



# man pages section 9: DDI and DKI Kernel Functions

---

Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054  
U.S.A.

Part No: 817-3949-10  
September 2004

Copyright 2004 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright 2004 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Certaines parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux États-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux États-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



040714@9061



# Contents

---

**Preface** 13

**Introduction** 19

Intro(9F) 20

**Kernel Functions for Drivers** 39

adjmsg(9F) 40

alloca(9F) 41

anocancel(9F) 44

aphysio(9F) 45

ASSERT(9F) 47

backq(9F) 48

bcanput(9F) 49

bcmp(9F) 50

bcopy(9F) 51

biocloned(9F) 53

biodone(9F) 56

bioerror(9F) 58

biofini(9F) 59

bioinit(9F) 60

biomodified(9F) 61

bioreset(9F) 62

biosize(9F) 63

biowait(9F) 64

bp\_mapin(9F) 65

bp\_mapout(9F) 66  
 btop(9F) 67  
 btopr(9F) 68  
 bufcall(9F) 69  
 bzero(9F) 72  
 canput(9F) 73  
 canputnext(9F) 74  
 clrbuf(9F) 75  
 cmn\_err(9F) 76  
 condvar(9F) 81  
 copyb(9F) 84  
 copyin(9F) 86  
 copymsg(9F) 88  
 copyout(9F) 90  
 csx\_AccessConfigurationRegister(9F) 92  
 csx\_ConvertSize(9F) 94  
 csx\_ConvertSpeed(9F) 95  
 csx\_CS\_DDI\_Info(9F) 96  
 csx\_DeregisterClient(9F) 98  
 csx\_DupHandle(9F) 99  
 csx\_Error2Text(9F) 101  
 csx\_Event2Text(9F) 102  
 csx\_FreeHandle(9F) 103  
 csx\_Get8(9F) 104  
 csx\_GetFirstClient(9F) 105  
 csx\_GetFirstTuple(9F) 107  
 csx\_GetHandleOffset(9F) 109  
 csx\_GetMappedAddr(9F) 110  
 csx\_GetStatus(9F) 111  
 csx\_GetTupleData(9F) 115  
 csx\_MakeDeviceNode(9F) 117  
 csx\_MapLogSocket(9F) 119  
 csx\_MapMemPage(9F) 120  
 csx\_ModifyConfiguration(9F) 121  
 csx\_ModifyWindow(9F) 123  
 csx\_Parse\_CISTPL\_BATTERY(9F) 125  
 csx\_Parse\_CISTPL\_BYTEORDER(9F) 126  
 csx\_Parse\_CISTPL\_CFTABLE\_ENTRY(9F) 128

csx\_Parse\_CISTPL\_CONFIG(9F) 134  
 csx\_Parse\_CISTPL\_DATE(9F) 137  
 csx\_Parse\_CISTPL\_DEVICE(9F) 138  
 csx\_Parse\_CISTPL\_DEVICEGEO(9F) 141  
 csx\_Parse\_CISTPL\_DEVICEGEO\_A(9F) 143  
 csx\_Parse\_CISTPL\_FORMAT(9F) 145  
 csx\_Parse\_CISTPL\_FUNCE(9F) 147  
 csx\_Parse\_CISTPL\_FUNCID(9F) 155  
 csx\_Parse\_CISTPL\_GEOMETRY(9F) 157  
 csx\_Parse\_CISTPL\_JEDEC\_C(9F) 158  
 csx\_Parse\_CISTPL\_LINKTARGET(9F) 160  
 csx\_Parse\_CISTPL\_LONGLINK\_A(9F) 161  
 csx\_Parse\_CISTPL\_LONGLINK\_MFC(9F) 163  
 csx\_Parse\_CISTPL\_MANFID(9F) 165  
 csx\_Parse\_CISTPL\_ORG(9F) 166  
 csx\_Parse\_CISTPL\_SPCL(9F) 167  
 csx\_Parse\_CISTPL\_SWIL(9F) 169  
 csx\_Parse\_CISTPL\_VERS\_1(9F) 170  
 csx\_Parse\_CISTPL\_VERS\_2(9F) 171  
 csx\_ParseTuple(9F) 172  
 csx\_Put8(9F) 174  
 csx\_RegisterClient(9F) 175  
 csx\_ReleaseConfiguration(9F) 178  
 csx\_RepGet8(9F) 180  
 csx\_RepPut8(9F) 182  
 csx\_RequestConfiguration(9F) 184  
 csx\_RequestIO(9F) 188  
 csx\_RequestIRQ(9F) 193  
 csx\_RequestSocketMask(9F) 196  
 csx\_RequestWindow(9F) 198  
 csx\_ResetFunction(9F) 203  
 csx\_SetEventMask(9F) 204  
 csx\_SetHandleOffset(9F) 206  
 csx\_ValidateCIS(9F) 207  
 datamsg(9F) 208  
 ddi\_add\_intr(9F) 209  
 ddi\_add\_softintr(9F) 212  
 ddi\_binding\_name(9F) 219

ddi\_btop(9F) 220  
ddi\_check\_acc\_handle(9F) 221  
ddi\_copyin(9F) 223  
ddi\_copyout(9F) 226  
ddi\_create\_minor\_node(9F) 229  
ddi\_device\_copy(9F) 231  
ddi\_device\_zero(9F) 233  
ddi\_devid\_compare(9F) 234  
ddi\_dev\_is\_needed(9F) 238  
ddi\_dev\_is\_sid(9F) 240  
ddi\_dev\_nintrs(9F) 241  
ddi\_dev\_nregs(9F) 242  
ddi\_dev\_regsize(9F) 243  
ddi\_dev\_report\_fault(9F) 244  
ddi\_dma\_addr\_bind\_handle(9F) 247  
ddi\_dma\_addr\_setup(9F) 251  
ddi\_dma\_alloc\_handle(9F) 253  
ddi\_dma\_buf\_bind\_handle(9F) 255  
ddi\_dma\_buf\_setup(9F) 259  
ddi\_dma\_burstsizes(9F) 261  
ddi\_dma\_coff(9F) 262  
ddi\_dma\_curwin(9F) 263  
ddi\_dma\_devalign(9F) 264  
ddi\_dmae(9F) 265  
ddi\_dma\_free(9F) 269  
ddi\_dma\_free\_handle(9F) 270  
ddi\_dma\_get\_attr(9F) 271  
ddi\_dma\_getwin(9F) 272  
ddi\_dma\_htoc(9F) 274  
ddi\_dma\_mem\_alloc(9F) 275  
ddi\_dma\_mem\_free(9F) 278  
ddi\_dma\_movwin(9F) 279  
ddi\_dma\_nextcookie(9F) 281  
ddi\_dma\_nextseg(9F) 283  
ddi\_dma\_nextwin(9F) 285  
ddi\_dma\_numwin(9F) 287  
ddi\_dma\_segtocookie(9F) 288  
ddi\_dma\_set\_sbus64(9F) 290

ddi\_dma\_setup(9F) 291  
ddi\_dma\_sync(9F) 293  
ddi\_dma\_unbind\_handle(9F) 295  
ddi\_driver\_major(9F) 296  
ddi\_driver\_name(9F) 297  
ddi\_enter\_critical(9F) 298  
ddi\_ffs(9F) 299  
ddi\_get8(9F) 300  
ddi\_get\_cred(9F) 302  
ddi\_get\_devstate(9F) 303  
ddi\_get\_driver\_private(9F) 304  
ddi\_getiminor(9F) 305  
ddi\_get\_instance(9F) 306  
ddi\_get\_kt\_did(9F) 307  
ddi\_get\_lbolt(9F) 308  
ddi\_get\_parent(9F) 309  
ddi\_get\_pid(9F) 310  
ddi\_get\_time(9F) 311  
ddi\_in\_panic(9F) 312  
ddi\_intr\_hilevel(9F) 313  
ddi\_io\_get8(9F) 314  
ddi\_iomin(9F) 315  
ddi\_iopb\_alloc(9F) 316  
ddi\_io\_put8(9F) 318  
ddi\_io\_rep\_get8(9F) 320  
ddi\_io\_rep\_put8(9F) 322  
ddi\_log\_sysevent(9F) 324  
ddi\_mapdev(9F) 328  
ddi\_mapdev\_intercept(9F) 331  
ddi\_mapdev\_set\_device\_acc\_attr(9F) 333  
ddi\_map\_regs(9F) 335  
ddi\_mem\_alloc(9F) 337  
ddi\_mem\_get8(9F) 339  
ddi\_mem\_put8(9F) 340  
ddi\_mem\_rep\_get8(9F) 342  
ddi\_mem\_rep\_put8(9F) 344  
ddi\_mmap\_get\_model(9F) 346  
ddi\_model\_convert\_from(9F) 348

ddi\_node\_name(9F) 350  
ddi\_no\_info(9F) 351  
ddi\_peek(9F) 352  
ddi\_poke(9F) 354  
ddi\_prop\_create(9F) 356  
ddi\_prop\_exists(9F) 360  
ddi\_prop\_get\_int(9F) 362  
ddi\_prop\_lookup(9F) 364  
ddi\_prop\_op(9F) 369  
ddi\_prop\_update(9F) 373  
ddi\_put8(9F) 377  
ddi\_regs\_map\_free(9F) 379  
ddi\_regs\_map\_setup(9F) 380  
ddi\_remove\_minor\_node(9F) 382  
ddi\_removing\_power(9F) 383  
ddi\_rep\_get8(9F) 385  
ddi\_report\_dev(9F) 387  
ddi\_rep\_put8(9F) 388  
ddi\_root\_node(9F) 390  
ddi\_segmap(9F) 391  
ddi\_slaveonly(9F) 394  
ddi\_soft\_state(9F) 395  
ddi\_umem\_alloc(9F) 400  
ddi\_umem\_iosetup(9F) 402  
ddi\_umem\_lock(9F) 404  
delay(9F) 406  
devmap\_default\_access(9F) 408  
devmap\_devmem\_setup(9F) 410  
devmap\_do\_ctxmgt(9F) 413  
devmap\_set\_ctx\_timeout(9F) 416  
devmap\_setup(9F) 417  
devmap\_unload(9F) 419  
disksort(9F) 421  
drv\_getparm(9F) 422  
drv\_hztousec(9F) 424  
drv\_priv(9F) 425  
drv\_usectohz(9F) 426  
drv\_usecwait(9F) 427

dupb(9F) 428  
dupmsg(9F) 431  
enableok(9F) 432  
esballoc(9F) 433  
esbcall(9F) 435  
flushband(9F) 436  
flushq(9F) 437  
freeb(9F) 439  
freemsg(9F) 440  
freerbuf(9F) 441  
freezestr(9F) 442  
geterror(9F) 443  
gethrtime(9F) 444  
getmajor(9F) 445  
getminor(9F) 446  
get\_pktiopb(9F) 447  
getq(9F) 449  
getrbuf(9F) 450  
gld(9F) 451  
hat\_getkpfnum(9F) 454  
id32\_alloc(9F) 455  
inb(9F) 456  
insq(9F) 458  
IOC\_CONVERT\_FROM(9F) 460  
kmem\_alloc(9F) 461  
kmem\_cache\_create(9F) 463  
kstat\_create(9F) 468  
kstat\_delete(9F) 470  
kstat\_install(9F) 471  
kstat\_named\_init(9F) 472  
kstat\_queue(9F) 473  
linkb(9F) 475  
makecom(9F) 476  
makedevice(9F) 478  
max(9F) 479  
min(9F) 480  
mkiocb(9F) 481  
mod\_install(9F) 484

msgdsize(9F) 485  
 msgpullup(9F) 486  
 mt-streams(9F) 487  
 mutex(9F) 489  
 nochpoll(9F) 492  
 nodev(9F) 493  
 noenable(9F) 494  
 nulldev(9F) 495  
 nvlist\_add\_boolean(9F) 496  
 nvlist\_alloc(9F) 498  
 nvlist\_lookup\_boolean(9F) 500  
 nvlist\_next\_nvpair(9F) 502  
 nvlist\_remove(9F) 504  
 nvpair\_value\_byte(9F) 505  
 OTHERQ(9F) 507  
 outb(9F) 508  
 pci\_config\_get8(9F) 510  
 pci\_config\_setup(9F) 512  
 pci\_report\_pmcap(9F) 513  
 pci\_save\_config\_regs(9F) 515  
 physio(9F) 517  
 pm\_busy\_component(9F) 519  
 pm\_create\_components(9F) 521  
 pm\_get\_normal\_power(9F) 522  
 pm\_power\_has\_changed(9F) 524  
 pm\_raise\_power(9F) 526  
 pm\_trans\_check(9F) 529  
 pollwakeup(9F) 531  
 proc\_signal(9F) 532  
 ptob(9F) 534  
 pullupmsg(9F) 535  
 put(9F) 537  
 putbq(9F) 538  
 putctl1(9F) 539  
 putctl(9F) 540  
 putnext(9F) 542  
 putnextctl1(9F) 543  
 putnextctl(9F) 544

putq(9F) 546  
qassociate(9F) 547  
qbufcall(9F) 549  
qenable(9F) 550  
qprocson(9F) 551  
qreply(9F) 552  
qsize(9F) 554  
qtimeout(9F) 555  
qunbufcall(9F) 556  
quntimeout(9F) 557  
qwait(9F) 558  
qwriter(9F) 560  
RD(9F) 561  
rmalloc(9F) 562  
rmallocmap(9F) 565  
rmalloc\_wait(9F) 566  
rmfree(9F) 567  
rmvb(9F) 568  
rmvq(9F) 569  
rlock(9F) 571  
SAMESTR(9F) 574  
scsi\_abort(9F) 575  
scsi\_alloc\_consistent\_buf(9F) 576  
scsi\_cname(9F) 578  
scsi\_destroy\_pkt(9F) 580  
scsi\_dmaget(9F) 581  
scsi\_errmsg(9F) 583  
scsi\_free\_consistent\_buf(9F) 586  
scsi\_get\_device\_type\_scsi\_options(9F) 587  
scsi\_hba\_attach\_setup(9F) 589  
scsi\_hba\_init(9F) 592  
scsi\_hba\_lookup\_capstr(9F) 593  
scsi\_hba\_pkt\_alloc(9F) 595  
scsi\_hba\_probe(9F) 597  
scsi\_hba\_tran\_alloc(9F) 598  
scsi\_ifgetcap(9F) 599  
scsi\_init\_pkt(9F) 603  
scsi\_log(9F) 607

scsi\_pktalloc(9F) 609  
scsi\_poll(9F) 611  
scsi\_probe(9F) 612  
scsi\_reset(9F) 614  
scsi\_reset\_notify(9F) 615  
scsi\_setup\_cdb(9F) 616  
scsi\_slave(9F) 617  
scsi\_sync\_pkt(9F) 619  
scsi\_transport(9F) 620  
scsi\_unprobe(9F) 621  
scsi\_vu\_errmsg(9F) 622  
semaphore(9F) 625  
sprintf(9F) 627  
stoi(9F) 629  
strchr(9F) 630  
strcmp(9F) 631  
strcpy(9F) 632  
strlen(9F) 633  
strlog(9F) 634  
strqget(9F) 636  
strqset(9F) 637  
STRUCT\_DECL(9F) 638  
swab(9F) 643  
testb(9F) 644  
timeout(9F) 646  
uiomove(9F) 648  
unbufcall(9F) 649  
unlinkb(9F) 650  
untimeout(9F) 651  
ureadc(9F) 653  
uwritec(9F) 654  
va\_arg(9F) 655  
vsprintf(9F) 657  
WR(9F) 660

