMELEN + 1];
size_t u8len, u16len;
int ret;
.
.
.
u8len = u16len = MAXNAMELEN;
ret = uconv_u8tou16(u8s, &u8len, u16s, &u16len,
    UCONV_IN_LITTLE_ENDIAN | UCONV_EMIT_BOM);
if (ret != 0) {
     /* Conversion error occurred. */
     return (ret);
}
.
```

EXAMPLE 3 Convert a `UTF-8` string into `UTF-16` in little-endian byte ordering. *(Continued)*

```
.
.
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `uconv_u16tou32(3C)`, `attributes(5)`

The Unicode Standard (`http://www.unicode.org`)

Notes Each `UTF-16` or `UTF-32` character maps to an `UTF-8` character that might need one to maximum of four bytes.

One `UTF-32` or `UTF-8` character can yield two 16-bit unsigned integers as a `UTF-16` character, which is a surrogate pair if the Unicode scalar value is bigger than `U+FFFF`.

Ill-formed `UTF-16` surrogate pairs are seen as illegal characters during the conversion.

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 rwflag,
     uio_t *uio_p);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters

address	Source/destination kernel address of the copy.
nbytes	Number of bytes to copy.
rwflag	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.

rwflag 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 The `uiomove()` function returns `0` upon success or `EFAULT` on failure.

Context User context only, if `uio_segflg` is set to `UIO_USERSPACE`. User, interrupt, or kernel 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.

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 The `unbufcall` function 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 The `unbufcall` function can be called from user, interrupt, or kernel context.

See Also `bufcall`(9F), `esbbcall`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name

unlinkb – remove a message block from the head of a message

Synopsis

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



mblk_t *unlinkb(mblk_t *mp);
```

Interface Level

Architecture independent level 1 (DDI/DKI).

Parameters

mp	Pointer to the message.

Description

The `unlinkb()` function 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, the `unlinkb()` function 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

The `unlinkb()` function can be called from user, interrupt, or kernel 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.

```
mblk_t *
makedata(mp)
    mblk_t *mp;
{
  mblk_t *nmp;

  nmp = unlinkb(mp);
  freeb(mp);
  return(nmp);
}
```

See Also

`linkb`(9F)

Writing Device Drivers

STREAMS Programming Guide

Name untimeout – cancel previous timeout function call

Synopsis

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



clock_t untimeout(timeout_id_t id);
```

Interface Level Architecture independent level 1 (DDI/DKI).

Parameters *id* Opaque timeout `ID` from a previous `timeout(9F)` call.

Description The `untimeout()` function cancels a pending `timeout(9F)` request. `untimeout()` 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 `untimeout()` or a deadlock may result.

Since no mutex should be held across the call to `untimeout()`, 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 `untimeout()` 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 The `untimeout()` function returns `-1` if the *id* is not found. Otherwise, it returns an integer value greater than or equal to `0`.

Context The `untimeout()` function can be called from user, interrupt, or kernel context.

Examples 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 */
        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

Name ureadc – add character to a uio structure

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 The `ureadc()` function 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 and `EFAULT` on failure.

Context The `ureadc()` function can be called from user, interrupt, or kernel context.

See Also `uiomove`(9F), `uwritec`(9F), `iovec`(9S), `uio`(9S)

Writing Device Drivers

Name

usb_alloc_request, usb_alloc_ctrl_req, usb_free_ctrl_req, usb_alloc_bulk_req, usb_free_bulk_req, usb_alloc_intr_req, usb_free_intr_req, usb_alloc_isoc_req, usb_free_isoc_req – Allocate and free USB transfer requests

Synopsis

```
#include <sys/usb/usba.h>

usb_ctrl_req_t *usb_alloc_ctrl_req(dev_info_t *dip, size_t len,
     usb_flags_t flags);

void usb_free_ctrl_req(usb_ctrl_req_t *request);

usb_bulk_req_t *usb_alloc_bulk_req(dev_info_t dip, size_t len,
     usb_flags_t flags);

void usb_free_bulk_req(usb_bulk_req_t *request);

usb_intr_req_t *usb_alloc_intr_req(dev_info_t *dip, size_t len,
     usb_flags_t flags);

void usb_free_intr_req(usb_intr_req_t *request);

usb_isoc_req_t *usb_alloc_isoc_req(dev_info_t *dip,
     uint_t isoc_pkts_count, size_t len, usb_flags_t flags);

void usb_free_isoc_req(usb_isoc_req_t *request);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

For `usb_alloc_ctrl_req()`, `usb_alloc_bulk_req()` and `usb_alloc_intr_req()`:

dip	Pointer to the device's dev_info structure.
len	Length of *data* for this request.
flags	Only USB_FLAGS_SLEEP is recognized. Wait for resources if not immediately available.

For `usb_alloc_isoc_req()`:

dip	Pointer to the device's dev_info structure.
isoc_pkts_count	Number of isochronous packet descriptors to associate with this request. Must be greater than zero.
len	Length of *data* for this isochronous request.
flags	Only USB_FLAGS_SLEEP is recognized. Wait for resources if not immediately available.

For `usb_free_ctrl_req()`, `usb_free_bulk_req()`, `usb_free_intr_req()` and `usb_free_isoc_req()`:

request	Pointer to the request structure to be freed. Can be `NULL`.

Description

The `usb_alloc_ctrl_req()`, `usb_alloc_bulk_req()`, `usb_alloc_intr_req()`, and `usb_alloc_isoc_req()` functions allocate control, bulk, interrupt, or isochronous requests. Optionally, these functions can also allocate an mblk of the specified length to pass data associated with the request. (For guidelines on mblk data allocation, see the manpage for the relevant transfer function).

The `usb_alloc_isoc_req()` function also allocates a number of isochronous packet descriptors (usb_isoc_pkt_descr_t) specified by isoc_pkts_count to the end of the request proper (usb_isoc_req_t). See `usb_isoc_request(9S)` for more information on isochronous packet descriptors.

These functions always succeed when the USB_FLAGS_SLEEP flag is set, provided that they are given valid args and are not called from interrupt context.

The `usb_free_ctrl_req()`, `usb_free_bulk_req()`, `usb_free_intr_req()`, and `usb_free_isoc_req()` functions free their corresponding request. If the request's data block pointer is non-zero, the data block is also freed. For isoc requests, the array of packet descriptors is freed.

Return Values

For `usb_alloc_ctrl_req()`, `usb_alloc_bulk_req()`, `usb_alloc_intr_req()` and `usb_alloc_isoc_req()`:

On success: returns a pointer to the appropriate usb_xxx_request_t.

On failure: returns `NULL`. Fails because the dip argument is invalid, USB_FLAGS_SLEEP is not set and memory is not available or because USB_FLAGS_SLEEP is set but the call was made in interrupt context.

For `usb_free_ctrl_req()`, `usb_free_bulk_req()`, `usb_free_intr_req()` and `usb_free_isoc_req()`: None.

Context

The allocation routines can always be called from kernel and user context. They may be called from interrupt context only if USB_FLAGS_SLEEP is not specified.

The free routines may be called from kernel, user, and interrupt context.

Examples

```
/* This allocates and initializes an asynchronous control
 * request which will pass no data.  Asynchronous requests
 * are used when they cannot block the calling thread.
 */

usb_ctrl_req_t *ctrl_req;

if ((ctrl_req = usb_alloc_ctrl_req(dip, 0, 0)) == NULL) {
        return (FAILURE);
}
```

```
/* Now initialize. */
ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_DEV_TO_HOST |
    USB_DEV_REQ_STANDARD | USB_DEV_REQ_RCPT_DEV;

ctrl_req->ctrl_bRequest      = (uint8_t)USB_REQ_GET_STATUS;
...
...
ctrl_req->ctrl_callback     = normal_callback;
ctrl_req->ctrl_exc_callback = exception_callback;
...
...
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_get_current_frame_number(9F)`, `usb_get_max_pkts_per_isoc_request(9F)`, `usb_pipe_get_max_bulk_transfer_size(9F)`, `usb_pipe_bulk_xfer(9F)`, `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_intr_xfer(9F)`, `usb_pipe_isoc_xfer(9F)`, `usb_bulk_request(9S)`, `usb_ctrl_request(9S)`, `usb_intr_request(9S)`, `usb_isoc_request(9S)`

Name

usb_client_attach, usb_client_detach – USBA framework registration of client USB drivers

Synopsis

```
#define USBDRV_MAJOR_VER    <major>
#define USBDRV_MINOR_VER    <minor>
#include <sys/usb/usba.h>
```

```
int usb_client_attach(dev_info_t *dip,
    uint_t version, usb_flags_t flags);
```

```
void usb_client_detach(dev_info_t *dip,
    usb_client_dev_data_t *dev_data);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

For `usb_client_attach()`:

dip	Pointer to the device's dev_info structure.
version	Must be set to USBDRV_VERSION. (See below.)
flags	Not used.

For `usb_client_detach()`:

dip	Pointer to the device's dev_info structure.
dev_data	Pointer to a usb_client_dev_data_t to free. Can be NULL.

Description

The `usb_client_attach()` function registers a driver with the USBA framework and must be called before any other USBA function. Usually, `usb_client_attach()` is followed by a call to `usb_get_dev_data`(9F).

The `usb_client_detach()` function unregisters a driver with the USBA framework. The `usb_client_detach()` function releases memory for all strings, descriptors and trees set up by `usb_get_dev_data`(9F) when its dev_data argument is non-NULL. The `usb_client_detach()` function is the last USBA function a client calls before completing `detach`(9E). It is not necessary to call `usb_client_detach()` during a suspend operation.

VERSIONING

USBDRV_VERSION is a macro which creates a version number based on the USBDRV_MAJOR_VER and USBDRV_MINOR_VER definitions. It must be passed as the version argument.

For drivers version 2.0 or greater, the value of USBDRV_MAJOR_VERSION must match its corresponding USBA_MAJOR_VER value in `<sys/usb/usbai.h>`, and the value of USBDRV_MINOR_VERSION must not be greater than its corresponding USBA_MINOR_VER value also in <sys/usb/usbai.h>.

Version 0.8 drivers from previous releases are binary compatible and run on Solaris 10, but are not compilable. Version 0.8 binary compatibility will not be supported in subsequent Solaris OS releases.

Definitions of USBDRV_MAJOR_VERSION and USBDRV_MINOR_VERSION must appear in the client driver above the reference to `<sys/usb/usba.h>`. Note that different releases have different USBA_[MAJOR|MINOR]_VER numbers.

Return Values

For `usb_client_attach()`:

USB_SUCCESS	Registration is successful.
USB_INVALID_ARGS	*dip* is `NULL`.
USB_INVALID_CONTEXT	Called from interrupt context. Not called from an attach routine context.
USB_INVALID_VERSION	Version passed in version is invalid.
USB_FAILURE	Other internal error.

For `usb_client_detach()`:

USB_INVALID_ARGS	*dip* is `NULL`.
USB_INVALID_CONTEXT	Not called from an attach routine context.

Context

The `usb_client_attach()` function may only be called from `attach`(9E).

The `usb_client_detach()` function may be called only from `attach`(9E) or `detach`(9E).

Examples

```
  if (usb_client_attach(dip, USBDRV_VERSION, 0) != USB_SUCCESS) {
            cmn_err (CE_WARN, "%s%d: Couldn't register USB device",
                ddi_driver_name(dip), ddi_get_instance(dip));

           return (USB_FAILURE);
   }

   if (usb_get_dev_data(dip, &dev_data, USB_PARSE_LVL_IF, 0) !=
       USB_SUCCESS) {
           cmn_err (CE_WARN, "%s%d: Couldn't get device descriptor data.",
               ddi_driver_name(dip), ddi_get_instance(dip));

           return (USB_FAILURE);
   }
```

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `attach`(9E), `detach`(9E), `usb_get_dev_data`(9F)

Name usb_clr_feature – Clear feature of USB device, interface or endpoint

Synopsis

```
#include <sys/usb/usba.h>



int usb_clr_feature(dev_info_t *dip,
     uint_t request_type, uint_t feature,
     uint_t which, usb_flags_t flags,
     void (*callback) (usb_pipe_handle_t pipe_handle,
     usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
     usb_opaque_t callback_arg);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's dev_info structure.
pipe_handle	Pipe handle to device, device interface or endpoint.
request_type	bmRequestType to be used. One of the following: USB_DEV_REQ_RCPT_DEV - Clear feature on device. USB_DEV_REQ_RCPT_IF - Clear feature on interface. USB_DEV_REQ_RCPT_EP - Clear feature on endpoint.
feature	Feature to be cleared. Can be any device-defined device-, interface-, or endpoint-specific feature, including the following which are defined in the *USB 2.0* specification: USB_EP_HALT - Clear a HALT on an endpoint. USB_DEV_REMOTE_WAKEUP - Clear REMOTE_WAKEUP on a device. USB_DEV_TEST_MODE - Clear TEST_MODE on a device.
which	Device, interface or endpoint on which to clear the feature. One of: Interface number, for interfaces. Endpoint number, for endpoints. 0 for devices.
flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for completion and do not call callback.
callback	Callback handler to notify of asynchronous completion.
callback_arg	Second argument passed to callback handler.

Description

The `usb_clr_feature()` function clears a specific feature of a device, interface or endpoint. This function always blocks and waits for resources if not available, regardless of the flags argument.

This call blocks for completion if USB_FLAGS_SLEEP is set in flags. It returns immediately and calls the callback upon completion if USB_FLAGS_SLEEP is not set.

Return Values

USB_SUCCESS	Feature was successfully cleared.
USB_INVALID_ARGS	*dip* argument is NULL.
USB_INVALID_PIPE	*pipe_handle* argument is NULL
USB_INVALID_CONTEXT	Called from interrupt context with USB_FLAGS_SLEEP flag set.
USB_FAILURE	Clearing of feature was unsuccessful.

Context

May always be called from user or kernel context. May be called from interrupt context only if USB_FLAGS_SLEEP is not set in flags.

If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags(9S)` for more information on callbacks.

Examples

```
if (usb_clr_feature(dip, pipe_handle, USB_DEV_REQ_RCPT_EP,
       USB_EP_HALT, data_endpoint_num, 0) == USB_FAILURE) {
           cmn_err (CE_WARN,
               "%s%d: Error clearing halt condition on data endpoint %d.",
               ddi_driver_name(dip), ddi_get_instance(dip),
               data_endpoint_num);
   }
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_get_status(9F)`, `usb_pipe_reset(9F)`, `usb_pipe_get_state(9F)`, `usb_callback_flags(9S)`

Name usb_create_pm_components – Create power management components for USB devices

Synopsis

```
#include <sys/usb/usba.h>



int usb_create_pm_components(dev_info_t *dip, uint_t *pwrstates);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
pwrstates	Address into which a mask which lists power states capable by device is returned. This is a bitmask containing zero or more of the following values:

USB_DEV_PWRMASK_D0	Corresponds to USB_DEV_OS_PWR_3 or full power.
USB_DEV_PWRMASK_D1	Corresponds to USB_DEV_OS_PWR_2.
USB_DEV_PWRMASK_D2	Corresponds to USB_DEV_OS_PWR_1.
USB_DEV_PWRMASK_D3	Corresponds to USB_DEV_OS_PWR_0 or no power.

Description The `usb_create_pm_components()` function creates pm component properties that assume the standard USB D0-D3 powerlevels (USB_DEV_PWR_D0 - USB_DEV_PWR_D3). See the device's relevant USB descriptor to determine the device's power management capabilities and account for bus-powered devices. The `usb_create_pm_components()` function also updates the pm-components property in the device's dev_info structure.

Note that these USB power levels are inverse of OS power levels. For example, USB_DEV_OS_PWR_0 and USB_DEV_PWR_D3 are equivalent levels corresponding to powered-down.

Return Values

USB_SUCCESS	Power management facilities in device are recognized by system.
USB_FAILURE	An error occurred.

Context May be called from user or kernel context.

Examples

```
   uint_t *pwrstates;

    /* Hook into device’s power management.  Enable remote wakeup. */
    if (usb_create_pm_components(dip, pwrstates) == USB_SUCCESS) {
            usb_handle_remote_wakeup(dip,USB_REMOTE_WAKEUP_ENABLE);
    }
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `usb_clr_feature`(9F), `usb_register_hotplug_cbs`(9F), `usb_get_cfg`(9F), `usb_get_dev_data`(9F), `usb_handle_remote_wakeup`(9F), `pm_idle_component`(9F), `pm_busy_component`(9F), `pm_raise_power`(9F), `pm_lower_power`(9F), `usb_cfg_descr`(9S)

Name usb_get_addr – Retrieve device USB address

Synopsis

```
#include <sys/usb/usba.h>

int usb_get_addr(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's dev_info structure.

Description The `usb_get_addr()` function returns the current USB bus address for debugging purposes. The returned address is unique for a specific USB bus, and may be replicated if multiple host controller instances are present on the system.

Return Values On success: USB device address.

On failure: returns 0. Fails if dip is NULL.

Context May be called from user, kernel or interrupt context.

Examples

```
int usb_addr;

usb_addr = usb_get_addr(dip);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_pipe_open(9F)`

Name usb_get_alt_if, usb_set_alt_if, usb_get_if_number, usb_owns_device – Get and set alternate interface values

Synopsis

```
#include <sys/usb/usba.h>



int usb_get_alt_if(dev_info_t *dip, uint_t interface_number,
     uint_t *alternate_number, usb_flags_t flags);

int usb_set_alt_if(dev_info_t *dip, uint_t interface_number,
     uint_t *alternate_number, usb_flags_t flags,
     void (*callback)(usb_pipe_handle_t pipe_handle,
     usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
     usb_opaque_t callback_arg);



int usb_get_if_number(dev_info_t *dip);



boolean_t usb_owns_device(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_get_alt_if()`:

dip	Pointer to device's dev_info structure.
interface_number	Interface of the desired alternate.
alternate_number	Address where current alternate setting is returned.
flags	No flags are recognized. Reserved for future expansion.

For `usb_set_alt_if()`:

dip	Pointer to device's dev_info structure.
interface_number	Interface of the desired alternate.
alternate_number	Alternate interface number to be set.
flags	Only USB_FLAGS_SLEEP is recognized. Wait for completion and do not call callback.
callback	Callback handler to notify of asynchronous completion.
callback_arg	Second argument passed to callback handler.

For `usb_get_if_number()`:

dip	Pointer to device's dev_info structure.

For `usb_owns_device()`:

dip	Pointer to device's dev_info structure.

Description

USB devices can have multiple configurations, each with many interfaces. Within interfaces are alternate settings, and within alternate settings are endpoints.

Each interface within a configuration may be represented by the kernel as a device node. Only one set of device nodes (interfaces as determined by the configuration) can be active at one time.

Alternates to an interface represent different ways the kernel sees a device node. Only one alternate setting within an interface can be active (or selected) at one time. The functions presented in this manpage get or set interface or alternate setting information.

The `usb_get_alt_if()` function requests the device to return the current alternate setting of the given interface. This function ignores the flags argument and always blocks.

The `usb_set_alt_if()` function requests the device to set the interface and its alternate setting as specified. Because this call changes the current device's interface and sets the new interface's mode of operation as seen by the system, the driver must insure that all pipes other than the default control pipe are closed and quiescent. To avoid contending with another driver for a different part of the device, the driver must be bound to the entire device.

If USB_FLAGS_SLEEP is set in flags, `usb_set_alt_if()` blocks until completed. Otherwise, `usb_set_alt_if()` returns immediately and calls the callback handler when completed.

callback is the asynchronous callback handler and takes the following arguments:

usb_pipe_handle_t pipe_handle	Handle of the default control pipe used to perform the request.
usb_opaque_t callback_arg	Callback_arg specified to `usb_set_alt_if()`.
int rval	Request status.
usb_cb_flags_t callback_flags:	Status of the queueing operation. Can be: USB_CB_NO_INFO - Callback was uneventful. USB_CB_ASYNC_REQ_FAILED - Error queueing request. USB_CB_NO_RESOURCES - Error allocating resources.

The `usb_get_if_number()` function returns the interface number, or USB_COMBINED_NODE or USB_DEVICE_NODE node indicating that the driver is bound to the entire device. (See Return Values below.)

The `usb_owns_device()` function returns B_TRUE if the driver of the dip argument owns the entire device, or B_FALSE if it owns just a particular interface.

Return Values For `usb_get_alt_if()`:

USB_SUCCESS	Interface's alternate setting was successfully obtained.
USB_INVALID_ARGS	Pointer to alternate_number and/or dip are NULL.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_FAILURE	The interface number is invalid. An access error occurred.

For `usb_set_alt_if()`:

USB_SUCCESS	Alternate interface was successfully set.
USB_INVALID_ARGS	dip is NULL. USB_FLAGS_SLEEP is clear and callback is NULL.
USB_INVALID_PERM	dip does not own the interface to be set.
USB_INVALID_CONTEXT	Called from interrupt context with USB_FLAGS_SLEEP specified.
USB_INVALID_PIPE	Pipe handle is NULL, invalid, or refers to a pipe that is closing or closed.
USB_FAILURE	The interface number and/or alternate setting are invalid. Pipes were open. An access error occurred.

For `usb_get_if_number()`:

USB_COMBINED_NODE if the driver is responsible for the entire active device configuration. The dip doesn't correspond to an entire physical device.

USB_DEVICE_NODE if the driver is responsible for the entire device. The dip corresponds to an entire physical device.

interface number: otherwise.

For `usb_owns_device()`:

B_TRUE	Driver of the dip argument owns the entire device.
B_FALSE	Driver of the dip argument owns only the current interface.

Context The `usb_get_if_number()` and `usb_owns_device()` functions may be called from user or kernel context.

The `usb_set_alt_if()` function may always be called from user or kernel context. It may be called from interrupt context only if USB_FLAGS_SLEEP is not set in flags. If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags`(9S) for more information on callbacks.

The `usb_get_alt_if()` function may be called from user or kernel context.

Examples

```
/* Change alternate setting of interface 0.  Wait for completion. */
if (usb_set_alt_if(
    dip, 0, new_alternate_setting_num, USB_FLAGS_SLEEP, NULL, 0) !=
    USB_SUCCESS) {
        cmn_err (CE_WARN,
            "%s%d: Error setting alternate setting on pipe",
            ddi_driver_name(dip), ddi_get_instance(dip));
        }
}
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `usb_pipe_ctrl_xfer`(9F), `usb_get_dev_data`(9F), `usb_get_string_descr`(9F), `usb_get_cfg`(9F)

Name usb_get_cfg, usb_set_cfg – Get and set current USB device configuration

Synopsis

```
#include <sys/usb/usba.h>



int usb_get_cfg(dev_info_t *dip, uint_t cfgval, usb_flags_t flags);

int usb_set_cfg(dev_info_t *dip, uint_t cfg_index, usb_flags_t flags,
     void (*callback)(usb_pipe_handle_t pipe_handle, usb_opaque_t callback_arg,
     int  rval,  usb_cb_flags_t flags), usb_opaque_t callback_arg);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_get_cfg()`:

dip	Pointer to device's dev_info structure.
cfgval	Pointer to returned configuration value.
flags	Not used. Always waits for completion.

For `usb_set_cfg()`:

dip	Pointer to device's dev_info structure.
cfg_index	Desired device configuration index. Set to USB_DEV_DEFAULT_CONFIG_INDEX to restore default configuration.
flags	Only USB_FLAGS_SLEEP is recognized. Wait for completion and do not call callback.
callback	Callback handler to notify of asynchronous completion.
callback_arg	Second argument passed to callback handler.

Description The `usb_get_cfg()` function retrieves the current configuration. It ignores the flags argument and always blocks while contacting the device.

The `usb_set_cfg()` function sets a new configuration. Because this call changes the device's mode of operation, the device must be quiescent and have all pipes, with the exception of the default control pipe, closed. The driver must have control over the entire device and cannot own just a single interface on a composite device. Additionally, its device node must not be a parent to other device nodes that can be operated by other drivers. The driver must own the device exclusively, otherwise drivers managing other parts of the device would be affected without their knowledge or control.

This call updates all internal USBA framework data structures, whereas issuing a raw USB_REQ_SET_CFG device request does not. The `usb_set_cfg()` function is the only supported programmatic way to change device configuration.

This call blocks if USB_FLAGS_SLEEP is set in flags. It returns immediately and calls the callback on completion if USB_FLAGS_SLEEP is not set.

Return Values For `usb_get_cfg()`:

USB_SUCCESS	New configuration is retrieved.
USB_INVALID_ARGS	*cfgval* or *dip* is NULL.
USB_FAILURE	Configuration cannot be retrieved.

For `usb_set_cfg()`:

USB_SUCCESS	New configuration is set.
USB_INVALID_ARGS	dip is NULL. USB_FLAGS_SLEEP is clear and callback is NULL.
USB_INVALID_CONTEXT	Called from interrupt context with USB_FLAGS_SLEEP specified.
USB_INVALID_PERM	Caller does not own entire device or device is a parent to child devices.
USB_BUSY	One or more pipes other than the default control pipe are open on the device.
USB_INVALID_PIPE	Pipe handle is NULL or invalid, or pipe is closing or closed.
USB_FAILURE	An illegal configuration is specified. One or more pipes other than the default control pipe are open on the device.

Context The `usb_get_cfg()` function may be called from user or kernel context.

The `usb_set_cfg()` function may be called from user or kernel context always. It may be called from interrupt context only if USB_FLAGS_SLEEP is not set in flags.

If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags`(9S) for more information on callbacks.

Examples

```
Setting the configuration to the one at index 1 (in the
array of usb_cfg_data_t configuration nodes as returned
by usb_get_dev_data()), and verifying what the configuration
is at that index. (See usb_get_dev_data(9F)).

uint_t cfg_index = 1;
```

```
/*
 * Assume all pipes other than the default control pipe
 * are closed and make sure all requests to the default
 * control pipe have completed. /
 */


if (usb_set_cfg(dip, cfg_index, USB_FLAGS_SLEEP, NULL, 0) != USB_SUCCESS) {
        cmn_err (CE_WARN,
            "%s%d: Error setting USB device to configuration #%d",
            ddi_driver_name(dip), ddi_get_instance(dip), cfg_index);
}

if (usb_get_cfg(dip, &bConfigurationValue, 0) == USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: USB device active configuration is %d",
            ddi_driver_name(dip), ddi_get_instance(dip),
            bConfigurationValue);
} else {
        ...
        ...
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_get_alt_if(9F)`, `usb_get_dev_data(9F)`, `usb_get_string_descr(9F)`, `usb_pipe_open(9F)`, `usb_callback_flags(9S)`, `usb_cfg_descr(9S)`, `usb_ep_descr(9S)`, `usb_if_descr(9S)`

Name usb_get_current_frame_number – Return current logical usb frame number

Synopsis

```
#include <sys/usb/usba.h>



usb_frame_number_t usb_get_current_frame_number(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.