**Index 661**

# Preface

---

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

---

## Overview

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

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

- Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

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

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.								
SYNOPSIS	<p>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.</p> <p>The following special characters are used in this section:</p> <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 10px;">[ ]</td> <td>Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.</td> </tr> <tr> <td style="padding-right: 10px;">. . .</td> <td>Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "filename . . .".</td> </tr> <tr> <td style="padding-right: 10px;"> </td> <td>Separator. Only one of the arguments separated by this character can be specified at a time.</td> </tr> <tr> <td style="padding-right: 10px;">{ }</td> <td>Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.</td> </tr> </table>	[ ]	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.
[ ]	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 <code>ioctl(2)</code> system call is called <code>ioctl</code> and generates its own heading. <code>ioctl</code> calls for a specific device are listed alphabetically (on the man page for that specific device). <code>ioctl</code> calls are used for a particular class of devices all of which have an <code>io</code> ending, such as <code>mtio(7I)</code> .
OPTIONS	This section lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output – standard output, standard error, or output files – generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or -1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable <code>errno</code> 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 <code>example%</code> , or if the user must be superuser, <code>example#</code> . 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 <code>attributes(5)</code> 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.



# Introduction

---

## Intro(9F)

<b>NAME</b>	Intro – introduction to DDI/DKI functions
<b>DESCRIPTION</b>	<p>Section 9F describes the kernel functions available for use by device drivers. See <a href="#">Intro(9E)</a> for an overview of device driver interfaces.</p> <p>In this section, the information for each driver function is organized under the following headings:</p> <ul style="list-style-type: none"><li>■ <b>NAME</b> summarizes the function’s purpose.</li><li>■ <b>SYNOPSIS</b> shows the syntax of the function’s entry point in the source code. <code>#include</code> directives are shown for required headers.</li><li>■ <b>INTERFACE LEVEL</b> describes any architecture dependencies.</li><li>■ <b>ARGUMENTS</b> describes any arguments required to invoke the function.</li><li>■ <b>DESCRIPTION</b> describes general information about the function.</li><li>■ <b>RETURN VALUES</b> describes the return values and messages that can result from invoking the function.</li><li>■ <b>CONTEXT</b> indicates from which driver context (user, kernel, interrupt, or high-level interrupt) the function can be called.</li><li>■ A driver function has <i>user context</i> if it was directly invoked because of a user thread. The <code>read(9E)</code> entry point of the driver, invoked by a <code>read(2)</code> system call, has user context.</li><li>■ A driver function has <i>kernel context</i> if it was invoked by some other part of the kernel. In a block device driver, the <code>strategy(9E)</code> 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 <code>strategy(9E)</code> has kernel context.</li><li>■ <i>Interrupt context</i> is kernel context, but also has an interrupt level associated with it. Driver interrupt routines have interrupt context.</li></ul> <p><b>Note</b> – 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 <a href="#">mutex(9F)</a> man page for a more complete discussion of proper mutex handling for drivers.</p> <ul style="list-style-type: none"><li>■ <i>High-level interrupt context</i> is a more restricted form of interrupt context. If <code>ddi_intr_hilevel(9F)</code> indicates that an interrupt is high-level, driver interrupt routines added for that interrupt with <code>ddi_add_intr(9F)</code> run in high-level interrupt context. These interrupt routines are only allowed to call <code>ddi_trigger_softintr(9F)</code>, <code>mutex_enter(9F)</code> and <code>mutex_exit(9F)</code>. Furthermore, <code>mutex_enter(9F)</code> and <code>mutex_exit(9F)</code> may only be called on mutexes initialized with the <code>ddi_iblock_cookie</code> returned by <code>ddi_get_iblock_cookie(9F)</code>.</li><li>■ <b>SEE ALSO</b> indicates functions that are related by usage and sources, and which can be referred to for further information.</li><li>■ <b>EXAMPLES</b> shows how the function can be used in driver code.</li></ul>

**STREAMS Kernel  
Function Summary**

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

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

<b>Routine</b>	<b>Type</b>
adjmsg	DDI/DKI
allocb	DDI/DKI
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
datamsg	DDI/DKI
dupb	DDI/DKI
dupmsg	DDI/DKI
enableok	DDI/DKI
esballoc	DDI/DKI
esbcall	DDI/DKI
flushband	DDI/DKI
flushq	DDI/DKI
freeb	DDI/DKI
freemsg	DDI/DKI
freezestr	DDI/DKI
getq	DDI/DKI
insq	DDI/DKI
linkb	DDI/DKI
msgdsize	DDI/DKI

## Intro(9F)

<b>Routine</b>	<b>Type</b>
msgpullup	DDI/DKI
mt-streams	Solaris DDI
noenable	DDI/DKI
OTHERQ	DDI/DKI
pullupmsg	DDI/DKI
put	DDI/DKI
putbq	DDI/DKI
putctl	DDI/DKI
putctl1	DDI/DKI
putnext	DDI/DKI
putnextctl	DDI/DKI
putq	DDI/DKI
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

<b>Routine</b>	<b>Type</b>
strqget	DDI/DKI
strqset	DDI/DKI
testb	DDI/DKI
unbufcall	DDI/DKI
unfreezestr	DDI/DKI
unlinkb	DDI/DKI
WR	DDI/DKI

The following table summarizes the functions not specific to STREAMS.

<b>Routine</b>	<b>Type</b>
ASSERT	DDI/DKI
anocancel	Solaris DDI
aphysio	Solaris DDI
bcmp	DDI/DKI
bcopy	DDI/DKI
biodone	DDI/DKI
bioclone	Solaris DDI
biofini	Solaris DDI
bioinit	Solaris DDI
biomodified	Solaris DDI
biosize	Solaris DDI
bioerror	Solaris DDI
bioreset	Solaris DDI
biowait	DDI/DKI
bp_mapin	DDI/DKI
bp_mapout	DDI/DKI
btop	DDI/DKI
btopr	DDI/DKI
bzero	DDI/DKI

<b>Routine</b>	<b>Type</b>
cmn_err	DDI/DKI
copyin	DDI/DKI
copyout	DDI/DKI
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_intr	Solaris DDI
ddi_add_softintr	Solaris DDI
ddi_btop	Solaris DDI
ddi_btopr	Solaris DDI
ddi_copyin	Solaris DDI
ddi_copyout	Solaris DDI
ddi_create_minor_node	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

<b>Routine</b>	<b>Type</b>
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_htoc	Solaris SPARC DDI
ddi_dma_mem_alloc	Solaris DDI
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

## Intro(9F)

<b>Routine</b>	<b>Type</b>
ddi_dmae_release	Solaris x86 DDI
ddi_dmae_stop	Solaris x86 DDI
ddi_enter_critical	Solaris DDI
ddi_exit_critical	Solaris DDI
ddi_ffs	Solaris DDI
ddi_fls	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_driver_private	Solaris DDI
ddi_get_iblock_cookie	Solaris DDI
ddi_get_instance	Solaris DDI
ddi_get_name	Solaris DDI
ddi_get_parent	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
ddi_getlongprop_buf	Solaris DDI
ddi_getprop	Solaris DDI
ddi_getproplen	Solaris DDI
ddi_getw	Solaris DDI
ddi_intr_hilevel	Solaris DDI
ddi_io_get16	Solaris DDI
ddi_io_get32	Solaris DDI

<b>Routine</b>	<b>Type</b>
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
ddi_map_regs	Solaris DDI
ddi_mapdev	Solaris DDI
ddi_mapdev_intercept	Solaris DDI
ddi_mapdev_nointercept	Solaris DDI

<b>Routine</b>	<b>Type</b>
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
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

<b>Routine</b>	<b>Type</b>
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_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_poke16	Solaris DDI
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

<b>Routine</b>	<b>Type</b>
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
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_intr	Solaris DDI
ddi_remove_minor_node	Solaris DDI

<b>Routine</b>	<b>Type</b>
ddi_remove_softintr	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
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_trigger_softintr	Solaris DDI

## Intro(9F)

<b>Routine</b>	<b>Type</b>
ddi_umem_alloc	Solaris DDI
ddi_umem_free	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
devmap_setup	Solaris DDI
devmap_umem_setup	Solaris DDI
devmap_unload	Solaris DDI
disksort	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
getmajor	DDI/DKI
getminor	DDI/DKI
getrbuf	DDI/DKI
hat_getkpfnum	DKI only
inb	Solaris x86 DDI
inl	Solaris x86 DDI
inw	Solaris x86 DDI

<b>Routine</b>	<b>Type</b>
kmem_alloc	DDI/DKI
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
makecom_g0	Solaris DDI
makecom_g0_s	Solaris DDI
makecom_g1	Solaris DDI
makecom_g5	Solaris DDI
makedevice	DDI/DKI
max	DDI/DKI
min	DDI/DKI
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

<b>Routine</b>	<b>Type</b>
mutex_owned	Solaris DDI
mutex_tryenter	Solaris DDI
nochpoll	Solaris DDI
nodev	DDI/DKI
nulldev	DDI/DKI
numtos	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
pci_config_putl	Solaris DDI
pci_config_putw	Solaris DDI
pci_config_setup	Solaris DDI
pci_config_teardown	Solaris DDI
physio	Solaris DDI
pollwakeup	DDI/DKI
proc_ref	Solaris DDI
proc_signal	Solaris DDI

<b>Routine</b>	<b>Type</b>
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
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

## Intro(9F)

<b>Routine</b>	<b>Type</b>
scsi_errmsg	Solaris DDI
scsi_free_consistent_buf	Solaris DDI
scsi_hba_attach	Solaris DDI
scsi_hba_attach_setup	Solaris DDI
scsi_hba_detach	Solaris DDI
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_slave	Solaris DDI
scsi_sname	Solaris DDI

<b>Routine</b>	<b>Type</b>
scsi_sync_pkt	Solaris DDI
scsi_transport	Solaris DDI
scsi_unprobe	Solaris DDI
scsi_unslave	Solaris DDI
sema_destroy	Solaris DDI
sema_init	Solaris DDI
sema_p	Solaris DDI
sema_p_sig	Solaris DDI
sema_try	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
swab	DDI/DKI
timeout	DDI/DKI
uiomove	DDI/DKI
untimeout	DDI/DKI
ureadc	DDI/DKI
uwritec	DDI/DKI
va_arg	Solaris DDI
va_end	Solaris DDI
va_start	Solaris DDI
vcmn_err	DDI/DKI
vsprintf	Solaris DDI

Intro(9F)

**SEE ALSO** | Intro(9E), mutex(9F)

# Kernel Functions for Drivers

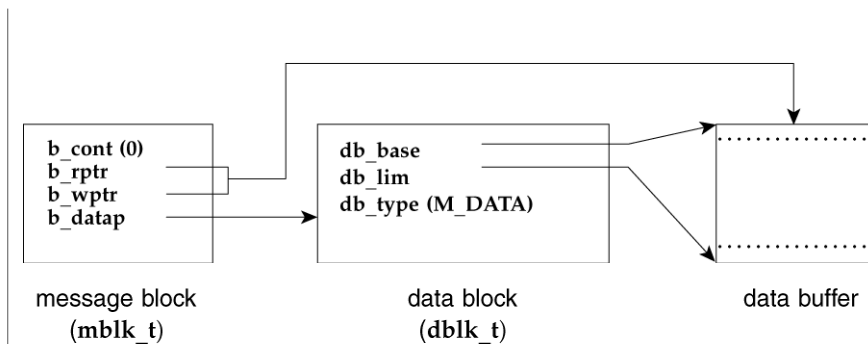
---

## adjmsg(9F)

<b>NAME</b>	adjmsg – trim bytes from a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>adjmsg</b>(mblk_t *mp, ssize_t len);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>mp</i> Pointer to the message to be trimmed. <i>len</i> The number of bytes to be removed.
<b>DESCRIPTION</b>	<p>The <code>adjmsg()</code> function removes bytes from a message. <code> len </code> (the absolute value of <code>len</code>) specifies the number of bytes to be removed. The <code>adjmsg()</code> function only trims bytes across message blocks of the same type.</p> <p>The <code>adjmsg()</code> function finds the maximal leading sequence of message blocks of the same type as that of <code>mp</code> and starts removing bytes either from the head of that sequence or from the tail of that sequence. If <code>len</code> is greater than 0, <code>adjmsg()</code> removes bytes from the start of the first message block in that sequence. If <code>len</code> is less than 0, it removes bytes from the end of the last message block in that sequence.</p> <p>The <code>adjmsg()</code> function fails if <code> len </code> is greater than the number of bytes in the maximal leading sequence it finds.</p> <p>The <code>adjmsg()</code> 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 <code> len </code>.</p>
<b>RETURN VALUES</b>	The <code>adjmsg()</code> function returns: 1           Successful completion. 0           An error occurred.
<b>CONTEXT</b>	The <code>adjmsg()</code> function can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>STREAMS Programming Guide</i>