Description The `usb_get_current_frame_number()` function retrieves the current logical USB frame number.

Isochronous requests can be started on a particular numbered frame. An offset number of frames (typically between 4 and 10) can be added to the current logical frame number to specify the number of an upcoming frame to start an isochronous request.

The USB specification requires that the frame frequency (the period between start-of-frame packets) is one millisecond. The Solaris operating environment USB implementation uses a running counter of the number of milliseconds since boot as the current logical frame number.

Return Values On success, the `usb_get_current_frame_number()` function returns the current USB frame number. On failure it returns `0`. The function fails if *dip* is NULL.

Context May be called from user, kernel or interrupt context.

Examples

```
usb_pipe_handle_t handle;
usb_frame_number_t offset = 10;
usb_isoc_req_t *isoc_req;

isoc_req = usb_alloc_isoc_req(...);
  ...
  ...
isoc_req->isoc_frame_no = usb_get_current_frame_number(dip) + offset;
isoc_req->isoc_attributes = USB_ATTRS_ISOC_START_FRAME;
  ...
  ...
if (usb_pipe_isoc_xfer(handle, isoc_req, 0) != USB_SUCCESS) {
  ...
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `usb_alloc_isoc_req`(9F), `usb_get_max_pkts_per_isoc_request`(9F), `usb_pipe_isoc_xfer`(9F), `usb_pipe_get_max_bulk_transfer_size`(9F), `usb_isoc_request`(9S)

Name usb_get_dev_data, usb_free_dev_data, usb_free_descr_tree, usb_print_descr_tree – Retrieve device configuration information

Synopsis `#include <sys/usb/usba.h>`

```
int usb_get_dev_data(dev_info_t *dip, usb_client_dev_data_t **dev_data,
     usb_reg_parse_lvl_t parse_level, usb_flags_t flags);

void usb_free_dev_data(dev_info_t *dip, usb_client_dev_data_t *dev_data);

void usb_free_descr_tree(dev_info_t *dip, usb_client_dev_data_t *dev_data);

int usb_print_descr_tree(dev_info_t *dip, usb_client_dev_data_t *dev_data);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_get_dev_data()`:

dip	Pointer to device's dev_info structure.
dev_data	Address in which pointer to info is returned.
parse_level	Portion of device represented in the tree of parsed descriptors. See below for possible usb_reg_parse_lvl_t values and explanations.
flags	Not used.

For `usb_free_dev_data()`:

dip	Pointer to device's dev_info structure.
dev_data	Pointer to usb_client_dev_data_t to be freed.

For `usb_free_descr_tree()`:

dip	Pointer to device's dev_info structure.
dev_data	Pointer to usb_client_dev_data_t containing the descriptor tree to free.

For `usb_print_descr_tree()`:

dip	Pointer to device's dev_info structure.
dev_data	Pointer to usb_client_dev_data_t containing the descriptor tree to display on-screen.

Description The `usb_get_dev_data()` function interrogates a device and returns its configuration information in a usb_client_dev_data_t structure. Most USBA functions require information which comes from a usb_client_dev_data_t, and all other functions in this man page operate on this structure. Please see `usb_client_dev_data(9S)` for a full content description. Pass the usb_client_dev_data_t structure to `usb_client_detach(9F)` to completely deallocate it.

A descriptor tree is included in the information returned. The usb_reg_parse_lvl_t type represents the extent of the device to be represented by the returned tree (2nd arg to usb_get_dev_data) or what is actually represented in the returned tree (dev_parse_level field of the returned usb_client_dev_data_t). It has the following possible values:

USB_PARSE_LVL_NONE	Build no tree. dev_n_cfg returns 0, dev_cfg and dev_curr_cfg are returned NULL, and the dev_curr_xxx fields are invalid.
USB_PARSE_LVL_IF	If configuration number and interface properties are set (as when different interfaces are viewed by the OS as different device instances), parse configured interface only. If an OS device instance is set up to represent an entire physical device, USB_PARSE_LVL_IF works like USB_PARSE_LVL_ALL.
USB_PARSE_LVL_CFG	Parse entire configuration of configured interface only. Behaves similarly to USB_PARSE_LVL_IF, except that entire configuration is returned.
USB_PARSE_LVL_ALL	Parse entire device (all configurations), even when driver is bound to a single interface of a single configuration.

The `usb_free_dev_data()` function undoes what `usb_get_dev_data()` set up. It releases memory for all strings, descriptors, and trees set up by `usb_get_dev_data()`.

The `usb_free_descr_tree()` function frees the descriptor tree of its usb_client_dev_data_t argument, while leaving the rest of the information intact. The intent is for drivers to free memory after copying needed descriptor information from the tree. Upon return, the following usb_client_dev_data_t fields are modified as follows: dev_cfg is NULL, dev_n_cfg is zero and dev_parse_level is USB_PARSE_LVL_NONE. Additionally, dev_curr_cfg is NULL and dev_curr_if is invalid.

The `usb_print_descr_tree()` function is an easy-to-use diagnostic aid which dumps the descriptor tree to the screen when the system is verbose booted (`boot -v`). Output is spaced with blank lines for readability and provides you with an on-screen look at what a device has to offer.

Return Values For `usb_get_dev_data()`:

USB_SUCCESS	Registration is successful.
USB_INVALID_ARGS	*dip* or *dev_data* is `NULL`. *parse_level* is invalid.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_INVALID_VERSION	`usb_client_attach`(9F) was not called first.
USB_FAILURE	Bad descriptor info or other internal error.

For `usb_free_dev_data()`: None

For `usb_free_descr_tree()`: None, but no operation occurs if *dip* and/or *dev_data* are NULL.

For `usb_print_descr_tree()`:

USB_SUCCESS	Descriptor tree dump is successful.
USB_INVALID_ARGS	*dev_data* or *dip* are NULL.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_FAILURE	Other error.

Context The `usb_get_dev_data()` and `usb_print_descr_tree()` functions may be called from user or kernel context.

The `usb_free_dev_data()` and `usb_free_descr_tree()` functions may be called from user, kernel or interrupt context.

Examples

```
In this example, assume a device has the configuration shown
below, and the endpoint of config 2, iface 1, alt 1
which supports intr IN transfers needs to be found.
Config 2, iface 1 is the "default" config/iface for the
current OS device node.

      config 1
          iface 0
              endpt 0
      config 2
          iface 0
          iface 1
              alt 0
                  endpt 0
                      cv 0
              alt 1
                  endpt 0
                  endpt 1
                      cv 0
                  endpt 2
              alt 2
                  endpt 0
                      cv 0

      usb_client_dev_data_t *dev_data;
      usb_ep_descr_t ep_descr;
      usb_ep_data_t *ep_tree_node;
      uint8_t interface = 1;
      uint8_t alternate = 1;
      uint8_t first_ep_number = 0;
```

```
/*
 * We want default config/iface, so specify USB__PARSE_LVL_IF.
 * Default config will be returned as dev_cfg[0].
/
  if (usb_get_dev_data(dip, &dev_data,
    USB_PARSE_LVL_IF, 0) != USB_SUCCESS) {
        cmn_err (CE_WARN,
            "%s%d: Couldn't get USB configuration descr tree",
            ddi_driver_name(dip), ddi_get_instance(dip));

        return (USB_FAILURE);
}

ep_tree_node = usb_lookup_ep_data(dip, dev_data, interface,
     alternate, first_ep_number, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
if (ep_tree_node != NULL) {
    ep_descr = ep_tree_node->ep_descr;
} else {
        cmn_r (CE_WARN,
            "%s%d: Device is missing intr-IN endpoint",
            ddi_driver_name(dip), ddi_get_instance(dip));

        usb_free_descr_tree(dip, &dev_data);

        return (USB_FAILURE);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_client_attach(9F)`, `usb_get_alt_if(9F)`, `usb_get_cfg(9F)`, `usb_get_string_descr(9F)`, `usb_lookup_ep_data(9F)`, `usb_parse_data(9F)`, `usb_pipe_open(9F)`, `usb_cfg_descr(9S)`, `usb_client_dev_data(9S)`, `usb_ep_descr(9S)`, `usb_if_descr(9S)`, `usb_string_descr(9S)`

Name usb_get_max_pkts_per_isoc_request – Get maximum number of packets allowed per isochronous request

Synopsis

```
#include <sys/usb/usba.h>



uint_t usb_get_max_pkts_per_isoc_request(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters *dip* Pointer to the device's `dev_info` structure.

Description The `usb_get_max_pkts_per_isoc_request()` function returns the maximum number of isochronous packets per request that the host control driver can support. This number can be used to determine the maximum amount of data which can be handled by a single isochronous request. That length is found by:

```
max = usb_get_max_pkts_per_isoc_request(dip) * endpoint_max_packet_size;
```

where endpoint_max_packet_size is the wMaxPacketSize field of the isochronous endpoint over which the transfer will take place.

Return Values On success, the `usb_get_current_frame_number()` function returns the maximum number of isochronous pkts per request. On failure it returns `0`. The function fails if *dip* is NULL.

Context May be called from user, kernel or interrupt context.

Examples

```
/*
 * Set up to receive periodic isochronous data, requesting
 * the maximum amount for each transfer.
 */

int pkt;
/* Get max packet size from endpoint descriptor. */
uint_t ep_max_pkt_size = ep_descr.wMaxPacketSize;
uint_t isoc_pkts_count = usb_get_max_pkts_per_isoc_request(dip);

/*
 * Allocate an isoc request, specifying the max number of packets
 * and the greatest size transfer possible.
 */
usb_isoc_req_t *isoc_req = usb_alloc_isoc_req(dip,
    isoc_pkts_count,
    isoc_pkts_count * ep_max_pkt_size,
    USB_FLAGS_SLEEP);

/* Init each packet descriptor for maximum size. */
for (pkt = 0; pkt < isoc_pkts_count; pkt++) {
```

```
    isoc_req->isoc_pkt_descr[pkt].isoc_pkt_length = ep_max_pkt_size;

}


/* Set the length of a packet in the request too. */
isoc_req->isoc_pkts_length = ep_max_pkt_size;

/* Other isoc request initialization. */
...
...

if (usb_pipe_isoc_xfer(pipe, isoc_req, USB_FLAGS_NOSLEEP) != USB_SUCCESS) {
    ...
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_pipe_isoc_xfer(9F)`, `usb_alloc_request(9F)`, `usb_get_current_frame_number(9F)`. `usb_ep_descr(9S)`, `usb_isoc_request(9S)`

Name usb_get_status – Get status of a USB device/endpoint/interface

Synopsis

```
#include <sys/usb/usba.h>



int usb_get_status(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
     uint_t request_type, uint_t which, uint16_t *status,
     usb_flags_t flags);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to device's dev_info structure.
pipe_handle	Default control pipe handle on which request is made.
request_type	bmRequestType. Either: USB_DEV_REQ_RCPT_DEV — Get device status. USB_DEV_REQ_RCPT_IF — Get interface status. USB_DEV_REQ_RCPT_EP — Get endpoint status.
which	Device, interface or endpoint from which to get status. Either number of interface or endpoint, or 0 if device status requested.
status	Address into which the status is written.
flags	None are recognized.

Description The `usb_get_status()` function returns the status of a device, interface or endpoint. All status requests use the default control pipe. Length of data returned is USB_GET_STATUS_LEN bytes. Always block and wait for resources if not available, regardless of the flags argument.

When the *request_type* recipient is USB_DEV_REQ_RCPT_DEV, device status is requested. Status returned includes bits for USB_DEV_SLF_PWRD_STATUS (device is currently self-powered) and USB_DEV_RWAKEUP_STATUS (device has remote wakeup enabled). A set bit indicates the corresponding status.

When the *request_type* is USB_DEV_REQ_RCPT_EP, endpoint status is requested. Status returned includes bits for USB_EP_HALT_STATUS (endpoint is halted). A set bit indicates the corresponding status.

When the *request_type* is USB_DEV_REQ_RCPT_IF, interface status is requested and USB_IF_STATUS (zero) is returned.

Return Values

USB_SUCCESS	Status returned successfully in the status argument.
USB_INVALID_ARGS	Status pointer and/or dip argument is NULL.
USB_INVALID_PIPE	Pipe handle is NULL.
USB_FAILURE	Status not returned successfully.

Context May be called from user or kernel context.

Examples

```
uint16_t status;

if (usb_get_status(
    dip, pipe_handle, USB_DEV_REQ_RCPT_DEV, 0 &status, 0) ==
    USB_SUCCESS) {
        if (status & USB_DEV_SLF_PWRD_STATUS) {
                cmn_err (CE_WARN,
                    "%s%d: USB device is running on its own power.",
                    ddi_driver_name(dip), ddi_get_instance(dip));
        }
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also attributes(5), usb_clr_feature(9F), usb_get_alt_if(9F), usb_pipe_get_state(9F), usb_get_cfg(9F),

Name usb_get_string_descr – Get string descriptor from device

Synopsis

```
#include <sys/usb/usba.h>



int usb_get_string_descr(dev_info_t *dip,
     uint16_t langid, uint8_t index,
     char *buf, size_t buflen);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
langid	Language ID. Currently only USB_LANG_ID (English ascii) is valid.
index	String index indicating descriptor to retrieve.
buf	Address into which the string descriptor is placed.
buflen	Size of buf in bytes.

Description The `usb_get_string_descr()` function retrieves a parsed string descriptor from a device. *dip* specifies the device, while *index* indicates which descriptor to return.

String descriptors provide information about other descriptors, or information that is encoded in other descriptors, in readable form. Many descriptor types have one or more index fields which identify string descriptors. (See Sections *9.5* and *9.6* of the *USB 2.0* specification.) For example, a configuration descriptor's seventh byte contains the string descriptor index describing a specific configuration.

Retrieved descriptors that do not fit into *buflen* bytes are truncated. All returned descriptors are null-terminated.

Return Values

USB_SUCCESS	String descriptor is returned in *buf*.
USB_INVALID_ARGS	*dip* or *buf* are `NULL`, or *index* or *buflen* is `0`.
USB_FAILURE	Descriptor cannot be retrieved.

Context May be called from user or kernel context.

Examples

```
   /* Get the first string descriptor. */

    char buf[SIZE];

    if (usb_get_string_descr(
        dip, USB_LANG_ID, 0, buf, SIZE) == USB_SUCCESS) {
            cmn_err (CE_NOTE, "%s%d: %s",
                ddi_driver_name(dip), ddi_get_instance(dip), buf);
    }
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_get_dev_data(9F)`, `usb_string_descr(9S)`

Name usb_handle_remote_wakeup – Enable or disable remote wakeup on USB devices

Synopsis

```
#include <sys/usb/usba.h>



int usb_handle_remote_wakeup(dev_info_t *dip, int cmd);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
cmd	Command. Either USB_REMOTE_WAKEUP_ENABLE or USB_REMOTE_WAKEUP_DISABLE.

Description The `usb_handle_remote_wakeup()` function enables or disables remote wakeup on a USB device. This call can block.

Return Values

USB_SUCCESS	Remote wakeup is successfully enabled or disabled.
USB_FAILURE	Remote wakeup is not supported by the device. An internal error occurred.

Context May be called from user or kernel context.

Examples

```
uint_t *pwrstates;

/* Hook into device’s power management.  Enable remote wakeup. */
if (usb_create_pm_components(dip, pwrstates) == USB_SUCCESS) {
        usb_handle_remote_wakeup(dip, USB_REMOTE_WAKEUP_ENABLE);
}
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `pm_busy_component`(9F), `pm_idle_component`(9F), `pm_lower_power`(9F), `pm_raise_power`(9F), `usb_clr_feature`(9F), `usb_create_pm_components`(9F), `usb_get_cfg`(9F), `usb_get_dev_data`(9F), `usb_register_hotplug_cbs`(9F), `usb_cfg_descr`(9S)

Name

usb_lookup_ep_data – Lookup endpoint information

Synopsis

```
#include <sys/usb/usba.h>



usb_ep_data_t *usb_lookup_ep_data(dev_info_t *dip,
     usb_client_dev_data_t *dev_datap, uint_t interface,
     uint_t alternate, uint_t skip, uint_t type, uint_t direction);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
dev_datap	Pointer to a `usb_client_dev_data_t` structure containing tree.
interface	Number of interface in which endpoint resides.
alternate	Number of interface alternate setting in which endpoint resides.
skip	Number of endpoints which match the requested type and direction to skip before finding one to retrieve.
type	Type of endpoint. This is one of: USB_EP_ATTR_CONTROL, USB_EP_ATTR_ISOCH, USB_EP_ATTR_BULK, or USB_EP_ATTR_INTR. Please see `usb_pipe_open`(9F) for more information.
direction	Direction of endpoint, either USB_EP_DIR_OUT or USB_EP_DIR_IN. This argument is ignored for bi-directional control endpoints.

Description

The `usb_lookup_ep_data()` function returns endpoint information from the tree embedded in client data returned from `usb_get_dev_data`. It operates on the current configuration (pointed to by the dev_curr_cfg field of the `usb_client_dev_data_t` argument). It skips the first <skip> number of endpoints it finds which match the specifications of the other arguments, and then retrieves information on the next matching endpoint it finds. Note that it does not make a copy of the data, but points to the tree itself.

Return Values

On success: the tree node corresponding to the desired endpoint.

On failure: returns NULL. Fails if *dip* or *dev_datap* are NULL, if the desired endpoint does not exist in the tree, or no tree is present in dev_datap.

Context

May be called from user, kernel or interrupt context.

Examples

Retrieve the polling interval for the second interrupt endpoint at interface 0, alt 3:

```
uint8_t interval = 0;
usb_ep_data_t *ep_node = usb_lookup_ep_data(
    dip, dev_datap, 0, 3, 1, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
if (ep_node != NULL) {
```

```
        interval = ep_node->ep_descr.bInterval;
}
```

Retrieve the maximum packet size for the first control pipe at interface 0, alt 4:

```
uint16_t maxPacketSize = 0;
usb_ep_data_t *ep_node = usb_lookup_ep_data(
    dip, dev_datap, 0, 4, 0, USB_EP_ATTR_CONTROL, 0);
if (ep_node != NULL) {
        maxPacketSize = ep_node->ep_descr.wMaxPacketSize;
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_get_dev_data(9F)`, `usb_pipe_open(9F)`, `usb_cfg_descr(9S)`, `usb_if_descr(9S)`, `usb_ep_descr(9S)`

Name

usb_parse_data – Tokenize and align the bytes of raw variable-format data

Synopsis

```
#include <sys/usb/usba.h>



size_t usb_parse_data(char *format, uchar_t  *data,
     size_t datalen, void *structure, size_t structlen);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

format	Null terminated string describing the format of the data structure for general-purpose byte swapping. The letters "c," "s," "l," and "L" represent 1, 2, 4 and 8 byte quantities, respectively. A descriptor that consists of a short and two bytes would be described by "scc." A number preceding a letter serves as a multiplier of that letter. A format equivalent to "scc" is "s2c."
data	Raw descriptor data to parse.
datalen	Length, in bytes, of the raw descriptor data buffer.
structure	Destination data buffer where parsed data is returned.
structlen	Length, in bytes, of the destination data buffer. Parsed result length will not exceed this value.

Description

The `usb_parse_data` function parses data such as a variable-format class- or vendor-specific descriptor. The function also tokenizes and aligns the bytes of raw descriptor data into fields of a variable-format descriptor.

While the USBA framework can parse the endpoint, interface, configuration, and string descriptors defined by the *USB 2.0* specification, the format of class- or vendor-specific descriptors cannot be explicitly defined by the specification and will be unique for each. The *format* argument defines how to parse such a descriptor.

While the USB specification defines bit ordering as little-endian, this routine (like the entire API), converts the data to the endianness of the host.

The *structlen* parameter defines the size of the destination data buffer. Data is truncated to this size if the destination data buffer is too small.

Return Values

On success: Returns the size (in bytes) of the parsed data result.

On failure: Returns 0. (Same as USB_PARSE_ERROR).

Context

May be called from user, kernel or interrupt context.

Examples

```
  /*
   * Parse raw descriptor data in buf, putting result into ret_descr.
   * ret_buf_len holds the size of ret_descr buf;  routine returns
```

```
 * number of resulting bytes.
 *
 * Descriptor being parsed has 2 chars, followed by one short,
 * 3 chars and one more short.
 */
size_t size_of_returned_descr;
xxx_descr_t ret_descr;

 size_of_returned_descr = usb_parse_data("ccscccs",
    buf, sizeof(buf), (void *)ret_descr, (sizeof)xxx_descr_t));
if (size_of_returned_descr < (sizeof (xxx_descr_t))) {
         /* Data truncated. */
}

or:

size_of_returned_descr = usb_parse_data("2cs3cs",
    buf, sizeof(buf), (void *)ret_descr, (sizeof)xxx_descr_t));
if (size_of_returned_descr < (sizeof (xxx_descr_t))) {
        /* Data truncated. */
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also attributes(5), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_get_cfg(9F)

Name usb_pipe_bulk_xfer – USB bulk transfer function

Synopsis

```
#include <sys/usb/usba.h>

int usb_pipe_bulk_xfer(usb_pipe_handle_t pipe_handle,
     usb_bulk_req_t *request, usb_flags_t flags);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

pipe_handle
Bulk pipe handle on which request is made.

request
Pointer to bulk transfer request.

flags
USB_FLAGS_SLEEP is the only flag recognized. Wait for request to complete.

Description The `usb_pipe_bulk_xfer()` function requests the USBA framework to perform a transfer through a USB bulk pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request. Requests are synchronous when USB_FLAGS_SLEEP has been specified in flags. Calls for synchronous requests will not return until their transaction has completed. Asynchronous requests (made without specifying the USB_FLAGS_SLEEP flag) notify the caller of their completion via a callback function.

Requests for bulk transfers must have mblks attached to store data. Allocate an mblk for data when a request is allocated via `usb_alloc_bulk_req`(9F) by passing a positive value for the *len* argument.

Return Values

USB_SUCCESS
Transfer was successful.

USB_INVALID_ARGS
Request is `NULL`.

USB_INVALID_CONTEXT
Called from interrupt context with the USB_FLAGS_SLEEP flag set.

USB_INVALID_REQUEST
The request has been freed or otherwise invalidated.

A set of conflicting attributes were specified. See `usb_bulk_request`(9S).

The normal and/or exception callback was NULL and USB_FLAGS_SLEEP was not set.

Data space is not provided to a bulk request:

```
(bulk_data = NULL or bulk_len = 0)
```

USB_INVALID_PIPE
Pipe handle is NULL or invalid.

Pipe is closing or closed.

USB_PIPE_ERROR
Pipe handle refers to a pipe which is in the USB_PIPE_STATE_ERROR state.

USB_NO_RESOURCES
Memory, descriptors or other resources are unavailable.

USB_HC_HARDWARE_ERROR
Host controller is in error state.

USB_FAILURE
An asynchronous transfer failed or an internal error occurred.

A bulk request requested too much data:

```
(length > usb_get_max_bulk_xfer size())
```

The pipe is in a unsuitable state (error, busy, not ready).

Additional status information may be available in the bulk_completion_reason and bulk_cb_flags fields of the request. Please see `usb_completion_reason`(9S) and `usb_callback_flags`(9S) for more information.

Context May be called from kernel or user context without regard to arguments. May be called from interrupt context only when the USB_FLAGS_SLEEP flag is clear.

Examples

```
/* Allocate, initialize and issue a synchronous bulk request. */

 usb_bulk_req_t bulk_req;
 mblk_t *mblk;

 bulk_req = usb_alloc_bulk_req(dip, bp->b_bcount, USB_FLAGS_SLEEP);

 bulk_req->bulk_attributes   = USB_ATTRS_AUTOCLEARING;
 mblk = bulk_req->bulk_data;
 bcopy(buffer, mblk->b_wptr, bp->b_bcount);
 mblk->b_wptr += bp->b_bcount;

 if ((rval = usb_pipe_bulk_xfer(pipe, bulk_req, USB_FLAGS_SLEEP))
     != USB_SUCCESS) {
         cmn_err (CE_WARN, "%s%d: Error writing bulk data.",
             ddi_driver_name(dip), ddi_get_instance(dip));
 }
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `usb_alloc_request`(9F), `usb_get_cfg`(9F), `usb_get_status`(9F), `usb_pipe_ctrl_xfer`(9F), `usb_pipe_get_state`(9F), `usb_pipe_intr_xfer`(9F), `usb_pipe_isoc_xfer`(9F), `usb_pipe_open`(9F), `usb_pipe_reset`(9F), `usb_bulk_request`(9S), `usb_callback_flags`(9S), `usb_completion_reason`(9S), `usb_ctrl_request`(9S), `usb_intr_request`(9S), `usb_isoc_request`(9S)

Name usb_pipe_close – Close and cleanup a USB device pipe

Synopsis

```
#include <sys/usb/usba.h>



void usb_pipe_close(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
     usb_flags_t flags,
     void (*callback)(usb_pipe_handle_t pipe_handle,
     usb_opaque_t arg, int rval,
     usb_cb_flags_t flags), usb_opaque_t callback_arg);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
pipe_handle	Handle of pipe to close. Cannot be a handle to the default control pipe.
flags	USB_FLAGS_SLEEP is the only flag recognized. Set it to wait for resources, for pipe to become free, and for all pending request callbacks to complete.
callback	This function is called on completion if the USB_FLAGS_SLEEP flag is not specified. Mandatory if the USB_FLAGS_SLEEP flag has not been specified.
callback_arg	Second argument to callback function.

Description The `usb_pipe_close()` function closes the pipe pointed to by *pipe_handle*, releases all related resources and then frees the pipe handle. This function stops polling if the pipe to be closed is an interrupt-IN or isochronous-IN pipe. The default control pipe cannot be closed.

Pipe cleanup includes waiting for the all pending requests in the pipe to finish, and then flushing residual requests remaining after waiting for several seconds. Exception handlers of flushed requests are called with a completion reason of USB_CR_FLUSHED.

If USB_FLAGS_SLEEP is specified in *flags*, wait for all cleanup operations to complete before calling the callback handler and returning.

If USB_FLAGS_SLEEP is not specified in *flags*, an asynchronous close (to be done in a separate thread) is requested. Return immediately. The callback handler is called after all pending operations are completed.

The *callback* parameter is the callback handler and takes the following arguments:

usb_pipe_handle_t pipe_handle	Handle of the pipe to close.
usb_opaque_t callback_arg	Callback_arg specified to `usb_pipe_close()`.
int rval	Return value of close operation
usb_cb_flags_t callback_flags	Status of queueing operation. Can be:

USB_CB_NO_INFO	Callback was uneventful.
USB_CB_ASYNC_REQ_FAILED	Error starting asynchronous request.