<b>NAME</b>	allocb – allocate a message block														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *allocb(size_t size, uint_t pri);</pre>														
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).														
<b>DESCRIPTION</b>	<p>allocb() tries to allocate a STREAMS message block. Buffer allocation fails only when the system is out of memory. If no buffer is available, the bufcall(9F) function can help a module recover from an allocation failure.</p> <p>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.</p> <p>The fields in the mblk_t structure are initialized as follows:</p> <table border="0"> <tr> <td>b_cont</td> <td>set to NULL</td> </tr> <tr> <td>b_rptr</td> <td>points to the beginning of the data buffer</td> </tr> <tr> <td>b_wptr</td> <td>points to the beginning of the data buffer</td> </tr> <tr> <td>b_datap</td> <td>points to the dblk_t structure</td> </tr> </table> <p>The fields in the dblk_t structure are initialized as follows:</p> <table border="0"> <tr> <td>db_base</td> <td>points to the first byte of the data buffer</td> </tr> <tr> <td>db_lim</td> <td>points to the last byte + 1 of the buffer</td> </tr> <tr> <td>db_type</td> <td>set to M_DATA</td> </tr> </table> <p>The following figure identifies the data structure members that are affected when a message block is allocated.</p>	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	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
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														
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														

## allocb(9F)



**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** `allocb()` can be called from user or interrupt 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)

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

**SEE ALSO** RD(9F), bufcall(9F), 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.

anocancel(9F)

<b>NAME</b>	anocancel – prevent cancellation of asynchronous I/O request
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>anocancel</b>( );</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	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).
<b>RETURN VALUES</b>	anocancel() returns ENXIO.
<b>SEE ALSO</b>	aread(9E), awrite(9E), aphysio(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	aphysio – perform asynchronous physical I/O
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt; #include &lt;sys/uio.h&gt; #include &lt;sys/aio_req.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>aphysio</b>(int *stratstruct buf *, int *cancelstruct buf *, dev_t             dev, int rw, void *mincntstruct buf *, struct aio_req *aio_reqp);</pre>
<b>PARAMETERS</b>	<p><i>strat</i>                    Pointer to device strategy routine.</p> <p><i>cancel</i>                   Pointer to driver cancel routine. Used to cancel a submitted request. The driver must pass the address of the function <code>anocancel(9F)</code> because cancellation is not supported.</p> <p><i>dev</i>                      The device number.</p> <p><i>rw</i>                        Read/write flag. This is either <code>B_READ</code> when reading from the device or <code>B_WRITE</code> when writing to the device.</p> <p><i>mincnt</i>                   Routine which bounds the maximum transfer unit size.</p> <p><i>aio_reqp</i>                 Pointer to the <code>aio_req(9S)</code> structure which describes the user I/O request.</p>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	<p><code>aphysio()</code> performs asynchronous I/O operations between the device and the address space described by <code>aio_reqp→aio_uio</code>.</p> <p>Prior to the start of the transfer, <code>aphysio()</code> 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, <i>strat</i>, is then called one or more times to perform the physical I/O operations. <code>aphysio()</code> does not wait for each transfer to complete, but returns as soon as the necessary requests have been made.</p> <p><code>aphysio()</code> calls <i>mincnt</i> 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 <i>mincnt</i> routine should call <code>aphysio()</code> with <code>minphys(9F)</code>. <code>minphys(9F)</code> is the system <i>mincnt</i> routine. <code>minphys(9F)</code> ensures the transfer size does not exceed any system limits.</p> <p>If a driver supplies a local <i>mincnt</i> routine, this routine should perform the following actions:</p> <ul style="list-style-type: none"> <li>■ If <code>bp→b_bcount</code> exceeds a device limit, set <code>bp→b_bcount</code> to a value supported by the device.</li> <li>■ Call <code>minphys(9F)</code> to ensure that the driver does not circumvent additional system limits.</li> </ul>
<b>RETURN VALUES</b>	<code>aphysio()</code> returns:

## aphysio(9F)

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.

<b>NAME</b>	ASSERT, assert – expression verification
<b>SYNOPSIS</b>	<pre>#include &lt;sys/debug.h&gt;  void <b>ASSERT</b>(EX) ;</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>EX</i> boolean expression.
<b>DESCRIPTION</b>	ASSERT() is a macro which checks to see if the expression <i>EX</i> 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.
<b>CONTEXT</b>	ASSERT() can be used from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

## backq(9F)

<b>NAME</b>	backq – get pointer to the queue behind the current queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  queue_t *backq(queue_t *cq);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>cq</i> The pointer to the current queue. queue_t is an alias for the queue(9S) structure.
<b>DESCRIPTION</b>	backq() returns a pointer to the queue preceding <i>cq</i> (the current queue). If <i>cq</i> is a read queue, backq() returns a pointer to the queue downstream from <i>cq</i> , unless it is the stream end. If <i>cq</i> is a write queue, backq() returns a pointer to the next queue upstream from <i>cq</i> , unless it is the stream head.
<b>RETURN VALUES</b>	If successful, backq() returns a pointer to the queue preceding the current queue. Otherwise, it returns NULL.
<b>CONTEXT</b>	backq() can be called from user or interrupt context.
<b>SEE ALSO</b>	queue(9S) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	bcanput – test for flow control in specified priority band
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>bcanput</b>(queue_t *q, unsigned char pri);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>q</i>            Pointer to the message queue.</p> <p><i>pri</i>          Message priority.</p>
<b>DESCRIPTION</b>	<p>bcanput ( ) searches through the stream (starting at <i>q</i>) 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 <i>pri</i> in the queue.</p> <p>If <i>pri</i> is 0, bcanput ( ) is equivalent to a call with canput(9F).</p> <p>canputnext ( <i>q</i> ) and bcanputnext ( <i>q</i>, <i>pri</i> ) should always be used in preference to canput ( <i>q</i>→<i>q_next</i> ) and bcanput ( <i>q</i>→<i>q_next</i>, <i>pri</i> ) respectively.</p>
<b>RETURN VALUES</b>	<p>1            If a message of priority <i>pri</i> can be placed on the queue.</p> <p>0            If the priority band is full.</p>
<b>CONTEXT</b>	bcanput ( ) can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>bcanputnext(9F), canput(9F), canputnext(9F), putbq(9F), putnext(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>
<b>WARNINGS</b>	Drivers are responsible for both testing a queue with bcanput ( ) and refraining from placing a message on the queue if bcanput ( ) fails.

## bcmp(9F)

<b>NAME</b>	bcmp – compare two byte arrays
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  int <b>bcmp</b>(const void *s1, const void *s2, size_t len);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>s1</i>            Pointer to the first character string.</p> <p><i>s2</i>            Pointer to the second character string.</p> <p><i>len</i>           Number of bytes to be compared.</p>
<b>DESCRIPTION</b>	bcmp() compares two byte arrays of length <i>len</i> .
<b>RETURN VALUES</b>	bcmp() returns 0 if the arrays are identical, or 1 if they are not.
<b>CONTEXT</b>	bcmp() can be called from user or interrupt context.
<b>SEE ALSO</b>	strcmp(9F) <i>Writing Device Drivers</i>
<b>NOTES</b>	Unlike strcmp(9F), bcmp() does not terminate when it encounters a null byte.

<b>NAME</b>	bcopy – copy data between address locations in the kernel
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt;  void <b>bcopy</b>(const void *from, void *to, size_t bcount);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>from</i>        Source address from which the copy is made.</p> <p><i>to</i>            Destination address to which copy is made.</p> <p><i>bcount</i>      The number of bytes moved.</p>
<b>DESCRIPTION</b>	<p>bcopy () copies <i>bcount</i> 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.</p> <p>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.</p>
<b>CONTEXT</b>	bcopy () can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Copying data between address locations in the kernel:</p> <p>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.</p> <pre> 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-&gt;b_flags &amp; B_READ)               /* if read request, copy data */ 6                                        /* from RAM disk data block */ 7                                        /* to system buffer */ 8        bcopy(&amp;ramdblks[bp-&gt;b_blkno][0], bp-&gt;b_un.b_addr, 9            bp-&gt;b_bcount); 10 11 else                                    /* else write request, */ 12                                        /* copy data from a */ 13                                        /* system buffer to RAM disk */ 14                                        /* data block */ 15        bcopy(bp-&gt;b_un.b_addr, &amp;ramdblks[bp-&gt;b_blkno][0], 16            bp-&gt;b_bcount);</pre>
<b>SEE ALSO</b>	copyin(9F), copyout(9F)

bcopy(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.

<b>NAME</b>	bioclone – clone another buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>bp</i> Pointer to the buf(9S) structure describing the original I/O request.</p> <p><i>off</i> Offset within original I/O request where new I/O request should start.</p> <p><i>len</i> Length of the I/O request.</p> <p><i>dev</i> Device number.</p> <p><i>blkno</i> Block number on device.</p> <p><i>iodone</i> Specific biodone(9F) routine.</p> <p><i>bp_mem</i> Pointer to a buffer structure to be filled in or NULL.</p> <p><i>sleepflag</i> 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.</p>
<b>DESCRIPTION</b>	<p>bioclone() returns an initialized buffer to perform I/O to a portion of another buffer. The new buffer will be set up to perform I/O to the range within the original I/O request specified by the parameters <i>off</i> and <i>len</i>. An offset 0 starts the new I/O request at the same address as the original request. <i>off</i> + <i>len</i> must not exceed <i>b_bcount</i>, the length of the original request. The device number <i>dev</i> specifies the device to which the buffer is to perform I/O. <i>blkno</i> is the block number on device. It will be assigned to the <i>b_blkno</i> field of the cloned buffer structure. <i>iodone</i> lets the driver identify a specific biodone(9F) routine to be called by the driver when the I/O is complete. <i>bp_mem</i> determines from where the space for the buffer should be allocated. If <i>bp_mem</i> is NULL, bioclone() will allocate a new buffer using getrbuf(9F). If <i>sleepflag</i> is set to KM_SLEEP, the driver may sleep until space is freed up. If <i>sleepflag</i> 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 <i>bp_mem</i> is not NULL, it will be used as the space for the buffer structure. The driver has to ensure that <i>bp_mem</i> is initialized properly either using getrbuf(9F) or bioinit(9F).</p> <p>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.</p>

## bioclone(9F)

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

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

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

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

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

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

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

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

**SEE ALSO** `biodone(9F)`, `bp_mapin(9F)`, `freerbuf(9F)`, `getrbuf(9F)`, `buf(9S)`

*Writing Device Drivers*

## biodone(9F)

<b>NAME</b>	biodone – release buffer after buffer I/O transfer and notify blocked threads
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt;  void <b>biodone</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to a buf(9S) structure.
<b>DESCRIPTION</b>	<p>biodone() notifies blocked processes waiting for the I/O to complete, sets the B_DONE flag in the b_flags field of the buf(9S) structure, and releases the buffer if the I/O is asynchronous. biodone() is called by either the driver interrupt or strategy(9E) routines when a buffer I/O request is complete.</p> <p>biodone() provides the capability to call a completion routine if <i>bp</i> 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.</p>
<b>CONTEXT</b>	biodone() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p>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).</p> <pre>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-&gt;b_blkno; /* get block number */ 9 10     if ((blkno &lt; 0)    (blkno &gt;= 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) &amp;&amp; (bp-&gt;b_flags &amp; B_READ)) { 17             /* 18              * If read is for block beyond RAM disk 19              * limits, mark EOF condition. 20              */ 21             bp-&gt;b_resid = bp-&gt;b_bcount; /* compute return value */ 22 23             } else { /* I/O attempt is beyond */</pre>

## biodone(9F)

```
24         bp->b_error = ENXIO;    /* limits of RAM disk */
25         bp->b_flags |= B_ERROR; /* return error */
26     }
27     biodone(bp);    /* mark I/O complete (B_DONE) */
28     /*
29     * Wake any processes awaiting this I/O
30     * or release buffer for asynchronous
31     * (B_ASYNC) request.
32     */
33     return (0);
34 }
```

**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.

## bioerror(9F)

<b>NAME</b>	bioerror – indicate error in buffer header
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt; #include &lt;sys/ddi.h&gt;  void <b>bioerror</b>(struct buf *bp, int error);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>bp</i>            Pointer to the buf(9S) structure describing the transfer.</p> <p><i>error</i>         Error number to be set, or zero to clear an error indication.</p>
<b>DESCRIPTION</b>	<p>If <i>error</i> is non-zero, bioerror() indicates an error has occurred in the buf(9S) structure. A subsequent call to geterror(9F) will return <i>error</i>.</p> <p>If <i>error</i> is 0, the error indication is cleared and a subsequent call to geterror(9F) will return 0.</p>
<b>CONTEXT</b>	bioerror() can be called from any context.
<b>SEE ALSO</b>	strategy(9E), geterror(9F), getrbuf(9F), buf(9S)

<b>NAME</b>	biofini – uninitialized a buffer structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void<b>biofini</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buffer header structure.
<b>DESCRIPTION</b>	The <code>biofini()</code> function uninitialized a <code>buf(9S)</code> structure. If a buffer structure has been allocated and initialized using <code>kmem_alloc(9F)</code> and <code>bioinit(9F)</code> it needs to be uninitialized using <code>biofini()</code> before calling <code>kmem_free(9F)</code> . It is not necessary to call <code>biofini()</code> before freeing a buffer structure using <code>freerbuf(9F)</code> because <code>freerbuf()</code> will call <code>biofini()</code> directly.
<b>CONTEXT</b>	The <code>biofini()</code> function can be called from any context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <code>biofini()</code></p> <pre>struct buf *bp = kmem_alloc(biosize(), KM_SLEEP); bioinit(bp); /* use buffer */ biofini(bp); kmem_free(bp, biosize());</pre>
<b>SEE ALSO</b>	<p><code>bioinit(9F)</code>, <code>bioreset(9F)</code>, <code>biosize(9F)</code>, <code>freerbuf(9F)</code>, <code>kmem_alloc(9F)</code>, <code>kmem_free(9F)</code>, <code>buf(9S)</code></p> <p><i>Writing Device Drivers</i></p>

## bioinit(9F)

<b>NAME</b>	bioinit – initialize a buffer structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void<b>bioinit</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buffer header structure.
<b>DESCRIPTION</b>	The <code>bioinit()</code> function initializes a <code>buf(9S)</code> structure. A buffer structure contains state information which has to be initialized if the memory for the buffer was allocated using <code>kmem_alloc(9F)</code> . This is not necessary for a buffer allocated using <code>getrbuf(9F)</code> because <code>getrbuf()</code> will call <code>bioinit()</code> directly.
<b>CONTEXT</b>	The <code>bioinit()</code> function can be called from any context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <code>bioinit()</code></p> <pre>struct buf *bp = kmem_alloc(biosize(), KM_SLEEP); bioinit(bp); /* use buffer */</pre>
<b>SEE ALSO</b>	<code>biofini(9F)</code> , <code>bioreset(9F)</code> , <code>biosize(9F)</code> , <code>getrbuf(9F)</code> , <code>kmem_alloc(9F)</code> , <code>buf(9S)</code> <i>Writing Device Drivers</i>

<b>NAME</b>	biomodified – check if a buffer is modified
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int biomodified(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buffer header structure.
<b>DESCRIPTION</b>	<p>The <code>biomodified()</code> function returns status to indicate if the buffer is modified. The <code>biomodified()</code> function is only supported for paged- I/O request, that is the <code>B_PAGEIO</code> flag must be set in the <code>b_flags</code> field of the <code>buf(9S)</code> structure. The <code>biomodified()</code> 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 <code>biomodified()</code> function can be used to detect any modifications to the memory pages while I/O is in progress.</p> <p>A device driver can use <code>biomodified()</code> for disk mirroring. An application is allowed to <code>mmap</code> 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 <code>biomodified()</code> 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.</p>
<b>RETURN VALUES</b>	<p>The <code>biomodified()</code> function returns the following values:</p> <ul style="list-style-type: none"> <li>1            Buffer is modified.</li> <li>0            Buffer is not modified.</li> <li>-1          Buffer is not used for paged I/O request.</li> </ul>
<b>CONTEXT</b>	<code>biomodified()</code> can be called from any context.
<b>SEE ALSO</b>	<p><code>bp_mapin(9F)</code>, <code>buf(9S)</code></p> <p><i>Writing Device Drivers</i></p>

bioreset(9F)

<b>NAME</b>	bioreset – reuse a private buffer header after I/O is complete
<b>SYNOPSIS</b>	<pre>#include &lt;sys/buf.h&gt; #include &lt;sys/ddi.h&gt;  void <b>bioreset</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buf(9S) structure.
<b>DESCRIPTION</b>	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).
<b>CONTEXT</b>	bioreset () can be called from any context.
<b>SEE ALSO</b>	strategy(9E), bioinit(9F), biofini(9F), freerbuf(9F), getrbuf(9F), kmem_alloc(9F), kmem_free(9F), buf(9S)
<b>NOTES</b>	<i>bp</i> must not describe a transfer in progress.

<b>NAME</b>	biosize – returns size of a buffer structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  size_t biosize(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	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.
<b>CONTEXT</b>	The biosize() function can be called from any context.
<b>SEE ALSO</b>	biofini(9F), bioinit(9F), getrbuf(9F), kmem_alloc(9F), buf(9S) <i>Writing Device Drivers</i>

## biowait(9F)

<b>NAME</b>	biowait – suspend processes pending completion of block I/O				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt;  int <b>biowait</b>(struct buf *bp);</pre>				
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).				
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buf structure describing the transfer.				
<b>DESCRIPTION</b>	<p>Drivers allocating their own buf structures with <code>getrbuf(9F)</code> can use the <code>biowait()</code> function to suspend the current thread and wait for completion of the transfer.</p> <p>Drivers must call <code>biodone(9F)</code> when the transfer is complete to notify the thread blocked by <code>biowait()</code>. <code>biodone()</code> is usually called in the interrupt routine.</p>				
<b>RETURN VALUES</b>	<table><tr><td>0</td><td>Upon success</td></tr><tr><td>non-zero</td><td>Upon I/O failure. <code>biowait()</code> calls <code>geterror(9F)</code> to retrieve the error number which it returns.</td></tr></table>	0	Upon success	non-zero	Upon I/O failure. <code>biowait()</code> calls <code>geterror(9F)</code> to retrieve the error number which it returns.
0	Upon success				
non-zero	Upon I/O failure. <code>biowait()</code> calls <code>geterror(9F)</code> to retrieve the error number which it returns.				
<b>CONTEXT</b>	<code>biowait()</code> can be called from user context only.				
<b>SEE ALSO</b>	<code>biodone(9F)</code> , <code>geterror(9F)</code> , <code>getrbuf(9F)</code> , <code>buf(9S)</code> <i>Writing Device Drivers</i>				

<b>NAME</b>	bp_mapin – allocate virtual address space
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt;  void <b>bp_mapin</b>(struct buf *bp) ;</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buffer header structure.
<b>DESCRIPTION</b>	<p>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-&gt;b_un.b_addr field of the buf(9S) structure. Virtual address space is then deallocated using the bp_mapout(9F) function.</p> <p>If a null page list is encountered, bp_mapin() returns without allocating space and no mapping is performed.</p>
<b>CONTEXT</b>	bp_mapin() can be called from user and kernel contexts.
<b>SEE ALSO</b>	bp_mapout(9F), buf(9S)
	<i>Writing Device Drivers</i>

## bp\_mapout(9F)

<b>NAME</b>	bp_mapout – deallocate virtual address space
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt;  void <b>bp_mapout</b> (struct buf *bp) ;</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buffer header structure.
<b>DESCRIPTION</b>	<p>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.</p>
<b>CONTEXT</b>	bp_mapout () can be called from user context only.
<b>SEE ALSO</b>	strategy(9E), bp_mapin(9F), buf(9S) <i>Writing Device Drivers</i>

<b>NAME</b>	btop – convert size in bytes to size in pages (round down)
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  unsigned long <b>btop</b>(unsigned long <i>numbytes</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>numbytes</i> Number of bytes.
<b>DESCRIPTION</b>	<b>btop</b> ( ) 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 <b>btop</b> (4096) returns 2, and <b>btop</b> (4097) returns 2 as well. <b>btop</b> (0) returns 0.
<b>RETURN VALUES</b>	The return value is always the number of pages. There are no invalid input values, and therefore no error return values.
<b>CONTEXT</b>	<b>btop</b> ( ) can be called from user or interrupt context.
<b>SEE ALSO</b>	<b>btopr</b> (9F), <b>ddi_btop</b> (9F), <b>ptob</b> (9F) <i>Writing Device Drivers</i>

## btopr(9F)

<b>NAME</b>	btopr – convert size in bytes to size in pages (round up)
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  unsigned long <b>btopr</b>(unsigned long <i>numbytes</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>numbytes</i> Number of bytes.
<b>DESCRIPTION</b>	<code>btopr()</code> 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 <code>btopr(4096)</code> returns 2, and <code>btopr(4097)</code> returns 3.
<b>RETURN VALUES</b>	The return value is always the number of pages. There are no invalid input values, and therefore no error return values.
<b>CONTEXT</b>	<code>btopr()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>btop(9F)</code> , <code>ddi_btopr(9F)</code> , <code>ptob(9F)</code> <i>Writing Device Drivers</i>

<b>NAME</b>	bufcall – call a function when a buffer becomes available
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/stream.h&gt;  bufcall_id_t <b>bufcall</b>(size_t <i>size</i>, uint_t <i>pri</i>, void *<i>func</i>void *<i>arg</i>,                     void *<i>arg</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>size</i>        Number of bytes required for the buffer.</p> <p><i>pri</i>         Priority of the allocb(9F) allocation request (not used).</p> <p><i>func</i>        Function or driver routine to be called when a buffer becomes available.</p> <p><i>arg</i>         Argument to the function to be called when a buffer becomes available.</p>
<b>DESCRIPTION</b>	bufcall() serves as a timeout(9F) call of indeterminate length. When a buffer allocation request fails, bufcall() can be used to schedule the routine <i>func</i> , to be called with the argument <i>arg</i> when a buffer becomes available. <i>func</i> may call allocb() or it may do something else.
<b>RETURN VALUES</b>	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, <i>func</i> is never called and 0 is returned.
<b>CONTEXT</b>	bufcall() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Calling a function when a buffer becomes available:</p> <p>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</p> <p>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).</p> <p>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 <i>q</i> once a buffer of the specified size (sizeof (struct hdr)) becomes available. If it does, qenable(9F) will put <i>q</i> 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.</p>

## bufcall(9F)

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

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

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

```
1  struct hdr {
2      unsigned int h_size;
3      int          h_version;
4  };
5
6  void xxxsrv(q)
7      queue_t *q;
8  {
9      mblk_t *bp;
10     mblk_t *mp;
11     struct hdr *hp;
12
13     while ((mp = getq(q)) != NULL) { /* get next message */
14         if (mp->b_datap->db_type >= QPCTL || /* if high priority */
15             canputnext(q)) { /* normal & can be passed */
16             if (mp->b_datap->db_type != M_DATA)
17                 putnext(q, mp); /* send all but M_DATA */
18             else {
19                 bp = allocb(sizeof(struct hdr), BPRI_LO);
20                 if (bp == NULL) { /* if unsuccessful */
21                     putbq(q, mp); /* put it back */
22                     if (!bufcall(sizeof(struct hdr), BPRI_LO,
23                                 qenable, q)) /* try to reschedule */
24                         timeout(qenable, q, drv_usectohz(500000));
25                     return (0);
26                 }
27                 hp = (struct hdr *)bp->b_wptr;
28                 hp->h_size = msgdsize(mp); /* initialize header */
29                 hp->h_version = 1;
30                 bp->b_wptr += sizeof(struct hdr);
31                 bp->b_datap->db_type = M_PROTO; /* make M_PROTO */
32                 bp->b_cont = mp; /* link it */
33                 putnext(q, bp); /* pass it on */
34             }
35         } else { /* normal priority, canputnext failed */
36             putbq(q, mp); /* put back on the message queue */
37             return (0);
38         }
39     }
40     return (0);
41 }
```

**SEE ALSO** `srv(9E)`, `allocb(9F)`, `canputnext(9F)`, `esballoc(9F)`, `esbcall(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)`.

## bzero(9F)

<b>NAME</b>	bzero – clear memory for a given number of bytes
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  void <b>bzero</b>(void *addr, size_t bytes);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>addr</i> Starting virtual address of memory to be cleared.</p> <p><i>bytes</i> The number of bytes to clear starting at <i>addr</i>.</p>
<b>DESCRIPTION</b>	bzero() clears a contiguous portion of memory by filling it with zeros.
<b>CONTEXT</b>	bzero() can be called from user or interrupt context.
<b>SEE ALSO</b>	bcopy(9F), clrbuf(9F), kmem_zalloc(9F) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	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.

<b>NAME</b>	canput – test for room in a message queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; int <b>canput</b> (queue_t *q) ;</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>q</i> Pointer to the message queue.
<b>DESCRIPTION</b>	<p>canput () searches through the stream (starting at <i>q</i>) 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.</p> <p>canputnext (<i>q</i>) and bcanputnext (<i>q, pri</i>) should always be used in preference to canput (<i>q→q_next</i>) and bcanput (<i>q→q_next, pri</i>) respectively.</p>
<b>RETURN VALUES</b>	<p>1            If the message queue is not full.</p> <p>0            If the queue is full.</p>
<b>CONTEXT</b>	canput () can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>bcanput(9F), bcanputnext(9F), canputnext(9F), putbq(9F), putnext(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>
<b>WARNINGS</b>	Drivers are responsible for both testing a queue with canput () and refraining from placing a message on the queue if canput () fails.

## canputnext(9F)

<b>NAME</b>	canputnext, bcanputnext – test for room in next module’s message queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>canputnext</b> (queue_t *q) ; int <b>bcanputnext</b> (queue_t *q, unsigned char pri) ;</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i> Pointer to a message queue belonging to the invoking module.</p> <p><i>pri</i> Minimum priority level.</p>
<b>DESCRIPTION</b>	<p>The invocation <code>canputnext (q) ;</code> is an atomic equivalent of the <code>canput (q→q_next) ;</code> routine. That is, the STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing <i>q</i> through its <code>q_next</code> field and then invoking <code>canput(9F)</code> proceeds without interference from other threads.</p> <p><code>bcanputnext (q, pri) ;</code> is the equivalent of the <code>bcanput (q→q_next, pri) ;</code> routine.</p> <p><code>canputnext (q) ;</code> and <code>bcanputnext (q, pri) ;</code> should always be used in preference to <code>canput (q→q_next) ;</code> and <code>bcanput (q→q_next, pri) ;</code> respectively.</p> <p>See <code>canput(9F)</code> and <code>bcanput(9F)</code> for further details.</p>
<b>RETURN VALUES</b>	<p>1 If the message queue is not full.</p> <p>0 If the queue is full.</p>
<b>CONTEXT</b>	<code>canputnext ()</code> and <code>bcanputnext ()</code> can be called from user or interrupt context.
<b>WARNINGS</b>	Drivers are responsible for both testing a queue with <code>canputnext ()</code> or <code>bcanputnext ()</code> and refraining from placing a message on the queue if the queue is full.
<b>SEE ALSO</b>	<p><code>bcanput(9F)</code>, <code>canput(9F)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

<b>NAME</b>	clrbuf – erase the contents of a buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt;  void <b>clrbuf</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buf(9S) structure.
<b>DESCRIPTION</b>	clrbuf() zeros a buffer and sets the <i>b_resid</i> member of the buf(9S) structure to 0. Zeros are placed in the buffer starting at <i>bp</i> → <i>b_un.b_addr</i> for a length of <i>bp</i> → <i>b_bcount</i> bytes. <i>b_un.b_addr</i> and <i>b_bcount</i> are members of the buf(9S) data structure.
<b>CONTEXT</b>	clrbuf() can be called from user or interrupt context.
<b>SEE ALSO</b>	getrbuf(9F), buf(9S) <i>Writing Device Drivers</i>