Return Values Status is returned to the caller via the callback handler's rval argument. Possible callback hander rval argument values are:

USB_INVALID_PIPE	Pipe handle specifies a pipe which is closed or closing.
USB_INVALID_ARGS	*dip* or *pipe_handle* arguments are NULL.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_INVALID_PERM	Pipe handle specifies the default control pipe.
USB_FAILURE	Asynchronous resources are unavailable. In this case, USB_CB_ASYNC_REQ_FAILED is passed in as the *callback_flags* arg to the callback hander.

Exception handlers of any queued requests which were flushed are called with a completion reason of USB_CR_FLUSHED. Exception handlers of periodic pipe requests which were terminated are called with USB_CR_PIPE_CLOSING.

Note that messages mirroring the above errors are logged to the console logfile on error. (This provides status for calls which otherwise could provide status).

Context May be called from user or kernel context regardless of arguments. May not be called from a callback executing in interrupt context. Please see `usb_callback_flags(9S)` for more information on callbacks.

If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags(9S)` for more information on callbacks.

Examples

```
/* Synchronous close of pipe. */
usb_pipe_close(dip, pipe, USB_FLAGS_SLEEP, NULL, NULL);

    --------

/* Template callback. */
void close_callback(usb_pipe_handle_t, usb_opaque_t, usb_cb_flags_t);

/* Asynchronous close of pipe. */
usb_pipe_close(dip, pipe, 0, close_callback, callback_arg);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_get_status(9F)`, `usb_pipe_drain_reqs(9F)`, `usb_pipe_get_state(9F)`, `usb_pipe_open(9F)`, `usb_pipe_reset(9F)`, `usb_callback_flags(9S)`

Name

usb_pipe_ctrl_xfer, usb_pipe_ctrl_xfer_wait – USB control pipe transfer functions

Synopsis

```
#include <sys/usb/usba.h>



int  usb_pipe_ctrl_xfer(usb_pipe_handle_t pipe_handle,
     usb_ctrl_req_t *request,
     usb_flags_t flags);



int  usb_pipe_ctrl_xfer_wait(usb_pipe_handle_t pipe_handle,
     usb_ctrl_setup_t *setup,
     mblk_t **data, usb_cr_t * completion_reason,
     usb_cb_flags_t *cb_flags,
     usb__flags_t flags);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

For `usb_pipe_ctrl_xfer()`:

pipe_handle	Control pipe handle on which request is made.
request	Pointer to control transfer request.
flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for all pending request callbacks to complete.

For `usb_pipe_ctrl_xfer_wait()`:

pipe_handle	Control pipe handle on which request is made.
setup	Pointer to setup parameters. (See below.)
data	Pointer to mblk containing data bytes to transfer with command. Ignored if NULL.
completion_reason	Returns overall completion status. Ignored if NULL. Please see `usb_callback_flags(9S)` for more information.
callback_flags	Returns flags set either during autoclearing or some other callback, which indicate recovery handling done in callback. Ignored if NULL.
flags	No flags are recognized. Reserved for future expansion.

Description

The `usb_pipe_ctrl_xfer()` function requests the USBA framework to perform a transfer through a USB control pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request. Requests are synchronous when USB_FLAGS_SLEEP is specified in flags; calls for synchronous requests do not return until their transaction is completed. Asynchronous requests (made without specifying the USB_FLAGS_SLEEP flag) notifies the caller of their completion via a callback function.

The `usb_pipe_ctrl_xfer_wait()` function is a wrapper around `usb_pipe_ctrl_xfer()` that performs allocation and deallocation of all required data structures, and a synchronous control-pipe transfer. It takes a usb_ctrl_setup_t containing most usb setup parameters as an argument:

```
uchar_t      bmRequestType   /* characteristics of request. */
                             /* (See USB 2.0 spec, section 9.3). */
                             /*  Combine one direction of:   */
                             /*      USB_DEV_REQ_HOST_TO_DEV */
                             /*      USB_DEV_REQ_DEV_TO_HOST */
                             /*  with one request type of:   */
                             /*      USB_DEV_REQ_TYPE_STANDARD */
                             /*      USB_DEV_REQ_TYPE_CLASS  */
                             /*      USB_DEV_REQ_TYPE_VENDOR */
                             /*  with one recipient type of: */
                             /*      USB_DEV_REQ_RCPT_DEV    */
                             /*      USB_DEV_REQ_RCPT_IF     */
                             /*      USB_DEV_REQ_RCPT_EP     */
                             /*      USB_DEV_REQ_RCPT_OTHER. */

uchar_t      bRequest        /* request or command.          */
                             /* (See USB 2.0 spec, section   */
                             /* 9.3 for standard commands.)  */

uint16_t     wValue         /* value which varies according to */
                             /* the command (bRequest).      */

uint16_t     wIndex          /* value which varies according to */
                             /* the command, typically used to */
                             /* pass an index or offset. */

uint16_t     wLength         /* number of data bytes to transfer */
                             /* with command, if any.  Same as */
                             /* size of mblk "data" below.   */

usb_req_attrs_t attrs;       /* required request attributes */
```

Please see `usb_request_attributes`(9S), or refer to Section *5.5* of the *USB 2.0* specification for more information on these parameters. (The USB 2.0 specification is available at *www.usb.org*.)

Mblks for data are allocated optionally when a request is allocated via `usb_alloc_ctrl_req`(9F) by passing a positive value for the *len* argument. Control requests passing or receiving no supplemental data need not allocate an mblk.

Return Values

For `usb_pipe_ctrl_xfer()`:

USB_SUCCESS	Transfer was successful.

USB_INVALID_ARGS	Request is `NULL`.
USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_REQUEST	The request has been freed or otherwise invalidated. A set of conflicting attributes were specified. See `usb_request_attributes`(9S). The normal and/or exception callback is NULL and USB_FLAGS_SLEEP is not set. Data space not provided to a control request while ctrl_wLength is nonzero.
USB_INVALID_PIPE	Pipe handle is NULL or invalid. Pipe is closing or closed.
USB_NO_RESOURCES	Memory, descriptors or other resources unavailable.
USB_HC_HARDWARE_ERROR	Host controller is in error state.
USB_FAILURE	An asynchronous transfer failed or an internal error occurred. The pipe is in an unsuitable state (error, busy, not ready).

Additional status information may be available in the ctrl_completion_reason and ctrl_cb_flags fields of the request. Please see `usb_callback_flags`(9S) and `usb_completion_reason`(9S) for more information.

For `usb_pipe_ctrl_xfer_wait()`:

USB_SUCCESS	Request was successful.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_INVALID_ARGS	*dip* is NULL.

Any error code returned by `usb_pipe_ctrl_xfer()`.

Additional status information may be available in the ctrl_completion_reason and ctrl_cb_flags fields of the request. Please see `usb_callback_flags`(9S) and `usb_completion_reason`(9S) for more information.

Context The `usb_pipe_ctrl_xfer()` function may be called from kernel or user context without regard to arguments and from the interrupt context only when the USB_FLAGS_SLEEP flag is clear.

The `usb_pipe_ctrl_xfer_wait()` function may be called from kernel or user context.

Examples

```
/* Allocate, initialize and issue a synchronous control request. */

usb_ctrl_req_t ctrl_req;
void control_pipe_exception_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);

ctrl_req = usb_alloc_ctrl_req(dip, 0, USB_FLAGS_SLEEP);

ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_HOST_TO_DEV |
    USB_DEV_REQ_TYPE_CLASS | USB_DEV_REQ_RCPT_OTHER;

ctrl_req->ctrl_bRequest     = (uint8_t)USB_PRINTER_SOFT_RESET;
ctrl_req->ctrl_exc_cb       = control_pipe_exception_callback;
...
...
if ((rval = usb_pipe_ctrl_xfer(pipe, ctrl_req, USB_FLAGS_SLEEP))
    != USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: Error issuing USB cmd.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-------

/*
 * Allocate, initialize and issue an asynchronous control request to
 * read a configuration descriptor.
 */

usb_ctrl_req_t *ctrl_req;
void control_pipe_normal_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);
void control_pipe_exception_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);
struct buf *bp = ...;

ctrl_req =
    usb_alloc_ctrl_req(dip, sizeof(usb_cfg_descr_t), USB_FLAGS_SLEEP);

ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_DEV_TO_HOST |
    USB_DEV_REQ_TYPE_STANDARD | USB_DEV_REQ_RCPT_DEV;

ctrl_req->ctrl_wLength      = sizeof(usb_cfg_descr_t);
```

```
ctrl_req->ctrl_wValue       = USB_DESCR_TYPE_SETUP_CFG | 0;
ctrl_req->ctrl_bRequest     = (uint8_t)USB_REQ_GET_DESCR;
ctrl_req->ctrl_cb           = control_pipe_normal_callback;
ctrl_req->ctrl_exc_cb       = control_pipe_exception_callback;

/* Make buf struct available to callback handler. */
ctrl_req->ctrl_client_private = (usb_opaque_t)bp;
...
...
if ((rval = usb_pipe_ctrl_xfer(pipe, ctrl_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: Error issuing USB cmd.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-------

/* Call usb_pipe_ctrl_xfer_wait() to get device status. */

mblk_t *data;
usb_cr_t completion_reason;
usb_cb_flags_t callback_flags;
usb_ctrl_setup_t setup_params = {
    USB_DEV_REQ_DEV_TO_HOST |       /* bmRequestType */
        USB_DEV_REQ_TYPE_STANDARD | USB_DEV_REQ_RCPT_DEV,
    USB_REQ_GET_STATUS,             /* bRequest */
    0,                              /* wValue */
    0,                              /* wIndex */
    USB_GET_STATUS_LEN,             /* wLength */
    0                               /* attributes. */
};

if (usb_pipe_ctrl_xfer_wait(
    pipe,
    &setup_params,
    &data,
    &compleetion_reason,
    &callback_flags,
    0) != USB_SUCCESS) {
        cmn_err (CE_WARN,
            "%s%d: USB get status command failed: "
            "reason=%d callback_flags=0x%x",
            ddi_driver_name(dip), ddi_get_instance(dip),
            completion_reason, callback_flags);
        return (EIO);
}
```

```
/* Check data length.  Should be USB_GET_STATUS_LEN (2 bytes). */
length_returned = data->b_wptr - data->b_rptr;
if (length_returned != USB_GET_STATUS_LEN) {
      cmn_err (CE_WARN,
          "%s%d: USB get status command returned %d bytes of data.",
          ddi_driver_name(dip), ddi_get_instance(dip), length_returned);
        return (EIO);
}

/* Retrieve data in endian neutral way. */
status = (*(data->b_rptr + 1) << 8) | *(data->b_rptr);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_alloc_request(9F)`, `usb_get_cfg(9F)`, `usb_get_status(9F)`. `usb_pipe_bulk_xfer(9F)`, `usb_pipe_intr_xfer(9F)`, `usb_pipe_isoc_xfer(9F)`, `usb_pipe_open(9F)`, `usb_pipe_reset(9F)`, `usb_pipe_get_state(9F)`, `usb_bulk_request(9S)`, `usb_callback_flags(9S)`, `usb_ctrl_request(9S)`, `usb_completion_reason(9S)`, `usb_intr_request(9S)`, `usb_isoc_request(9S)`

Name usb_pipe_drain_reqs – Allow completion of pending pipe requests

Synopsis
```
#include <sys/usb/usba.h>



int usb_pipe_drain_reqs(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
     uint_t timeout, usb_flags_t usb_flags,
     void (*callback)(usb_pipe_handle_t pipe_handle,
     usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
     usb_opaque_t callback_arg);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
pipe_handle	Handle of the pipe containing pending requests.
timeout	Maximum wait time for requests to drain. Must be a non-negative value in seconds. Zero specifies no timeout.
flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for completion and do not call callback.
callback	Callback handler to notify of asynchronous completion.
callback_arg	Second argument passed to callback function.

Description The `usb_pipe_drain_reqs()` function provides waits for pending requests to complete and then provides synchronous or asynchronous notification that all pending requests on a non-shared pipe indicated by pipe_handle have completed. For a shared pipe (such as the default control pipe used by multiple drivers each managing one interface of a device), this function provides notification that all pending requests on that pipe that are associated with a given dip are completed.

The `usb_pipe_drain_reqs()` function can be used to notify a close procedure when the default control pipe is clear during device closure, thereby allowing the close procedure to continue safely. Normally, a synchronous call to `usb_pipe_close`(9F) allows all requests in a pipe to finish before returning. However, a client driver cannot close the default control pipe.

If USB_FLAGS_SLEEP is set in flags, block until all pending requests are completed. Otherwise, return immediately and call the callback handler when all pending requests are completed.

The *callback* parameter accepts the asynchronous callback handler, which takes the following arguments:

usb_pipe_handle_t default_pipe_handle	Handle of the pipe to drain.

usb_opaque_t callback_arg	callback_arg specified to `usb_pipe_drain_reqs()`.	
int rval	Request status.	
usb_cb_flags_t callback_flags	Status of the queueing operation. Can be:	
	USB_CB_NO_INFO	Callback was uneventful.
	USB_CB_ASYNC_REQ_FAILED	Error starting asynchronous request.

Return Values

USB_SUCCESS	Request is successful.
USB_INVALID_ARGS	*dip* argument is `NULL`. USB_FLAGS_SLEEP is clear and callback is `NULL`.
USB_INVALID_CONTEXT	Called from callback context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_PIPE	Pipe is not open, is closing or is closed.

Context

May be called from user or kernel context.

If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags`(9S) for more information on callbacks.