## cmn\_err(9F)

<b>NAME</b>	cmn_err, vcmn_err – display an error message or panic the system						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/cmn_err.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>cmn_err</b>(int level, char *format...);  #include &lt;sys/varargs.h&gt;  void <b>vcmn_err</b>(int level, char *format, va_list ap);</pre>						
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).						
<b>PARAMETERS</b>							
<b>cmn_err()</b>	<table><tr><td><i>level</i></td><td>A constant indicating the severity of the error condition.</td></tr><tr><td><i>format</i></td><td>The message to be displayed.</td></tr></table>	<i>level</i>	A constant indicating the severity of the error condition.	<i>format</i>	The message to be displayed.		
<i>level</i>	A constant indicating the severity of the error condition.						
<i>format</i>	The message to be displayed.						
<b>vcmn_err()</b>	vcmn_err() takes <i>level</i> and <i>format</i> as described for cmn_err(), but its third argument is different: <table><tr><td><i>ap</i></td><td>The variable argument list passed to the function.</td></tr></table>	<i>ap</i>	The variable argument list passed to the function.				
<i>ap</i>	The variable argument list passed to the function.						
<b>DESCRIPTION</b>							
<b>cmn_err()</b>	<p>cmn_err() displays a specified message on the console. cmn_err() can also panic the system. When the system panics, it attempts to save recent changes to data, display a “panic message” on the console, attempt to write a core file, and halt system processing. See the CE_PANIC level below.</p> <p><i>level</i> is a constant indicating the severity of the error condition. The four severity levels are:</p> <table><tr><td>CE_CONT</td><td>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().</td></tr><tr><td>CE_NOTE</td><td>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.</td></tr><tr><td>CE_WARN</td><td>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.</td></tr></table>	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_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:

<code>d,D,o,O,x,X,u</code>	The integer argument is converted to signed decimal ( <code>d</code> , <code>D</code> ), unsigned octal ( <code>o</code> , <code>O</code> ), unsigned hexadecimal ( <code>x</code> , <code>X</code> ), or unsigned decimal ( <code>u</code> ), respectively, and displayed. The letters <code>abcdef</code> are used for <code>x</code> and <code>X</code> conversion.
<code>c</code>	The character value of the argument is displayed.
<code>b</code>	The <code>%b</code> conversion specification allows bit values to be displayed meaningfully. Each <code>%b</code> 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 <code>^0'</code> ) 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.
<code>p</code>	The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to <code>%lx</code> . To avoid lint warnings, cast pointers to type <code>void *</code> when using the <code>%p</code> format specifier.

## cmn\_err(9F)

	<p>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 <code>NULL</code>, the string <code>&lt;null string&gt;</code> is used in its place.</p> <p>%            Copy a %; no argument is converted.</p> <p>The first character in <i>format</i> affects where the message will be written:</p> <p>!            The message goes only to the system log.</p> <p>^            The message goes only to the console.</p> <p>?            If <i>level</i> is also <code>CE_CONT</code>, 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 <code>kernel(1M)</code>. If neither condition is met, the '?' character has no effect and is simply ignored.</p> <p>Refer to <code>syslogd(1M)</code> to determine where the system log is written.</p> <p><code>cmn_err()</code> appends a <code>\n</code> to each <i>format</i>, except when <i>level</i> is <code>CE_CONT</code>.</p>
<b>vcmn_err()</b>	<p><code>vcmn_err()</code> is identical to <code>cmn_err()</code> except that its last argument, <i>ap</i>, is a pointer to a variable list of arguments. <i>ap</i> contains the list of arguments used by the conversion specifications in <i>format</i>. <i>ap</i> must be initialized by calling <code>va_start(9F)</code>. <code>va_end(9F)</code> is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by <code>va_start(9F)</code> and <code>va_end(9F)</code>, are possible.</p>
<b>RETURN VALUES</b>	<p>None. However, if an unknown <i>level</i> is passed to <code>cmn_err()</code>, the following panic error message is displayed:</p> <pre>panic: unknown level in cmn_err (level=level, msg=format)</pre>
<b>CONTEXT</b>	<p><code>cmn_err()</code> can be called from user, kernel, interrupt, or high-level interrupt context.</p>
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <code>cmn_err()</code></p> <p>This first example shows how <code>cmn_err()</code> 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).</p> <pre>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 };</pre>

**EXAMPLE 1** Using `cmn_err()` (Continued)

```

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

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

```

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

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

```

## cmn\_err(9F)

### EXAMPLE 4 Error Routine (Continued)

```
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);
}
```

**SEE ALSO** dmesg(1M), kernel(1M), printf(3C), ddi\_binding\_name(9F), sprintf(9F), va\_arg(9F), va\_end(9F), va\_start(9F), vsprintf(9F)

*Writing Device Drivers*

**WARNINGS** `cmn_err()` with the `CE_CONT` argument can be used by driver developers as a driver code debugging tool. However, using `cmn_err()` in this capacity can change system timing characteristics.

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

<b>NAME</b>	condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ksynch.h&gt;  void <b>cv_init</b>(kcondvar_t *cvp, char *name, kcv_type_t type, void              *arg);  void <b>cv_destroy</b>(kcondvar_t *cvp);  void <b>cv_wait</b>(kcondvar_t *cvp, kmutex_t *mp);  void <b>cv_signal</b>(kcondvar_t *cvp);  void <b>cv_broadcast</b>(kcondvar_t *cvp);  int <b>cv_wait_sig</b>(kcondvar_t *cvp, kmutex_t *mp);  clock_t <b>cv_timedwait</b>(kcondvar_t *cvp, kmutex_t *mp, clock_t                       timeout);  clock_t <b>cv_timedwait_sig</b>(kcondvar_t *cvp, kmutex_t *mp, clock_t                           timeout);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>cvp</i>            A pointer to an abstract data type kcondvar_t.</p> <p><i>mp</i>             A pointer to a mutual exclusion lock (kmutex_t), initialized by mutex_init(9F) and held by the caller.</p> <p><i>name</i>           Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they're a waste of kernel memory.)</p> <p><i>type</i>           The constant CV_DRIVER.</p> <p><i>arg</i>            A type-specific argument, drivers should pass arg as NULL.</p> <p><i>timeout</i>        A time, in absolute ticks since boot, when cv_timedwait() or cv_timedwait_sig() should return.</p>
<b>DESCRIPTION</b>	<p>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().</p> <p>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</p>

## condvar(9F)

point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with `cv_signal()` or `cv_broadcast()`, and then release the mutex.

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

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

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

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

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

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

### RETURN VALUES

0	For <code>cv_wait_sig()</code> and <code>cv_timedwait_sig()</code> indicates that the condition was not necessarily signaled and the function returned because a signal (as in <code>kill(2)</code> ) was pending.
-1	For <code>cv_timedwait()</code> and <code>cv_timedwait_sig()</code> indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.
>0	For <code>cv_wait_sig()</code> , <code>cv_timedwait()</code> or <code>cv_timedwait_sig()</code> indicates that the condition was met and the function returned due to a call to <code>cv_signal()</code> or <code>cv_broadcast()</code> , 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.

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

*Writing Device Drivers*

## copyb(9F)

<b>NAME</b>	copyb – copy a message block
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *copyb(mblk_t *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the message block from which data is copied.
<b>DESCRIPTION</b>	<code>copyb()</code> allocates a new message block, and copies into it the data from the block that <i>bp</i> denotes. The new block will be at least as large as the block being copied. <code>copyb()</code> uses the <code>b_rptr</code> and <code>b_wptr</code> members of <i>bp</i> to determine how many bytes to copy.
<b>RETURN VALUES</b>	If successful, <code>copyb()</code> returns a pointer to the newly allocated message block containing the copied data. Otherwise, it returns a <code>NULL</code> pointer.
<b>CONTEXT</b>	<code>copyb()</code> can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1 : Using copyb</b></p> <p>For each message in the list, test to see if the downstream queue is full with the <code>canputnext(9F)</code> function (line 21). If it is not full, use <code>copyb</code> to copy a header message block, and <code>dupmsg(9F)</code> to duplicate the data to be retransmitted. If either operation fails, reschedule a timeout at the next valid interval.</p> <p>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.</p> <pre>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     ... 10    int         h_address; 11    ... 12 }; 13 mblk_t *header; 14 void 15 retransmit(struct retrans *ret) 16 { 17     mblk_t *bp, *mp; 18     struct protoheader *php; 19 20     while (ret) { 21         if (!canputnext(ret-&gt;r_outq)) { /* no room */</pre>

**EXAMPLE 1** : Using copyb (Continued)

```

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

```

**SEE ALSO** allocb(9F), canputnext(9F), dupmsg(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

## copyin(9F)

<b>NAME</b>	copyin – copy data from a user program to a driver buffer						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  int <b>copyin</b>(const void *userbuf, void *driverbuf, size_t cn);</pre>						
<b>INTERFACE LEVEL</b>	This interface is obsolete. ddi_copyin(9F) should be used instead.						
<b>PARAMETERS</b>	<table><tr><td><i>userbuf</i></td><td>User program source address from which data is transferred.</td></tr><tr><td><i>driverbuf</i></td><td>Driver destination address to which data is transferred.</td></tr><tr><td><i>cn</i></td><td>Number of bytes transferred.</td></tr></table>	<i>userbuf</i>	User program source address from which data is transferred.	<i>driverbuf</i>	Driver destination address to which data is transferred.	<i>cn</i>	Number of bytes transferred.
<i>userbuf</i>	User program source address from which data is transferred.						
<i>driverbuf</i>	Driver destination address to which data is transferred.						
<i>cn</i>	Number of bytes transferred.						
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p>						
<b>RETURN VALUES</b>	<p>Under normal conditions, a 0 is returned indicating a successful copy. Otherwise, a -1 is returned if one of the following occurs:</p> <ul style="list-style-type: none"><li>■ Paging fault; the driver tried to access a page of memory for which it did not have read or write access.</li><li>■ Invalid user address, such as a user area or stack area.</li><li>■ Invalid address that would have resulted in data being copied into the user block.</li><li>■ Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.</li></ul> <p>If a -1 is returned to the caller, driver entry point routines should return EFAULT.</p>						
<b>CONTEXT</b>	copyin() can be called from user context only.						
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> An ioctl() Routine</p> <p>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.</p> <pre>1 struct device {          /* layout of physical device registers */ 2     int     control;     /* physical device control word */ 3     int     status;      /* physical device status word */ 4     short   rcv_char;    /* receive character from device */ 5     short   xmit_char;   /* transmit character to device */ 6 }; 7 8 extern struct device xx_addr[]; /* phys. device regs. location */</pre>						

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

```

 9      . . .
10  xx_ioctl(dev_t dev, int cmd, int arg, int mode,
11          cred_t *cred_p, int *rval_p)
12      . . .
13  {
14      register struct device *rp = &xx_addr[getminor(dev) >> 4];
15      switch (cmd) {
16
17          case XX_GETREGS: /* copy device regs. to user program */
18              if (copyin(arg, rp, sizeof(struct device)))
19                  return(EFAULT);
20              break;
21          . . .
22      }
23      . . .
24  }

```

**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*.

## copymsg(9F)

<b>NAME</b>	copymsg – copy a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  mblk_t *copymsg(mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>mp</i> Pointer to the message to be copied.
<b>DESCRIPTION</b>	copymsg() forms a new message by allocating new message blocks, and copying the contents of the message referred to by <i>mp</i> (using the copyb(9F) function). It returns a pointer to the new message.
<b>RETURN VALUES</b>	If the copy is successful, copymsg() returns a pointer to the new message. Otherwise, it returns a NULL pointer.
<b>CONTEXT</b>	copymsg() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1 : Using copymsg</b></p> <p>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).</p> <pre>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-&gt;b_datap-&gt;db_ref &gt; 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-&gt;b_cont) { 16       for (cp = tmp-&gt;b_rptr; cp &lt; tmp-&gt;b_wptr; cp++) { 17           if ((*cp &lt;= 'z') &amp;&amp; (*cp &gt;= 'a')) 18               *cp -= 0x20; 19       } 20   } 21   return(cmp); 22 }</pre>

**EXAMPLE 1** : Using copymsg (Continued)

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

*Writing Device Drivers*

*STREAMS Programming Guide*

## copyout(9F)

<b>NAME</b>	copyout – copy data from a driver to a user program						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  int <b>copyout</b>(const void *<i>driverbuf</i>, void *<i>userbuf</i>, size_t <i>cn</i>);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	This interface is obsolete. ddi_copyout(9F) should be used instead. <table><tr><td><i>driverbuf</i></td><td>Source address in the driver from which the data is transferred.</td></tr><tr><td><i>userbuf</i></td><td>Destination address in the user program to which the data is transferred.</td></tr><tr><td><i>cn</i></td><td>Number of bytes moved.</td></tr></table>	<i>driverbuf</i>	Source address in the driver from which the data is transferred.	<i>userbuf</i>	Destination address in the user program to which the data is transferred.	<i>cn</i>	Number of bytes moved.
<i>driverbuf</i>	Source address in the driver from which the data is transferred.						
<i>userbuf</i>	Destination address in the user program to which the data is transferred.						
<i>cn</i>	Number of bytes moved.						
<b>DESCRIPTION</b>	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.						
<b>RETURN VALUES</b>	Under normal conditions, a 0 is returned to indicate a successful copy. Otherwise, a –1 is returned if one of the following occurs: <ul style="list-style-type: none"><li>■ Paging fault; the driver tried to access a page of memory for which it did not have read or write access.</li><li>■ Invalid user address, such as a user area or stack area.</li><li>■ Invalid address that would have resulted in data being copied into the user block.</li><li>■ Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.</li></ul> If a –1 is returned to the caller, driver entry point routines should return EFAULT.						
<b>CONTEXT</b>	copyout () can be called from user context only.						
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> An ioctl () Routine</p> <p>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.</p> <pre>1 struct device {          /* layout of physical device registers */ 2     int     control;      /* physical device control word */ 3     int     status;       /* physical device status word */ 4     short   recv_char;    /* receive character from device */ 5     short   xmit_char;    /* transmit character to device */ 6 }; 7 8 extern struct device xx_addr[]; /* phys. device regs. location */ 9     . . . 10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,</pre>						

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

```

11     cred_t *cred_p, int *rval_p)
12     ...
13     {
14     register struct device *rp = &xx_addr[getminor(dev) >> 4];
15     switch (cmd) {
16
17     case XX_GETREGS:    /* copy device regs. to user program */
18         if (copyout(rp, arg, sizeof(struct device)))
19             return(EFAULT);
20         break;
21         ...
22     }
23     ...
24 }

```

**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*.

## csx\_AccessConfigurationRegister(9F)

<b>NAME</b>	csx_AccessConfigurationRegister – read or write a PC Card Configuration Register								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_AccessConfigurationRegister(client_handle_t ch,     access_config_reg_t *acr);</pre>								
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)								
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>acr</i> Pointer to an access_config_reg_t structure.</p>								
<b>DESCRIPTION</b>	This function allows a client to read or write a PC Card Configuration Register.								
<b>STRUCTURE MEMBERS</b>	<p>The structure members of access_config_reg_t are:</p> <pre>uint32_t Socket; /* socket number*/ uint32_t Action; /* register access operation*/ uint32_t Offset; /* config register offset*/ uint32_t Value; /* value read or written*/</pre> <p>The fields are defined as follows:</p> <table><tr><td>Socket</td><td>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</td></tr><tr><td>Action</td><td>May be set to CONFIG_REG_READ or CONFIG_REG_WRITE. All other values in the Action field are reserved for future use. If the Action field is set to CONFIG_REG_WRITE, the Value field is written to the specified configuration register. Card Services does not read the configuration register after a write operation. For that reason, the Value field is only updated by a CONFIG_REG_READ request.</td></tr><tr><td>Offset</td><td>Specifies the byte offset for the desired configuration register from the PC Card configuration register base specified in csx_RequestConfiguration(9F).</td></tr><tr><td>Value</td><td>Contains the value read from the PC Card Configuration Register for a read operation. For a write operation, the Value field contains the value to write to the configuration register. As noted above, on return from a write request, the Value field is the value written to the PC Card and not any changed value that may have resulted from the write request (that is, no read after write is performed).</td></tr></table> <p>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.</p>	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).
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).								

## csx\_AccessConfigurationRegister(9F)

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

<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_ARGS</code>	Specified arguments are invalid. Client specifies an <code>Offset</code> that is out of range or neither <code>CONFIG_REG_READ</code> or <code>CONFIG_REG_WRITE</code> is set.
<code>CS_UNSUPPORTED_MODE</code>	Client has not called <code>csx_RequestConfiguration(9F)</code> before calling this function.
<code>CS_BAD_HANDLE</code>	Client handle is invalid.
<code>CS_NO_CARD</code>	No PC card in socket.
<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

### CONTEXT

This function may be called from user or kernel context.

### SEE ALSO

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

*PCCard 95 Standard, PCMCIA/JEIDA*

## csx\_ConvertSize(9F)

<b>NAME</b>	csx_ConvertSize – convert device sizes						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_ConvertSize</b>(convert_size_t *cs);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)						
<b>PARAMETERS</b>	<b>cs</b> Pointer to a convert_size_t structure.						
<b>DESCRIPTION</b>	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 <i>devsize</i> format to <i>bytes</i> and vice versa.						
<b>STRUCTURE MEMBERS</b>	<p>The structure members of convert_size_t are:</p> <pre>uint32_t  Attributes; uint32_t  bytes; uint32_t  devsize;</pre> <p>The fields are defined as follows:</p> <p><b>Attributes</b> This is a bit-mapped field that identifies the type of size conversion to be performed. The field is defined as follows:</p> <pre>CONVERT_BYTES_TO_DEVSIZE     Converts bytes to devsize format.  CONVERT_DEVSIZE_TO_BYTES     Converts devsize format to bytes.</pre> <p><b>bytes</b> If CONVERT_BYTES_TO_DEVSIZE is set, the value in the bytes field is converted to a <i>devsize</i> format and returned in the devsize field.</p> <p><b>devsize</b> If CONVERT_DEVSIZE_TO_BYTES is set, the value in the devsize field is converted to a <i>bytes</i> value and returned in the bytes field.</p>						
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_SIZE</td><td>Invalid <i>bytes</i> or <i>devsize</i>.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	CS_SUCCESS	Successful operation.	CS_BAD_SIZE	Invalid <i>bytes</i> or <i>devsize</i> .	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.
CS_SUCCESS	Successful operation.						
CS_BAD_SIZE	Invalid <i>bytes</i> or <i>devsize</i> .						
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.						
<b>CONTEXT</b>	This function may be called from user or kernel context.						
<b>SEE ALSO</b>	csx_ModifyWindow(9F), csx_RequestWindow(9F) <i>PCCard 95 Standard, PCMCIA/JEIDA</i>						

<b>NAME</b>	csx_ConvertSpeed – convert device speeds
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_ConvertSpeed</b>(convert_speed_t *cs);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>cs</i>            Pointer to a <code>convert_speed_t</code> structure.</p>
<b>DESCRIPTION</b>	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 <i>devspeed</i> format to <i>nS</i> and vice versa.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of <code>convert_speed_t</code> are:</p> <pre>uint32_t        Attributes; uint32_t        nS; uint32_t        devspeed;</pre> <p>The fields are defined as follows:</p> <p><i>Attributes</i>        This is a bit-mapped field that identifies the type of speed conversion to be performed. The field is defined as follows:</p> <pre>                  CONVERT_NS_TO_DEVSPEED                   Converts <i>nS</i> to <i>devspeed</i> format                    CONVERT_DEVSPEED_TO_NS                   Converts <i>devspeed</i> format to <i>nS</i></pre> <p><i>nS</i>                If <code>CONVERT_NS_TO_DEVSPEED</code> is set, the value in the <i>nS</i> field is converted to a <i>devspeed</i> format and returned in the <i>devspeed</i> field.</p> <p><i>devspeed</i>        If <code>CONVERT_DEVSPEED_TO_NS</code> is set, the value in the <i>devspeed</i> field is converted to an <i>nS</i> value and returned in the <i>nS</i> field.</p>
<b>RETURN VALUES</b>	<pre>CS_SUCCESS                            Successful operation. CS_BAD_SPEED                         Invalid <i>nS</i> or <i>devspeed</i>. CS_BAD_ATTRIBUTE                     Bad <i>Attributes</i> value. CS_UNSUPPORTED_FUNCTION              No PCMCIA hardware installed.</pre>
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	<pre>csx_ModifyWindow(9F), csx_RequestWindow(9F)  PC Card 95 Standard, PCMCIA/JEIDA</pre>

## csx\_CS\_DDI\_Info(9F)

<b>NAME</b>	csx_CS_DDI_Info – obtain DDI information						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_CS_DDI_Info(cs_ddi_info_t *cdi);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI) <i>cdi</i> Pointer to a <i>cs_ddi_info_t</i> structure.						
<b>DESCRIPTION</b>	This function is a Solaris-specific extension that is used by clients that need to provide the <i>xx_getinfo</i> driver entry point (see <i>getinfo(9E)</i> ). It provides a method for clients to obtain DDI information based on their socket number and client driver name.						
<b>STRUCTURE MEMBERS</b>	<p>The structure members of <i>cs_ddi_info_t</i> are:</p> <pre>uint32_t Socket; /* socket number */ char* driver_name; /* unique driver name */ dev_info_t *dip; /* dip */ int32_t instance; /* instance */</pre> <p>The fields are defined as follows:</p> <p><i>Socket</i> This field must be set to the physical socket number that the client is interested in getting information about.</p> <p><i>driver_name</i> This field must be set to a string containing the name of the client driver to get information about.</p> <p>If <i>csx_CS_DDI_Info()</i> is used in a client's <i>xx_getinfo</i> function, then the client will typically extract the <i>Socket</i> value from the <i>*arg</i> argument and it <i>must</i> set the <i>driver_name</i> field to the same string used with <i>csx_RegisterClient(9F)</i>.</p> <p>If the <i>driver_name</i> is found on the <i>Socket</i>, the <i>csx_CS_DDI_Info()</i> function returns both the <i>dev_info</i> pointer and the <i>instance</i> fields for the requested driver instance.</p>						
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_SOCKET</td><td>Client not found on <i>Socket</i>.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	CS_SUCCESS	Successful operation.	CS_BAD_SOCKET	Client not found on <i>Socket</i> .	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.
CS_SUCCESS	Successful operation.						
CS_BAD_SOCKET	Client not found on <i>Socket</i> .						
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.						
<b>CONTEXT</b>	This function may be called from user or kernel context.						
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> : Using <i>csx_CS_DDI_Info</i></p> <p>The following example shows how a client might call the <i>csx_CS_DDI_Info()</i> in the client's <i>xx_getinfo</i> function to return the <i>dip</i> or the instance number:</p> <pre>static int pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,               void **result) {     int error = DDI_SUCCESS;</pre>						

**EXAMPLE 1** : Using csx\_CS\_DDI\_Info (Continued)

```

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*

## csx\_DeregisterClient(9F)

<b>NAME</b>	csx_DeregisterClient – remove client from Card Services list								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_DeregisterClient(client_handle_t ch);</pre>								
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)								
<b>PARAMETERS</b>	<i>ch</i> Client handle returned from csx_RegisterClient(9F).								
<b>DESCRIPTION</b>	<p>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 <i>client_handle_t</i> argument.</p> <p>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.</p>								
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_IN_USE</td><td>Resources not released by this client.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.								
<b>CONTEXT</b>	This function may be called from user or kernel context.								
<b>SEE ALSO</b>	csx_RegisterClient(9F) <i>PC Card 95 Standard, PCMCIA/JEIDA</i>								
<b>WARNINGS</b>	Clients should be prepared to receive callbacks until Card Services returns from this request successfully.								

<b>NAME</b>	csx_DupHandle – duplicate access handle																				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_DupHandle(acc_handle_t handle1, acc_handle_t *handle2,     uint32_t flags);</pre>																				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>handle1</i>            The access handle returned from csx_RequestIO(9F) or csx_RequestWindow(9F) that is to be duplicated.</p> <p><i>handle2</i>            A pointer to the newly-created duplicated data access handle.</p> <p><i>flags</i>                The access attributes that will be applied to the new handle.</p>																				
<b>DESCRIPTION</b>	<p>This function duplicates the handle, <i>handle1</i>, into a new handle, <i>handle2</i>, that has the access attributes specified in the <i>flags</i> argument. Both the original handle and the new handle are active and can be used with the common access functions.</p> <p>Both handles must be explicitly freed when they are no longer necessary.</p> <p>The <i>flags</i> argument is bit-mapped. The following bits are defined:</p> <table border="0"> <tr> <td>WIN_ACC_NEVER_SWAP</td> <td>Host endian byte ordering</td> </tr> <tr> <td>WIN_ACC_BIG_ENDIAN</td> <td>Big endian byte ordering</td> </tr> <tr> <td>WIN_ACC_LITTLE_ENDIAN</td> <td>Little endian byte ordering</td> </tr> <tr> <td>WIN_ACC_STRICT_ORDER</td> <td>Program ordering references</td> </tr> <tr> <td>WIN_ACC_UNORDERED_OK</td> <td>May re-order references</td> </tr> <tr> <td>WIN_ACC_MERGING_OK</td> <td>Merge stores to consecutive locations</td> </tr> <tr> <td>WIN_ACC_LOADCACHING_OK</td> <td>May cache load operations</td> </tr> <tr> <td>WIN_ACC_STORECACHING_OK</td> <td>May cache store operations</td> </tr> </table> <p>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 <i>flags</i> bits. Only one of the following bits may be specified:</p> <table border="0"> <tr> <td>WIN_ACC_STRICT_ORDER</td> <td>The data references must be issued by a CPU in program order. Strict ordering is the default behavior.</td> </tr> <tr> <td>WIN_ACC_UNORDERED_OK</td> <td>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).</td> </tr> </table>	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_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_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_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).																				