Examples

```
 mydev_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{

     ...
     ...

     mydev_state->pipe_state = CLOSED;

     /* Wait for pending requests of a pipe to finish.  Don’t timeout. */
     (void) usb_pipe_drain_reqs(
         dip, pipe_handle, 0, USB_FLAGS_SLEEP, NULL, 0);

     /*
      * Dismantle streams and tear down this instance,
      * now that all requests have been sent.
      */
     qprocsoff(q);
     ...
```

```
          ...
          ddi_remove_minor_node(dip, NULL);
          ...
          ...
 }
```

Notes For pipes other than the default control pipe, it is recommended to close the pipe using a synchronous `usb_pipe_close()`. `usb_pipe_close()` with the USB_FLAGS_SLEEP flag allows any pending requests in that pipe to complete before returning.

Do not call `usb_pipe_drain_reqs()` while additional requests are being submitted by a different thread. This action can stall the calling thread of `usb_pipe_drain_reqs()` unnecessarily.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_pipe_close(9F)`, `usb_pipe_reset(9F)`, `usb_callback_flags(9S)`

Name usb_pipe_get_max_bulk_transfer_size – Get maximum bulk transfer size

Synopsis

```
#include <sys/usb/usba.h>



int usb_pipe_get_max_bulk_transfer_size(dev_info_t dip, size_t *size);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's dev_info structure.
size	Returns the bulk transfer size.

Description The `usb_pipe_get_max_bulk_transfer_size()` function returns the maximum data transfer size in bytes that the host controller driver can support per bulk request. This information can be used to limit or break down larger requests to manageable sizes.

Return Values

USB_SUCCESS	Size is returned in size argument.
USB_INVALID_ARGS	*dip* and/or *size* argument is `NULL`.
USB_FAILURE	Size could not be returned. Zero is returned in *size* arg.

Context May be called from user, kernel or interrupt context.

Examples

```
    int xxx_attach(dev_info_t *dip, int command)
    {
            ...
            usb_pipe_get_max_bulk_transfer_size(dip, &state>max_xfer_size);
            ...
    }

    void xxx_minphys(struct buf bp)
    {
            ...
            if (bp->b_bcount > state->max_xfer_size) {
                    bp->b_bcount = state->max_xfer_size;
            }
            ...
    }
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes`(5), `usb_pipe_bulk_xfer`(9F), `usb_alloc_request`(9F), `usb_bulk_request`(9S)

Name

usb_pipe_get_state – Return USB pipe state

Synopsis

```
#include <sys/usb/usba.h>



int usb_pipe_get_state(usb_pipe_handle_t pipe_handle, usb_pipe_state_t *pipe_state,
     usb_flags_t usb_flags);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

pipe_handle	Handle of the pipe to retrieve the state.
pipe_state	Pointer to where pipe state is returned.
usb_flags	No flags are recognized. Reserved for future expansion.

Description

The `usb_pipe_get_state()` function retrieves the state of the pipe referred to by *pipe_handle* into the location pointed to by *pipe_state*.

Possible pipe states are:

USB_PIPE_STATE_CLOSED	Pipe is closed.
USB_PIPE_STATE_ACTIVE	Pipe is active and can send/receive data. Polling is active for isochronous and interrupt pipes.
USB_PIPE_STATE_IDLE	Polling is stopped for isochronous and interrupt-IN pipes.
USB_PIPE_STATE_ERROR	An error occurred. Client must call `usb_pipe_reset()`. Note that this status is not seen by a client driver if USB_ATTRS_AUTOCLEARING is set in the request attributes.
USB_PIPE_STATE_CLOSING	Pipe is being closed. Requests are being drained from the pipe and other cleanup is in progress.

Return Values

USB_SUCCESS	Pipe state returned in second argument.
USB_INVALID_ARGS	Pipe_state argument is `NULL`.
USB_INVALID_PIPE	Pipe_handle argument is `NULL`.

Context

May be called from user, kernel or interrupt context.

Examples

```
usb_pipe_handle_t pipe;
usb_pipe_state_t state;

/* Recover if the pipe is in an error state. */
if ((usb_pipe_get_state(pipe, &state, 0) == USB_SUCCESS) &&
    (state == USB_PIPE_STATE_ERROR)) {
        cmn_err (CE_WARN, "%s%d: USB Pipe error.",
```

```
                ddi_driver_name(dip), ddi_get_instance(dip));
            do_recovery();
    }
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_clr_feature(9F)`, `usb_get_cfg(9F)`. `usb_get_status(9F)`, `usb_pipe_close(9F)`, `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_open(9F)`. `usb_pipe_reset(9F)`

Name usb_pipe_intr_xfer, usb_pipe_stop_intr_polling – USB interrupt transfer and polling functions

Synopsis

```
#include <sys/usb/usba.h>

int usb_pipe_intr_xfer(usb_pipe_handle_t pipe_handle,
     usb_intr_req_t *request, usb_flags_t flags);

void usb_pipe_stop_intr_polling(usb_pipe_handle_t pipe_handle,
     usb__flags_t flags);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_pipe_intr_xfer()`:

pipe_handle
Interrupt pipe handle on which request is made.

request
Pointer to interrupt transfer request.

flags
USB_FLAGS_SLEEP is the only flag recognized. Wait for needed resources if unavailable. For requests specifying the USB_ATTRS_ONE_XFER attribute, wait for the request to complete.

For `usb_pipe_stop_intr_polling()`:

pipe_handle
Interrupt pipe handle on which to stop polling for data.

flags
USB_FLAGS_SLEEP is the only flag recognized. Wait for polling to stop.

Description The `usb_pipe_intr_xfer()` function requests the USBA framework to perform a transfer through a USB interrupt pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request.

There are three categories of interrupt transfers: periodic or polled interrupt-IN, single-transfer interrupt-IN, and (single-transfer) interrupt-OUT.

Periodic Interrupt-IN Transfers

Periodic or polled interrupt-IN transfers execute on input requests which do not have the USB_ATTRS_ONE_XFER attribute set. One request enables repetitive transfers at a periodic rate set by the endpoint's bInterval. There can be only one interrupt-IN request submitted at a time.

Periodic interrupt-IN transfers are always asynchronous. Client driver notification of new data is always via a callback. The USB_FLAGS_SLEEP flag is only to wait for resources to become available. Callbacks must always be in place to receive transfer completion notification. Please see `usb_callback_flags(9S)` for details on USB callbacks.

Calls made to `usb_pipe_intr_xfer()` for starting input polling need allocate only one request. The USBA framework allocates a new request each time polling has new data to return. (Note that each request returned must be freed via `usb_free_intr_req`(9F). Specify a zero length when calling `usb_alloc_intr_req()` to allocate the original request, since it will not be used to return data. Set the intr_len in the request to specify how much data can be returned per polling interval.

The original request passed to `usb_pipe_intr_xfer()` is used to return status when polling is terminated, or on an error condition when the USB_ATTRS_AUTOCLEARING attribute is set for the request. If autoclearing is not set, the current (non-original) request is returned on error. Call `usb_pipe_reset`(9F) to reset the pipe and get back the original request in this case. The USB_CR_STOPPED_POLLING flag is always set for callbacks where the original request is returned.

Single-transfer Interrupt-IN Transfers

Interrupt-IN requests which have the USB_ATTRS_ONE_XFER attribute perform a single transfer. Such requests are synchronous when the USB_FLAGS_SLEEP flag is specified. Calls for synchronous requests do not return until their transaction is complete, and their callbacks are optional. The request is returned to the client through the normal or the exception completion callback to signal either normal completion or an error condition.

Interrupt-OUT Transfers

Interrupt-OUT requests always set up for a single transfer. However, multiple requests can be queued and execute in periodic fashion until depleted.

Interrupt-OUT transfers are synchronous when the USB_FLAGS_SLEEP flag is set in the request's flags. Calls for synchronous transfers will not return until their transaction has completed. Calls for asynchronous transfers notify the client driver of transaction completion via a normal callback, or error completion via an exception callback.

The `usb_pipe_stop_intr_polling()` function terminates polling on interrupt-IN pipes and does the following:

1. Cease polling.
2. Allow any requests-in-progress to complete and be returned to the client driver through the normal callback mechanism.
3. Idle the pipe.
4. Return the original polling request to the client driver through an exception callback with a completion reason of USB_CR_STOPPED_POLLING.

The client driver may restart polling from an exception callback only if the callback corresponds to an original request. The callback handler checks for the following completion reasons to ensure that a callback corresponds to an original request:

```
USB_CR_STOPPED_POLLING,
USB_CR_PIPE_RESET,
USB_CR_PIPE_CLOSING,
USB_CR_NOT_SUPPORTED
```

The callback handler also checks the request's intr_data field to mark original polling requests, when the requests are created with a zero *len* argument. In this case, a NULL intr_data field distinguishes a returned original request from a request allocated by the framework during polling.

Mblks for data for interrupt-OUT requests are allocated when a request is allocated via `usb_alloc_intr_req(9F)` by passing a positive value for the *len* argument.

Return Values For `usb_pipe_intr_xfer()`

USB_SUCCESS
Transfer was successful.

USB_INVALID_ARGS
Request is `NULL`.

USB_INVALID_CONTEXT
Called from interrupt context with the USB_FLAGS_SLEEP flag set.

USB_INVALID_REQUEST
The request has been freed or otherwise invalidated.

A set of conflicting attributes was specified. See `usb_intr_request(9S)`.

The normal and/or exception callback was NULL, USB_FLAGS_SLEEP was not set and USB_ATTRS_ONE_XFER was not set.

An interrupt request was specified with a zero intr_len value.

An IN interrupt request was specified with both polling (USB_ATTRS_ONE_XFER clear in attributes) and non-zero timeout specified.

An IN interrupt request was specified with a non-NULL data argument.

An OUT interrupt request was specified with a NULL data argument.

USB_INVALID_PIPE
Pipe handle is NULL or invalid.

Pipe is closing or closed.

USB_PIPE_ERROR
Pipe handle refers to a pipe which is in the USB_PIPE_STATE_ERROR state.

USB_NO_RESOURCES
Memory, descriptors or other resources unavailable.

USB_HC_HARDWARE_ERROR
Host controller is in error state.

USB_FAILURE
An asynchronous transfer failed or an internal error occurred.

An intr polling request is made while polling is already in progress.

The pipe is in an unsuitable state (error, busy, not ready).

Additional status information may be available in the intr_completion_reason and intr_cb_flags fields of the request. Please see `usb_completion_reason`(9S) and `usb_callback_flags`(9S) for more information.

For `usb_pipe_stop_intr_polling()`

None, but fails if called with USB_FLAGS_SLEEP specified from interrupt context, pipe handle is invalid, NULL or pertains to a closing or closed pipe, or the pipe is in an error state. Error messages are logged to the console logfile.

Exception handlers' queued requests which are flushed by these commands before execution are returned with completion reason of USB_CR_FLUSHED.

Context

Both of these functions can be called from kernel or user context without regard to arguments, and may be called from interrupt context only when the USB_FLAGS_SLEEP flag is clear.

Examples

```
/* Start polling on interrupt-IN pipe. */

usb_intr_req_t intr_req;
void intr_pipe_callback(usb_pipe_handle_t, usb_intr_req_t*);
void intr_pipe_exception_callback(
    usb_pipe_handle_t, usb_intr_req_t*);
usb_ep_descr_t *ep_descr;

ep_descr = ...;
intr_req = usb_alloc_intr_req(dip, 0, USB_FLAGS_SLEEP);
...
...
intr_req->intr_attributes   = USB_ATTRS_SHORT_XFER_OK;
intr_req->intr_len          = ep_descr->wMaxPacketSize;
...
...
intr_req->intr_cb           = intr_pipe_callback;
intr_req->intr_exc_cb       = intr_pipe_exception_callback;

if ((rval = usb_pipe_intr_xfer(pipe, intr_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
      cmn_err (CE_WARN, "%s%d: Error starting interrupt pipe polling.",
          ddi_driver_name(dip), ddi_get_instance(dip));
}

-------
```

```
/* Stop polling before setting device idle.  Wait for polling to stop. */

usb_pipe_stop_intr_polling(pipe, USB_FLAGS_SLEEP);
(void) pm_idle_component(dip, 0);

-------

/* Allocate, initialize and issue a synchronous intr-OUT request. */

usb_intr_req_t intr_req;
mblk_t *mblk;
usb_ep_descr_t *ep_descr;

ep_descr = ...;

intr_req =
  usb_alloc_intr_req(dip, ep_descr->wMaxPacketSize, USB_FLAGS_SLEEP);

intr_req->intr_attributes   = USB_ATTRS_AUTOCLEARING;
mblk = intr_req->intr_data;
bcopy(buffer, mblk->b_wptr, ep_descr->wMaxPacketSize);
mblk->b_wptr += ep_descr->wMaxPacketSize;

if ((rval = usb_pipe_intr_xfer(pipe, intr_req, USB_FLAGS_SLEEP))
    != USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: Error writing intr data.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_alloc_request(9F)`, `usb_get_cfg(9F)`, `usb_get_status(9F)`, `usb_pipe_bulk_xfer(9F)`, `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_get_state(9F)`, `usb_pipe_isoc_xfer(9F)`, `usb_pipe_open(9F)`, `usb_pipe_reset(9F)`, `usb_bulk_request(9S)`, `usb_callback_flags(9S)`, `usb_completion_reason(9S)`, `usb_ctrl_request(9S)`, `usb_ep_descr(9S)`, `usb_intr_request(9S)`, `usb_isoc_request(9S)`,

Name usb_pipe_isoc_xfer, usb_pipe_stop_isoc_polling – USB isochronous transfer and polling functions