## csx\_DupHandle(9F)

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*

<b>NAME</b>	csx_Error2Text – convert error return codes to text strings				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Error2Text(error2text_t *er);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>er</i>            Pointer to an error2text_t structure.</p>				
<b>DESCRIPTION</b>	This function is a Solaris-specific extension that provides a method for clients to convert Card Services error return codes to text strings.				
<b>STRUCTURE MEMBERS</b>	<p>The structure members of error2text_t are:</p> <pre>uint32_t     item;                                /*the error code*/ char         text[CS_ERROR_MAX_BUFSIZE];       /*the error code*/</pre> <p>A pointer to the text for the Card Services error return code in the <i>item</i> field is returned in the <i>text</i> 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 <i>item</i> field is not found, the <i>text</i> field will be set to a string of the form:</p> <p><i>"{unknown Card Services return code}"</i></p>				
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	CS_SUCCESS	Successful operation.	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.
CS_SUCCESS	Successful operation.				
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.				
<b>CONTEXT</b>	This function may be called from user or kernel context.				
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> : Using the csxError2Text function</p> <pre>if ((ret = csx_RegisterClient(&amp;client_handle, &amp;                               client_reg)) != CS_SUCCESS) {     error2text_t error2text;     error2text.item = ret;     csx_Error2Text(&amp;error2text);     cmn_err(CE_CONT, "RegisterClient failed %s (0x%x)",             error2text.text, ret); }</pre>				
<b>SEE ALSO</b>	<p>csx_Event2Text(9F)</p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>				

## csx\_Event2Text(9F)

<b>NAME</b>	csx_Event2Text – convert events to text strings				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_Event2Text</b>(event2text_t *ev);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)				
<b>PARAMETERS</b>	<i>ev</i> Pointer to an event2text_t structure.				
<b>DESCRIPTION</b>	This function is a Solaris-specific extension that provides a method for clients to convert Card Services events to text strings.				
<b>STRUCTURE MEMBERS</b>	<p>The structure members of event2text_t are:</p> <pre>event_t    event;                /*the event code*/ char       text[CS_EVENT_MAX_BUFSIZE] /*the event code*/</pre> <p>The fields are defined as follows:</p> <p><i>event</i> The text for the event code in the <i>event</i> field is returned in the <i>text</i> field.</p> <p><i>text</i> The text string describing the name of the event.</p>				
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	CS_SUCCESS	Successful operation.	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.
CS_SUCCESS	Successful operation.				
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.				
<b>CONTEXT</b>	This function may be called from user or kernel context.				
<b>EXAMPLES</b>	<p><b>EXAMPLE 1:</b> Using csx_Event2Text ()</p> <pre>xx_event(event_t event, int priority, event_callback_args_t *eca) {     event2text_t    event2text;      event2text.event = event;     csx_Event2Text(&amp;event2text);     cmn_err(CE_CONT, "event %s (0x%x)", event2text.text, (int)event); }</pre>				
<b>SEE ALSO</b>	csx_event_handler(9E), csx_Error2Text(9F) <i>PC Card 95 Standard, PCMCIA/JEIDA</i>				

<b>NAME</b>	csx_FreeHandle – free access handle
<b>SYNOPSIS</b>	#include <sys/pccard.h>  int32_t <b>csx_FreeHandle</b> (acc_handle_t *handle);
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<i>handle</i> The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
<b>DESCRIPTION</b>	This function frees the handle, <i>handle</i> . 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.
<b>RETURN VALUES</b>	CS_SUCCESS                              Successful operation.  CS_UNSUPPORTED_FUNCTION              No PCMCIA hardware installed.
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	csx_DupHandle(9F), csx_RequestIO(9F), csx_RequestWindow(9F)  <i>PC Card95 Standard, PCMCIA/JEIDA</i>

## csx\_Get8(9F)

<b>NAME</b>	csx_Get8, csx_Get16, csx_Get32, csx_Get64 – read data from device address
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>handle</i>     The access handle returned from <code>csx_RequestIO(9F)</code>, <code>csx_RequestWindow(9F)</code>, or <code>csx_DupHandle(9F)</code>.</p> <p><i>offset</i>     The offset in bytes from the base of the mapped resource.</p>
<b>DESCRIPTION</b>	<p>These functions generate a read of various sizes from the mapped memory or device register.</p> <p>The <code>csx_Get8()</code>, <code>csx_Get16()</code>, <code>csx_Get32()</code>, and <code>csx_Get64()</code> functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, <i>handle</i>, at an offset in bytes represented by the <i>offset</i>, <i>offset</i>.</p> <p>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.</p>
<b>RETURN VALUES</b>	These functions return the value read from the mapped address.
<b>CONTEXT</b>	These functions may be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	<code>csx_DupHandle(9F)</code> , <code>csx_GetMappedAddr(9F)</code> , <code>csx_Put8(9F)</code> , <code>csx_RepGet8(9F)</code> , <code>csx_RepPut8(9F)</code> , <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code>
	<i>PC Card 95 Standard</i> , <i>PCMCIA/JEIDA</i>

<b>NAME</b>	csx_GetFirstClient, csx_GetNextClient – return first or next client												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_GetFirstClient(get_firstnext_client_t *fnc); int32_t csx_GetNextClient(get_firstnext_client_t *fnc);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>fnc</i> Pointer to a <code>get_firstnext_client_t</code> structure.</p>												
<b>DESCRIPTION</b>	The functions <code>csx_GetFirstClient()</code> and <code>csx_GetNextClient()</code> return information about the first or subsequent PC cards, respectively, that are installed in the system.												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of <code>get_firstnext_client_t</code> are:</p> <pre>uint32_t      Socket;          /* socket number */ uint32_t      Attributes;     /* attributes */ client_handle_t client_handle; /* client handle */ uint32_t      num_clients;    /* number of clients */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> If the <code>CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY</code> attribute is set, return information only on the PC card installed in this socket.</p> <p><b>Attributes</b> This field indicates the type of client. The field is bit-mapped; the following bits are defined:</p> <p><code>CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS</code> Return information on all clients.</p> <p><code>CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY</code> Return client information for the specified socket only.</p> <p><b>client_handle</b> The client handle of the PC card driver is returned in this field.</p> <p><b>num_clients</b> The number of clients is returned in this field.</p>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td><code>CS_SUCCESS</code></td> <td>Successful operation.</td> </tr> <tr> <td><code>CS_BAD_HANDLE</code></td> <td>Client handle is invalid.</td> </tr> <tr> <td><code>CS_BAD_SOCKET</code></td> <td>Socket number is invalid.</td> </tr> <tr> <td><code>CS_NO_CARD</code></td> <td>No PC Card in socket.</td> </tr> <tr> <td><code>CS_NO_MORE_ITEMS</code></td> <td>PC Card driver does not handle the <code>CS_EVENT_CLIENT_INFO</code> event.</td> </tr> <tr> <td><code>CS_UNSUPPORTED_FUNCTION</code></td> <td>No PCMCIA hardware installed.</td> </tr> </table>	<code>CS_SUCCESS</code>	Successful operation.	<code>CS_BAD_HANDLE</code>	Client handle is invalid.	<code>CS_BAD_SOCKET</code>	Socket number is invalid.	<code>CS_NO_CARD</code>	No PC Card in socket.	<code>CS_NO_MORE_ITEMS</code>	PC Card driver does not handle the <code>CS_EVENT_CLIENT_INFO</code> event.	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.
<code>CS_SUCCESS</code>	Successful operation.												
<code>CS_BAD_HANDLE</code>	Client handle is invalid.												
<code>CS_BAD_SOCKET</code>	Socket number is invalid.												
<code>CS_NO_CARD</code>	No PC Card in socket.												
<code>CS_NO_MORE_ITEMS</code>	PC Card driver does not handle the <code>CS_EVENT_CLIENT_INFO</code> event.												
<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<code>csx_event_handler(9E)</code>												

csx\_GetFirstClient(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*

<b>NAME</b>	csx_GetFirstTuple, csx_GetNextTuple – return Card Information Structure tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_GetFirstTuple(client_handle_t ch, tuple_t *tu); int32_t csx_GetNextTuple(client_handle_t ch, tuple_t *tu);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure.</p>
<b>DESCRIPTION</b>	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.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of tuple_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> This field is bit-mapped. The following bits are defined:</p> <p><b>TUPLE_RETURN_LINK</b> 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:</p> <pre>CISTPL_NULL CISTPL_LONGLINK_MFC CISTPL_LONGLINK_A CISTPL_LINKTARGET CISTPL_LONGLINK_C CISTPL_NO_LINK CISTPL_LONGLINK_CB CISTPL_END</pre> <p><b>TUPLE_RETURN_IGNORED_TUPLES</b> Return ignored tuples if set. Ignored tuples will be returned by this function if the TUPLE_RETURN_IGNORED_TUPLES bit in the Attributes field is set, see tuple(9S) for more information. The CIS is parsed from the location setup by the previous csx_GetFirstTuple() or csx_GetNextTuple() request.</p> <p><b>DesiredTuple</b> 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,</p>

## csx\_GetFirstTuple(9F)

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

<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_HANDLE</code>	Client handle is invalid.
<code>CS_NO_CARD</code>	No PC Card in socket.
<code>CS_NO_CIS</code>	No Card Information Structure (CIS) on PC card.
<code>CS_NO_MORE_ITEMS</code>	Desired tuple not found.
<code>CS_UNSUPPORTED_FUNCTION</code>	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*

<b>NAME</b>	csx_GetHandleOffset – return current access handle offset
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_GetHandleOffset(acc_handle_t handle, uint32_t *offset);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>handle</i>            Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).</p> <p><i>offset</i>            Pointer to a uint32_t in which the current access handle offset is returned.</p>
<b>DESCRIPTION</b>	This function returns the current offset for the access handle, <i>handle</i> , in <i>offset</i> .
<b>RETURN VALUES</b>	<p>CS_SUCCESS Successful operation.</p>
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	<p>csx_RequestIO(9F), csx_RequestIRQ(9F), csx_SetHandleOffset(9F)</p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>

## csx\_GetMappedAddr(9F)

<b>NAME</b>	csx_GetMappedAddr – return mapped virtual address	
<b>SYNOPSIS</b>	#include <sys/pccard.h>  int32_t <b>csx_GetMappedAddr</b> (acc_handle_t <i>handle</i> , void ** <i>addr</i> );	
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>handle</i> The access handle returned from <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> , or <code>csx_DupHandle(9F)</code> .  <i>addr</i> The virtual or I/O port number represented by the handle.	
<b>DESCRIPTION</b>	This function returns the mapped virtual address or the mapped I/O port number represented by the handle, <i>handle</i> .	
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	This function may be called from user, kernel, or interrupt context.	
<b>SEE ALSO</b>	<code>csx_DupHandle(9F)</code> , <code>csx_Get8(9F)</code> , <code>csx_Put8(9F)</code> , <code>csx_RepGet8(9F)</code> , <code>csx_RepPut8(9F)</code> , <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code>  <i>PC Card 95 Standard</i> , <i>PCMCIA/JEIDA</i>	

<b>NAME</b>	csx_GetStatus – return the current status of a PC Card and its socket
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_GetStatus(client_handle_t ch, get_status_t *gs);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>gs</i> Pointer to a get_status_t structure.</p>
<b>DESCRIPTION</b>	This function returns the current status of a PC Card and its socket.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of get_status_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p>Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p>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:</p> <p>CS_STATUS_WRITE_PROTECTED Card is write protected</p> <p>CS_STATUS_CARD_LOCKED Card is locked</p> <p>CS_STATUS_EJECTION_REQUEST Ejection request in progress</p> <p>CS_STATUS_INSERTION_REQUEST Insertion request in progress</p> <p>CS_STATUS_BATTERY_DEAD Card battery is dead</p> <p>CS_STATUS_BATTERY_DEAD Card battery is dead (BVD1)</p> <p>CS_STATUS_BATTERY_LOW Card battery is low (BVD2)</p> <p>CS_STATUS_CARD_READY Card is READY</p> <p>CS_STATUS_CARD_INSERTED Card is inserted</p>

## csx\_GetStatus(9F)

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)`

`csx_GetStatus(9F)`

*PC Card 95 Standard, PCMCIA/JEIDA*

<b>NAME</b>	csx_GetTupleData – return the data portion of a tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_GetTupleData(client_handle_t ch, tuple_t *tu);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure.</p>
<b>DESCRIPTION</b>	This function returns the data portion of a tuple, as returned by the csx_GetFirstTuple(9F) and csx_GetNextTuple(9F) functions.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of tuple_t are:</p> <p>The fields are defined as follows:</p> <pre>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 */</pre> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> Initialized by csx_GetFirstTuple(9F) or csx_GetNextTuple(9F); the client must not modify the value in this field.</p> <p><b>DesiredTuple</b> Initialized by csx_GetFirstTuple(9F) or csx_GetNextTuple(9F); the client must not modify the value in this field.</p> <p><b>TupleOffset</b> This field allows partial tuple information to be retrieved, starting anywhere within the tuple.</p> <p><b>TupleDataMax</b> 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.</p> <p><b>TupleDataLen</b> This field is the actual size of the tuple data body. It represents the number of tuple data body bytes returned.</p>

## csx\_GetTupleData(9F)

	TupleData	This field is an array of bytes containing the raw tuple data body contents.
	TupleCode	Initialized by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> ; the client must not modify the value in this field.
	TupleLink	Initialized by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> ; the client must not modify the value in this field.
<b>RETURN VALUES</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_BAD_ARGS	Data from prior <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> 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.
<b>CONTEXT</b>	This function may be called from user or kernel context.	
<b>SEE ALSO</b>	<code>csx_GetFirstTuple(9F)</code> , <code>csx_ParseTuple(9F)</code> , <code>csx_RegisterClient(9F)</code> , <code>csx_ValidateCIS(9F)</code> , <code>tuple(9S)</code>	
	<i>PC Card 95 Standard, PCMCIA/JEIDA</i>	

<b>NAME</b>	csx_MakeDeviceNode, csx_RemoveDeviceNode – create and remove minor nodes on behalf of the client
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>dn</i> Pointer to a make_device_node_t or remove_device_node_t structure.</p>
<b>DESCRIPTION</b>	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.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of make_device_node_t are:</p> <pre>uint32_t      Action;          /* device operation */ uint32_t      NumDevNodes;     /* number of nodes to create */ devnode_desc_t *devnode_desc; /* description of device nodes */</pre> <p>The structure members of remove_device_node_t are:</p> <pre>uint32_t      Action;          /* device operation */ uint32_t      NumDevNodes;     /* number of nodes to remove */ devnode_desc_t *devnode_desc; /* description of device nodes */</pre> <p>The structure members of devnode_desc_t are:</p> <pre>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 */</pre> <p>The Action field is used to specify the operation that csx_MakeDeviceNode() and csx_RemoveDeviceNode() should perform.</p> <p>The following Action values are defined for csx_MakeDeviceNode():</p> <p>CREATE_DEVICE_NODE Create NumDevNodes minor nodes</p> <p>The following Action values are defined for csx_RemoveDeviceNode():</p> <p>REMOVE_DEVICE_NODE Remove NumDevNodes minor nodes</p> <p>REMOVE_ALL_DEVICE_NODES Remove all minor nodes for this client</p>

## csx\_MakeDeviceNode(9F)

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

<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_HANDLE</code>	Client handle is invalid.
<code>CS_BAD_ATTRIBUTE</code>	The value of one or more arguments is invalid.
<code>CS_BAD_ARGS</code>	Action is invalid.
<code>CS_OUT_OF_RESOURCE</code>	Unable to create or remove device node.
<code>CS_UNSUPPORTED_FUNCTION</code>	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*