Synopsis

```
#include <sys/usb/usba.h>



int usb_pipe_isoc_xfer(usb_pipe_handle_t pipe_handle, usb_isoc_req_t *request,
     usb_flags_t flags);



void usb_pipe_stop_isoc_polling(usb_pipe_handle_t pipe_handle, usb__flags_t flags);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_pipe_isoc_xfer()`:

pipe_handle	Isochronous pipe handle on which request is made.
request	Pointer to isochronous transfer request.
flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for needed resources if unavailable.

For `usb_pipe_stop_isoc_polling()`:

pipe_handle	Isochronous pipe handle on which to stop polling for input.
flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for polling to stop.

Description The `usb_pipe_isoc_xfer()` function requests the USBA framework to perform a transfer through a USB isochronous pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request.

By their nature, isochronous transfers require several transactions for completion. Each request may contain several packet descriptors. Descriptors correspond to subtransfers to be made in different frames. A request is deemed completed once all packets of that request have been processed. It is illegal to specify the USB_ATTRS_ONE_XFER attribute in an isochronous request. The isochronous polling interval is always one millisecond, the period of a full-speed frame.

All isochronous requests are asynchronous, and will notify the caller of their completion via a callback function. All isochronous requests must specify normal and exception callback handlers.

Requests will wait for needed, unavailable resources when USB_FLAGS_SLEEP has been specified in flags. Requests made without USB_FLAGS_SLEEP set will fail if needed resources are not readily available.

No errors seen during request processing will result in aborted transfers or exception callbacks. Such errors will instead be logged in the packet descriptor's isoc_pkt_status field. These errors can be examined when the completed request is returned through a normal callback.

Isochronous-OUT TRANSFERS

Allocate room for data when allocating isochronous-OUT requests via usb_alloc_isoc_req(9F), by passing a positive value for the *len* argument. The data will be divided among the request transactions, each transaction represented by a packet descriptor. (See usb_isoc_request(9F). When all of the data has been sent, regardless of any errors encountered, a normal transfer callback will be made to notify the client driver of completion.

If a request is submitted while other requests are active or queued, and the new request has its USB_ATTRS_ISOC_XFER_ASAP attribute set, the host controller driver will queue the request to start on a frame which immediately follows the last frame of the last queued request.

Isochronous-IN TRANSFERS

All isochronous-IN transfers start background polling, and require only a single (original) request. The USBA framework will allocate a new request each time polling has new data to return. Specify a zero length when calling usb_alloc_isoc_req() to allocate the original request, since it will not be used to return data. Set the isoc_pkts_length in the request to specify how much data to poll per interval (the length of one packet in the request).

The original request passed to `usb_pipe_isoc_xfer()` will be used to return status when polling termination is requested, or for error condition notification. There can be only one isochronous-IN request submitted at a time.

CALLBACKS

Isochronous transfer normal-completion callbacks cannot block for any reason since they are called from interrupt context. They will have USB_CB_INTR_CONTEXT set in their callback flags to note this.

Isochronous exception callbacks have the following restrictions for blocking:

1. They can block for resources (for example to allocate memory).
2. They cannot block for synchronous completion of a command (for example usb_pipe_close(9F)) done on the same pipe. Asynchronous commands can be started, when the pipe's policy pp_max_async_reqs field is initialized to accommodate them.
3. They cannot block waiting for another callback to complete.
4. They cannot block waiting for a synchronous transfer request to complete. They can, however, make an asynchronous request (such as restarting polling with a new isochronous-IN transfer).

Please see the section on callbacks in `usb_callback_flags`(9S) for more information.

All isochronous transfer exception callbacks signify that polling has stopped. Polling requests are returned with the following completion reasons:

`USB_CR_STOPPED_POLLING`
`USB_CR_PIPE_CLOSING`

Note: There are no exception callbacks for error conditions.

The `usb_pipe_stop_isoc_polling()` function terminates polling on an isochronous-IN pipe. The `usb_pipe_stop_isoc_polling()` function does the following:

1. Cease polling.
2. Allow any requests-in-progress to complete and be returned to the client driver through the normal callback mechanism.
3. Idle the pipe.
4. Return the original polling request to the client driver through an exception callback with a completion reason of USB_CR_STOPPED_POLLING.

Return Values For `usb_pipe_isoc_xfer()`:

USB_SUCCESS	Transfer was successful.
USB_INVALID_ARGS	Request is `NULL`.
USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_REQUEST	The request has been freed or otherwise invalidated. A set of conflicting attributes were specified. See `usb_isoc_request`(9S). The normal and/or exception callback was NULL, USB_FLAGS_SLEEP was not set and USB_ATTRS_ONE_XFER was not set. An isochronous request was specified with a zeroed isoc_pkt_descr, a NULL isoc_pkt_descr, or a NULL data argument. An isochronous request was specified with USB_ATTRS_ISOC_XFER_ASAP and a nonzero isoc_frame_no.
USB_NO_FRAME_NUMBER	An isochronous request was not specified with one and only one of USB_ATTRS_ISOC_START_FRAME or USB_ATTRS_ISOC_XFER_ASAP specified. An isochronous request was specified with USB_ATTRS_ISOC_START_FRAME and a zero isoc_frame_no.

USB_INVALID_START_FRAME	An isochronous request was specified with an invalid starting frame number (less than current frame number, or zero) and USB_ATTRS_ISOC_START_FRAME specified.
USB_INVALID_PIPE	Pipe handle is NULL or invalid. Pipe is closing or closed.
USB_PIPE_ERROR	Pipe handle refers to a pipe which is in the USB_PIPE_STATE_ERROR state.
USB_NO_RESOURCES	Memory, descriptors or other resources unavailable.
USB_HC_HARDWARE_ERROR	Host controller is in error state.
USB_FAILURE	An asynchronous transfer failed or an internal error occurred.

```
An isoch request requested too much data:
    (length   >  (usb_get_max_pkts_per_isoc_request()  *
     endpoint's  wMaxPacketSize))
```

The pipe is in an unsuitable state (error, busy, not ready).

Additional status information may be available in the isoc_completion_reason and isoc_cb_flags fields of the request. Please see `usb_completion_reason(9S)` and `usb_callback_flags(9S)` for more information.

For `usb_pipe_stop_isoc_polling()`:

None, but will fail if called with USB_FLAGS_SLEEP specified from interrupt context; the pipe handle is invalid, NULL or pertains to a closing or closed pipe; or the pipe is in an error state. Messages regarding these errors will be logged to the console logfile.

Context

Both of these functions may be called from kernel or user context without regard to arguments. May be called from interrupt context only when the USB_FLAGS_SLEEP flag is clear.

Examples

```
/* Start polling on an isochronous-IN pipe. */

    usb_isoc_req_t isoc_req;
    void isoc_pipe_callback(usb_pipe_handle_t, usb_isoc_req_t*);
    void isoc_pipe_exception_callback(
        usb_pipe_handle_t, usb_isoc_req_t*);
    uint_t pkt_size;
    usb_ep_data_t *isoc_ep_tree_node;
    usb_ep_descr_t *isoc_ep_descr = ...;   /* From usb_lookup_ep_data() */
```

```
isoc_ep_descr = &isoc_ep_tree_node->ep_descr;
pkt_size = isoc_ep_descr->wMaxPacketSize;

isoc_req = usb_alloc_isoc_req(
    dip, num_pkts, NUM_PKTS * pkt_size, USB_FLAGS_SLEEP);
...
...
isoc_req->isoc_attributes = USB_ATTRS_ISOC_XFER_ASAP;
...
...
isoc_req->isoc_cb         = isoc_pipe_callback;
isoc_req->isoc_exc_cb     = isoc_pipe_exception_callback;
...
...
isoc_req->isoc_pkts_length  = pkt_size;
isoc_req->isoc_pkts_count   = NUM_PKTS;
for (pkt = 0; pkt < NUM_PKTS; pkt++) {
        isoc_req->isoc_pkt_descr[pkt].isoc_pkt_length = pkt_size;
}

if ((rval = usb_pipe_isoc_xfer(pipe, isoc_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
        cmn_err (CE_WARN,"%s%d: Error starting isochronous pipe polling.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-------

/* Stop polling before powering off device. Wait for polling to stop. */

usb_pipe_stop_isoc_polling(pipe, USB_FLAGS_SLEEP);
pm_idle_component(dip, 0);
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_alloc_request(9F)`, `usb_get_current_frame_number(9F)`, `usb_get_cfg(9F)`, `usb_get_max_pkts_per_isoc_request(9F)`, `usb_get_status(9F)`, `usb_pipe_bulk_xfer(9F)`, `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_get_state(9F)`, `usb_pipe_intr_xfer(9F)`, `usb_pipe_open(9F)`, `usb_pipe_reset(9F)`, `usb_bulk_request`(9S), `usb_callback_flags`(9S), `usb_completion_reason`(9S), `usb_ctrl_request`(9S), `usb_ep_descr`(9S), `usb_intr_request`(9S), `usb_isoc_request`(9S)

Name

usb_pipe_open – Open a USB pipe to a device

Synopsis

```
#include <sys/usb/usba.h>



int usb_pipe_open(dev_info_t *dip, usb_ep_descr_t *endpoint, usb_pipe_policy_t *pipe_policy,
     usb_flags_t flags, usb_pipe_handle_t *pipe_handle);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
endpoint	Pointer to endpoint descriptor.
pipe_policy	Pointer to *pipe_policy*. *pipe_policy* provides hints on pipe usage.
flags	USB_FLAGS_SLEEP is only flag that is recognized. Wait for memory resources if not immediately available.
pipe_handle	Address to where new pipe handle is returned. (The handle is opaque.)

Description

A pipe is a logical connection to an endpoint on a USB device. The `usb_pipe_open()` function creates such a logical connection and returns an initialized handle which refers to that connection.

The *USB 2.0* specification defines four endpoint types, each with a corresponding type of pipe. Each of the four types of pipes uses its physical connection resource differently. They are:

Control pipe	Used for bursty, non-periodic, reliable, host-initiated request/response communication, such as for command/status operations. These are guaranteed to get approximately 10% of frame time and will get more if needed and if available, but there is no guarantee on transfer promptness. Bidirectional.
Bulk pipe	Used for large, reliable, non-time-critical data transfers. These get the bus on a bandwidth-available basis. Unidirectional. Sample uses include printer data.
Interrupt pipe	Used for sending or receiving small amounts of reliable data infrequently but with bounded service periods, as for interrupt handling. Unidirectional.
Isochronous pipe	Used for large, unreliable, time-critical data transfers. Boasts a guaranteed constant data rate as long as there is data, but there are no retries of failed transfers. Interrupt and isochronous data are together guaranteed 90% of frame time as needed. Unidirectional. Sample uses include audio.

The type of endpoint to which a pipe connects (and therefore the pipe type) is defined by the `bmAttributes` field of that pipe's endpoint descriptor. (See `usb_ep_descr(9S)`). Opens to interrupt and isochronous pipes can fail if the required bandwidth cannot be guaranteed.

The polling interval for periodic (interrupt or isochronous) pipes, carried by the endpoint argument's bInterval field, must be within range. Valid ranges are:

Full speed: range of 1-255 maps to 1-255 ms.

Low speed: range of 10-255 maps to 10-255 ms.

High speed: range of 1-16 maps to (2**(bInterval-1)) * 125us.

Adequate bandwidth during transfers is guaranteed for all periodic pipes which are opened successfully. Interrupt and isochronous pipes have guaranteed latency times, so bandwidth for them is allocated when they are opened. (Please refer to Sections *5.7* and *5.8* of the *USB 2.0* specification which address isochronous and interrupt transfers.) Opens of interrupt and isochronous pipes fail if inadequate bandwidth is available to support their guaranteed latency time. Because periodic pipe bandwidth is allocated on pipe open, open periodic pipes only when needed.

The bandwidth required by a device varies based on polling interval, the maximum packet size (`wMaxPacketSize`) and the device speed. Unallocated bandwidth remaining for new devices depends on the bandwidth already allocated for previously opened periodic pipes.

The *pipe_policy* parameter provides a hint as to pipe usage and must be specified. It is a `usb_pipe_policy_t` which contains the following fields:

```
uchar_t         pp_max_async_reqs:
                   A hint indicating how many
                   asynchronous operations requiring
                   their own kernel thread will be
                   concurrently in progress, the highest
                   number of threads ever needed at one
                   time.  Allow at least one for
                   synchronous callback handling and as
                   many as are needed to accommodate the
                   anticipated parallelism of asynchronous*
                   calls to the following functions:
                           usb_pipe_close(9F)
                           usb_set_cfg(9F)
                           usb_set_alt_if(9F)
                           usb_clr_feature(9F)
                           usb_pipe_reset(9F)
                           usb_pipe_drain_reqs(9F)
                           usb_pipe_stop_intr_polling(9F)
                           usb_pipe_stop_isoc_polling(9F)
```

```
Setting to too small a value can
deadlock the pipe.

* Asynchronous calls are calls made
  without the USB_FLAGS_SLEEP flag being
  passed.  Note that a large number of
  callbacks becomes an issue mainly when
  blocking functions are called from
  callback handlers.
```

The control pipe to the default endpoints (endpoints for both directions with addr 0, sometimes called the default control pipe or default pipe) comes pre-opened by the hub. A client driver receives the default control pipe handle through `usb_get_dev_data(9F)`. A client driver cannot open the default control pipe manually. Note that the same control pipe may be shared among several drivers when a device has multiple interfaces and each interface is operated by its own driver.

All explicit pipe opens are exclusive; attempts to open an opened pipe fail.

On success, the pipe_handle argument points to an opaque handle of the opened pipe. On failure, it is set to NULL.

Return Values

USB_SUCCESS	Open succeeded.
USB_NO_RESOURCES	Insufficient resources were available.
USB_NO_BANDWIDTH	Insufficient bandwidth available. (isochronous and interrupt pipes).
USB_INVALID_CONTEXT	Called from interrupt handler with USB_FLAGS_SLEEP set.
USB_INVALID_ARGS	dip and/or pipe_handle is NULL. Pipe_policy is NULL.
USB_INVALID_PERM	Endpoint is NULL, signifying the default control pipe. A client driver cannot open the default control pipe.
USB_NOT_SUPPORTED	Isochronous or interrupt endpoint with maximum packet size of zero is not supported.
USB_HC_HARDWARE_ERROR	Host controller is in an error state.
USB_FAILURE	Pipe is already open. Host controller not in an operational state. Polling interval (`ep_descr bInterval` field) is out of range (intr or isoc pipes).

Context May be called from user or kernel context regardless of arguments. May also be called from interrupt context if the *USB_FLAGS_SLEEP* option is not set.

Examples

```
usb_ep_data_t *ep_data;
usb_pipe_policy_t policy;
usb_pipe_handle_t pipe;
usb_client_dev_data_t *reg_data;
uint8_t interface = 1;
uint8_t alternate = 1;
uint8_t first_ep_number = 0;

/* Initialize pipe policy. */
bzero(policy, sizeof(usb_pipe_policy_t));
policy.pp_max_async_requests = 2;

/* Get tree of descriptors for device. */
if (usb_get_dev_data(
    dip, USBDRV_VERSION, &reg_data, USB_FLAGS_ALL_DESCR, 0) !=
    USB_SUCCESS) {
        ...
}

/* Get first interrupt-IN endpoint. */
ep_data = usb_lookup_ep_data(dip, reg_data, interface, alternate,
    first_ep_number, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
if (ep_data == NULL) {
    ...
}

/* Open the pipe.  Get handle to pipe back in 5th argument. */
if (usb_pipe_open(dip, &ep_data.ep_descr
    &policy, USB_FLAGS_SLEEP, &pipe) != USB_SUCCESS) {
        ...
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `usb_get_alt_if(9F)`, `usb_get_cfg(9F)`, `usb_get_status(9F)`, `usb_get_dev_data(9F)`, `usb_pipe_bulk_xfer(9F)`, `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_close(9F)`, `usb_pipe_get_state(9F)`, `usb_pipe_intr_xfer(9F)`, `usb_pipe_isoc_xfer(9F)`, `usb_pipe_reset(9F)`, `usb_pipe_set_private(9F)`, `usb_ep_descr(9S)`, `usb_callback_flags(9S)`

Name usb_pipe_reset – Abort queued requests from a USB pipe and reset the pipe

Synopsis

```
#include <sys/usb/usba.h>

void usb_pipe_reset(dev_info_t *dip,
     usb_pipe_handle_t pipe_handle, usb_flags_t usb_flags,
     void (*callback)(usb_pipe_handle_t cb_pipe_handle,
     usb_opaque_t arg, int rval, usb_cb_flags_t flags),
     usb_opaque_t callback_arg);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters

dip	Pointer to the device's `dev_info` structure.
pipe_handle	Handle of the pipe to reset. Cannot be the handle to the default control pipe.
usb_flags	USB_FLAGS_SLEEP is the only flag recognized. Wait for completion.
callback	Function called on completion if the USB_FLAGS_SLEEP flag is not specified. If NULL, no notification of completion is provided.
callback_arg	Second argument to callback function.

Description Call `usb_pipe_reset()` to reset a pipe which is in an error state, or to abort a current request and clear the pipe. The `usb_pipe_reset()` function can be called on any pipe other than the default control pipe.

A pipe can be reset automatically when requests sent to the pipe have the USB_ATTRS_AUTOCLEARING attribute specified. Client drivers see an exception callback with the USB_CB_STALL_CLEARED callback flag set in such cases.

Stalls on pipes executing requests without the USB_ATTRS_AUTOCLEARING attribute set must be cleared by the client driver. The client driver is notified of the stall via an exception callback. The client driver must then call `usb_pipe_reset()` to clear the stall.

The `usb_pipe_reset()` function resets a pipe as follows:

1. Any polling activity is stopped if the pipe being reset is an interrupt or isochronous pipe.
2. All pending requests are removed from the pipe. An exception callback, if specified beforehand, is executed for each aborted request.
3. The pipe is reset to the idle state.

Requests to reset the default control pipe are not allowed. No action is taken on a pipe which is closing.

If USB_FLAGS_SLEEP is specified in *flags*, this function waits for the action to complete before calling the callback handler and returning. If not specified, this function queues the request and returns immediately, and the specified callback is called upon completion.

callback is the callback handler. It takes the following arguments:

usb_pipe_handle_t cb_pipe_handle	Handle of the pipe to reset.
usb_opaque_t callback_arg	Callback_arg specified to `usb_pipe_reset()`.
int rval	Return value of the reset call.
usb_cb_flags_t callback_flags	Status of the queueing operation. Can be: USB_CB_NO_INFO — Callback was uneventful. USB_CB_ASYNC_REQ_FAILED — Error starting asynchronous request.

Return Values Status is returned to the caller via the callback handler's rval argument. Possible callback hander rval argument values are:

USB_SUCCESS	Pipe successfully reset.
USB_INVALID_PIPE	*pipe_handle* specifies a pipe which is closed or closing.
USB_INVALID_ARGS	*dip* or *pipe_handle* arguments are `NULL`. USB_FLAGS_SLEEP is clear and callback is NULL.
USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_PERM	*pipe_handle* specifies the default control pipe.
USB_FAILURE	Asynchronous resources are unavailable. In this case, USB_CB_ASYNC_REQ_FAILED is passed in as the *callback_flags* arg to the callback hander.

Exception callback handlers of interrupt-IN and isochronous-IN requests which are terminated by these commands are called with a completion reason of USB_CR_STOPPED_POLLING.

Exception handlers of incomplete bulk requests are called with a completion reason of USB_CR_FLUSHED.

Exception handlers of unstarted requests are called with USB_CR_PIPE_RESET.

Note that messages mirroring the above errors are logged to the console logfile on error. This provides status for calls which could not otherwise provide status.

Context May be called from user or kernel context regardless of arguments. May be called from any callback with the USB_FLAGS_SLEEP clear. May not be called from a callback executing in interrupt context if the USB_FLAGS_SLEEP flag is set.

If the USB_CB_ASYNC_REQ_FAILED bit is clear in usb_cb_flags_t, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see `usb_callback_flags`(9S) for more information on callbacks.

Examples

```
void post_reset_handler(
    usb_pipe_handle_t, usb_opaque_t, int, usb_cb_flags_t);

/*
 * Do an asynchronous reset on bulk_pipe.
 * Execute post_reset_handler when done.
 */
usb_pipe_reset(dip, bulk_pipe, 0, post_reset_handler, arg);

/* Do a synchronous reset on bulk_pipe. */
usb_pipe_reset(dip, bulk_pipe, USB_FLAGS_SLEEP, NULL, NULL);
```

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes`(5), `usb_get_cfg`(9F), `usb_pipe_bulk_xfer`(9F), `usb_pipe_close`(9F), `usb_get_status`(9F), `usb_pipe_ctrl_xfer`(9F), `usb_pipe_drain_reqs`(9F), `usb_pipe_get_state`(9F), `usb_pipe_intr_xfer`(9F), `usb_pipe_isoc_xfer`(9F), `usb_pipe_open`(9F), `usb_pipe_stop_intr_polling`(9F), `usb_pipe_stop_isoc_polling`(9F), `usb_callback_flags`(9S)

Name

usb_pipe_set_private, usb_pipe_get_private – USB user-defined pipe data-field facility

Synopsis

```
#include <sys/usb/usba.h>

int usb_pipe_set_private(usb_pipe_handle_t pipe_handle, usb_opaque_t data);

usb_opaque_t usb_pipe_get_private (usb_pipe_handle_t pipe_handle);
```

Interface Level

Solaris DDI specific (Solaris DDI)

Parameters

For `usb_pipe_set_private()`:

pipe_handle	Pipe handle into which user-defined data is placed.
data	Data to store in the pipe handle.

For `usb_pipe_get_private()`:

pipe_handle	Pipe handle from which user-defined data is retrieved.

Description

The `usb_set_driver_private()` function initializes the user-private data field of the pipe referred to by *pipe_handle*, using *data*. The user-private data field is used to store any data the client desires and is not used in any way by the USBA or OS framework. Client drivers often store their soft-state here for convenient retrieval by their callback handlers.

The `usb_get_driver_private()` function retrieves the user-private data stored via `usb_set_driver_private()`, from the pipe referred to by *pipe_handle*.

Return Values

For `usb_pipe_set_private()`:

USB_SUCCESS	Private data has been successfully stored in pipe handle.
USB_INVALID_PIPE	*pipe_handle* argument is `NULL` or invalid. Pipe is closing or closed.
USB_INVALID_PERM	The *pipe_handle* argument refers to the default control pipe.

For `usb_pipe_get_private()`:

On success: usb_opaque_t pointer to data being retrieved.

On failure: `NULL`. Fails if pipe handle is `NULL` or invalid. Fails if pipe handle is to a pipe which is closing or closed.

Context

May be called from user, kernel or interrupt context.

Examples

```
    usb_pipe_handle_t pipe;

    /* Some driver defined datatype. */
    xxx_data_t *data = kmem_zalloc(...);
```

```
usb_pipe_set_private(pipe, data);

----

xxx_data_t *xxx_data_ptr = (xxx_data_t *)usb_pipe_get_private(pipe);
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also `attributes(5)`, `usb_pipe_open(9F)`, `usb_alloc_request(9F)`

Name usb_register_hotplug_cbs, usb_unregister_hotplug_cbs – Register/unregister for notification of device hotplug events

Synopsis

```
#include <sys/usb/usba.h>



int  usb_register_hotplug_cbs(dev_info_t *dip,
     int (*disconnection_event_handler)(dev_info_t *dip,
     int (*reconnection_event_handler)(dev_info_t *dip);

 void usb_unregister_hotplug_cbs(dev_info_t *dip);
```

Interface Level Solaris DDI specific (Solaris DDI)

Parameters For `usb_register_hotplug_cbs()`

dip	Pointer to the device's `dev_info` structure.
disconnection_event_handler	Called when device is disconnected. This handler takes a dev_info_t as an argument (representing the device being disconnected) and always returns USB_SUCCESS.
reconnection_event_handler	Called when device is reconnected. This handler takes a dev_info_t as an argument (representing the device being reconnected) and always returns USB_SUCCESS.

For `usb_unregister_hotplug_cbs()`:

dip	Pointer to the device's `dev_info` structure.

Description The `usb_register_hotplug_cbs()` function registers callbacks to be executed when the USB device represented by *dip* is hotplugged or removed.

The `usb_unregister_hotplug_cbs()` function unregisters or disengages callbacks from executing when the USB device represented by *dip* is hotplugged or removed.

Return Values For `usb_register_hotplug_cbs()`:

USB_SUCCESS	Callbacks were successfully registered.
USB_FAILURE	One or more arguments were NULL. Callbacks could not be successfully registered.

For `usb_unregister_hotplug_cbs()`: None

Context The `usb_register_hotplug_cbs()` function may be called only from `attach(9E)`.

The `usb_unregister_hotplug_cbs()` function may be called only from `detach(9E)`.

Registered callback handlers requiring the use of any DDI (section 9F) function (except ddi_taskq_* functions), should launch a separate thread using ddi_taskq_* routines for processing their event, to avoid deadlocks. The new thread can then safely call any DDI function it needs to handle the event.

The registered callback handlers execute in kernel context.

Examples

```
int remove_device(dev_info_t *)
{
        ...
        ...
        return (USB_SUCCESS);
}

int accommodate_device(dev_info_t *)
{
        ...
        ...
        return (USB_SUCCESS);
}

if (usb_register_hotplug_cbs(
    dip, remove_device, accommodate_device) == USB_FAILURE) {
        cmn_err (CE_WARN,
            "%s%d: Could not register hotplug handlers.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also

`attributes(5)`, `attach(9E)`, `detach(9E)`, `usb_get_status(9F)`

Name uwritec – remove a character from a uio structure

Synopsis

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



int uwritec(uio_t *uio_p);
```

Interface Level Architecture independent level 1 (DDI/DKI)

Parameters *uio_p* Pointer to the `uio`(9S) structure

Description The `uwritec()` function returns a character from the `uio` structure pointed to by *uio_p* and updates the `uio` structure. See `uiomove`(9F).

Return Values The next character for processing is returned on success, and `-1` is returned if `uio` is empty or if there is an error.

Context The `uwritec()` function can be called from user, interrupt, or kernel context.

See Also `uiomove`(9F), `ureadc`(9F), `iovec`(9S), `uio`(9S)

Writing Device Drivers

Name

va_arg, va_start, va_copy, va_end – handle variable argument list

Synopsis

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



void va_start(va_list pvar, name);

(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).

Parameters

`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.

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 Solaris DDI specific (Solaris DDI).

Parameters

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>\n` in *buf*.

```
vsprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```

`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` on SPARC, the string `<nullstring>` is used in its place; on x86, 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)`.

```
#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);
```

EXAMPLE 1 Using `vsprintf()` *(Continued)*

```
    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);

    /* 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

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 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

The `WR()` function can be called from user, interrupt, or kernel 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.

```
xxxclose(q, flag)
    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