<b>NAME</b>	csx_MapLogSocket – return the physical socket number associated with the client handle
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_MapLogSocket(client_handle_t ch, map_log_socket_t     *ls);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>ls</i> Pointer to a map_log_socket_t structure.</p>
<b>DESCRIPTION</b>	This function returns the physical socket number associated with the client handle.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of map_log_socket_t are:</p> <pre>uint32_t    LogSocket;        /* logical socket number */ uint32_t    PhyAdapter;      /* physical adapter number */ uint32_t    PhySocket;      /* physical socket number */</pre> <p>The fields are defined as follows:</p> <p><b>LogSocket</b> Not used by this implementation of Card Services and can be set to any arbitrary value.</p> <p><b>PhyAdapter</b> Returns the physical adapter number, which is always 0 in the Solaris implementation of Card Services.</p> <p><b>PhySocket</b> 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.</p>
<b>RETURN VALUES</b>	<p><b>CS_SUCCESS</b> Successful operation.</p> <p><b>CS_BAD_HANDLE</b> Client handle is invalid.</p> <p><b>CS_UNSUPPORTED_FUNCTION</b> No PCMCIA hardware installed.</p>
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	<p>csx_RegisterClient(9F)</p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>

## csx\_MapMemPage(9F)

<b>NAME</b>	csx_MapMemPage – map the memory area on a PC Card												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_MapMemPage</b>(window_handle_t <i>wh</i>, map_mem_page_t <i>*mp</i>);</pre>												
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)												
<b>PARAMETERS</b>	<i>wh</i> Window handle returned from csx_RequestWindow(9F). <i>mp</i> Pointer to a map_mem_page_t structure.												
<b>DESCRIPTION</b>	This function maps the memory area on a PC Card into a page of a window allocated with the csx_RequestWindow(9F) function.												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of map_mem_page_t are:</p> <pre>uint32_t    CardOffset;    /* card offset */ uint32_t    Page;         /* page number */</pre> <p>The fields are defined as follows:</p> <p><b>CardOffset</b> The absolute offset in bytes from the beginning of the PC Card to map into system memory.</p> <p><b>Page</b> Used internally by Card Services; clients must set this field to 0 before calling this function.</p>												
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_BAD_OFFSET</td><td>Offset is invalid.</td></tr><tr><td>CS_BAD_PAGE</td><td>Page is not zero.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	csx_ModifyWindow(9F), csx_ReleaseWindow(9F), csx_RequestWindow(9F) <i>PC Card 95 Standard, PCMCIA/JEIDA</i>												

<b>NAME</b>	csx_ModifyConfiguration – modify socket and PC Card Configuration Register
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_ModifyConfiguration(client_handle_t ch,                                modify_config_t *mc);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>mc</i> Pointer to a modify_config_t structure.</p>
<b>DESCRIPTION</b>	This function allows a socket and PC Card configuration to be modified. This function can only modify a configuration requested via csx_RequestConfiguration(9F).
<b>STRUCTURE MEMBERS</b>	<p>The structure members of modify_config_t are:</p> <pre>uint32_t Socket;          /* socket number */ uint32_t Attributes;     /* attributes to modify */ uint32_t Vpp1;          /* Vpp1 value */ uint32_t Vpp2;          /* Vpp2 value */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> This field is bit-mapped. The following bits are defined:</p> <p><b>CONF_ENABLE_IRQ_STEERING</b> Enable IRQ steering. Set to connect the PC Card IREQ line to a previously selected system interrupt.</p> <p><b>CONF_IRQ_CHANGE_VALID</b> IRQ change valid. Set to request the IRQ steering enable to be changed.</p> <p><b>CONF_VPP1_CHANGE_VALID</b> Vpp1 change valid. These bits are set to request a change to the corresponding voltage level for the PC Card.</p> <p><b>CONF_VPP2_CHANGE_VALID</b> Vpp2 change valid. These bits are set to request a change to the corresponding voltage level for the PC Card.</p> <p><b>CONF_VSOVERRIDE</b> 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.)</p>

## csx\_ModifyConfiguration(9F)

	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 <i>PC Card 95 Standard, PCMCIA/JEIDA</i> , systems must always support 5.0 volts for both Vcc and Vpp. (See WARNINGS.)
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>		This function may be called from user or kernel context.
<b>SEE ALSO</b>		csx_RegisterClient(9F), csx_ReleaseConfiguration(9F), csx_ReleaseIO(9F), csx_ReleaseIRQ(9F), csx_RequestConfiguration(9F), csx_RequestIO(9F), csx_RequestIRQ(9F)
		<i>PC Card 95 Standard, PCMCIA/JEIDA</i>
<b>WARNINGS</b>		<ol style="list-style-type: none"><li>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.</li><li>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.</li></ol>
<b>NOTES</b>		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.

<b>NAME</b>	csx_ModifyWindow – modify window attributes				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_ModifyWindow</b>(window_handle_t wh, modify_win_t *mw);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)				
<b>PARAMETERS</b>	<p><i>wh</i> Window handle returned from csx_RequestWindow(9F).</p> <p><i>mw</i> Pointer to a modify_win_t structure.</p>				
<b>DESCRIPTION</b>	<p>This function modifies the attributes of a window allocated by the csx_RequestWindow(9F) function.</p> <p>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.</p>				
<b>STRUCTURE MEMBERS</b>	<p>The structure members of modify_win_t are:</p> <pre>uint32_t    Attributes;           /* window flags */ uint32_t    AccessSpeed;         /* window access speed */</pre> <p>The fields are defined as follows:</p> <p><b>Attributes</b> This field is bit-mapped and defined as follows:</p> <p><b>WIN_MEMORY_TYPE_CM</b> Window points to Common Memory area. Set this to map the window to Common Memory.</p> <p><b>WIN_MEMORY_TYPE_AM</b> Window points to Attribute Memory area. Set this to map the window to Attribute Memory.</p> <p><b>WIN_ENABLE</b> Enable Window. The client must set this to enable the window.</p> <p><b>WIN_ACCESS_SPEED_VALID</b> AccessSpeed valid. The client must set this when the AccessSpeed field has a value that the client wants set for the window.</p> <p><b>AccessSpeed</b> 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 <i>PC Card 95 Standard</i>), 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.</p> <table border="0"> <tr> <td style="padding-right: 20px;">0</td> <td>Reserved: do not use</td> </tr> <tr> <td>1</td> <td>250 nsec</td> </tr> </table>	0	Reserved: do not use	1	250 nsec
0	Reserved: do not use				
1	250 nsec				

## csx\_ModifyWindow(9F)

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

<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_HANDLE</code>	Window handle is invalid.
<code>CS_NO_CARD</code>	No PC Card in socket.
<code>CS_BAD_OFFSET</code>	Error getting/setting window hardware parameters.
<code>CS_BAD_WINDOW</code>	Error getting/setting window hardware parameters.
<code>CS_BAD_SPEED</code>	<code>AccessSpeed</code> is invalid.
<code>CS_UNSUPPORTED_FUNCTION</code>	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*

<b>NAME</b>	csx_Parse_CISTPL_BATTERY – parse the Battery Replacement Date tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_BATTERY(client_handle_t ch, tuple_t *tu,                                 cistpl_battery_t *cb);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cb</i> Pointer to a cistpl_battery_t structure which contains the parsed CISTPL_BATTERY tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Battery Replacement Date tuple, CISTPL_BATTERY, into a form usable by PC Card drivers.</p> <p>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.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_battery_t are:</p> <pre>uint32_t rday; /* date battery last replaced */ uint32_t xday; /* date battery due for replacement */</pre> <p>The fields are defined as follows:</p> <p><i>rday</i> This field indicates the date on which the battery was last replaced.</p> <p><i>xday</i> This field indicates the date on which the battery should be replaced.</p>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>												

## csx\_Parse\_CISTPL\_BYTEORDER(9F)

<b>NAME</b>	csx_Parse_CISTPL_BYTEORDER – parse the Byte Order tuple														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_BYTEORDER(client_handle_t ch, tuple_t     *tu, cistpl_byteorder_t *cbo);</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)														
	<i>ch</i> Client handle returned from csx_RegisterClient(9F).														
	<i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).														
	<i>cbo</i> Pointer to a cistpl_byteorder_t structure which contains the parsed CISTPL_BYTEORDER tuple information upon return from this function.														
<b>DESCRIPTION</b>	<p>This function parses the Byte Order tuple, CISTPL_BYTEORDER, into a form usable by PC Card drivers.</p> <p>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.</p>														
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_byteorder_t are:</p> <pre>uint32_t    order;    /* byte order code */ uint32_t    map;     /* byte mapping code */</pre> <p>The fields are defined as follows:</p> <table><tr><td>order</td><td>This field specifies the byte order for multi-byte numeric data.</td></tr><tr><td></td><td>TPLBYTEORD_LOW Little endian order</td></tr><tr><td></td><td>TPLBYTEORD_VS Vendor specific</td></tr><tr><td>map</td><td>This field specifies the byte mapping for 16-bit or wider cards.</td></tr><tr><td></td><td>TPLBYTEMAP_LOW Byte zero is least significant byte</td></tr><tr><td></td><td>TPLBYTEMAP_HIGH Byte zero is most significant byte</td></tr><tr><td></td><td>TPLBYTEMAP_VS Vendor specific mapping</td></tr></table>	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
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														
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr></table>	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_SUCCESS	Successful operation.														
CS_BAD_HANDLE	Client handle is invalid.														
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.														
CS_NO_CARD	No PC Card in socket.														

csx\_Parse\_CISTPL\_BYTEORDER(9F)

CS\_NO\_CIS

No Card Information Structure (CIS) PC Card.

CS\_UNSUPPORTED\_FUNCTION

No PCMCIA hardware installed.

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

**SEE ALSO** csx\_GetFirstTuple(9F), csx\_GetTupleData(9F), csx\_RegisterClient(9F), csx\_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

## csx\_Parse\_CISTPL\_CFTABLE\_ENTRY(9F)

<b>NAME</b>	csx_Parse_CISTPL_CFTABLE_ENTRY – parse 16-bit Card Configuration Table Entry tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_CFTABLE_ENTRY(client_handle_t ch,     tuple_t *tu, cistpl_cftable_entry_t *cft);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F).  <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  <i>cft</i> Pointer to a cistpl_cftable_entry_t structure which contains the parsed CISTPL_CFTABLE_ENTRY tuple information upon return from this function.
<b>DESCRIPTION</b>	<p>This function parses the 16 bit Card Configuration Table Entry tuple, CISTPL_CFTABLE_ENTRY, into a form usable by PC Card drivers.</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_cftable_entry_t are:</p> <pre>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 */</pre> <p>The flags field is defined and bit-mapped as follows:</p> <p>CISTPL_CFTABLE_TPCE_DEFAULT This is a default configuration</p> <p>CISTPL_CFTABLE_TPCE_IF If configuration byte exists</p> <p>CISTPL_CFTABLE_TPCE_FS_PWR Power information exists</p> <p>CISTPL_CFTABLE_TPCE_FS_TD Timing information exists</p>

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.

## csx\_Parse\_CISTPL\_CFTABLE\_ENTRY(9F)

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

## csx\_Parse\_CISTPL\_CFTABLE\_ENTRY(9F)

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

csx\_Parse\_CISTPL\_CFTABLE\_ENTRY(9F)

CISTPL\_CFTABLE\_TPCE\_MI\_PWRDOWN  
Powerdown capable

CISTPL\_CFTABLE\_TPCE\_MI\_DRQ\_MASK  
DMAREQ mask

CISTPL\_CFTABLE\_TPCE\_MI\_DRQ\_SPK  
DMAREQ on SPKR

CISTPL\_CFTABLE\_TPCE\_MI\_DRQ\_IOIS  
DMAREQ on IOIS16

CISTPL\_CFTABLE\_TPCE\_MI\_DRQ\_INP  
DMAREQ on INPACK

CISTPL\_CFTABLE\_TPCE\_MI\_DMA\_8  
DMA width 8 bits

CISTPL\_CFTABLE\_TPCE\_MI\_DMA\_16  
DMA width 16 bits

**RETURN VALUES**

CS\_SUCCESS  
Successful operation.

CS\_BAD\_HANDLE  
Client handle is invalid.

CS\_UNKNOWN\_TUPLE  
Parser does not know how to parse tuple.

CS\_NO\_CARD  
No PC Card in socket.

CS\_NO\_CIS  
No Card Information Structure (CIS) on PC Card.

CS\_UNSUPPORTED\_FUNCTION  
No PCMCIA hardware installed.

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

**SEE ALSO** csx\_GetFirstTuple(9F), csx\_GetTupleData(9F),  
csx\_Parse\_CISTPL\_CONFIG(9F), csx\_RegisterClient(9F),  
csx\_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

## csx\_Parse\_CISTPL\_CONFIG(9F)

<b>NAME</b>	csx_Parse_CISTPL_CONFIG – parse Configuration tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_CONFIG(client_handle_t ch, tuple_t *tu,                                 cistpl_config_t *cc);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cc</i> Pointer to a cistpl_config_t structure which contains the parsed CISTPL_CONFIG tuple information upon return from this function.</p>
<b>DESCRIPTION</b>	<p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_config_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p><i>present</i> This field indicates which configuration registers are present on the PC Card.</p> <p>CONFIG_OPTION_REG_PRESENT Configuration Option Register present</p> <p>CONFIG_STATUS_REG_PRESENT Configuration Status Register present</p> <p>CONFIG_PINREPL_REG_PRESENT Pin Replacement Register present</p> <p>CONFIG_COPY_REG_PRESENT Copy Register present</p> <p>CONFIG_EXSTAT_REG_PRESENT Extended Status Register present</p> <p>CONFIG_IOBASE0_REG_PRESENT IO Base 0 Register present</p>

	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 <code>regs</code> 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.
<b>RETURN VALUES</b>	<p>CS_SUCCESS Successful operation.</p> <p>CS_BAD_HANDLE Client handle is invalid.</p> <p>CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.</p> <p>CS_NO_CARD No PC Card in socket.</p> <p>CS_NO_CIS No Card Information Structure (CIS) on PC Card.</p> <p>CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.</p>
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	<p><code>csx_GetFirstTuple(9F)</code>, <code>csx_GetTupleData(9F)</code>,  <code>csx_Parse_CISTPL_CFTABLE_ENTRY(9F)</code>, <code>csx_RegisterClient(9F)</code>,  <code>csx_ValidateCIS(9F)</code>, <code>tuple(9S)</code></p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>

csx\_Parse\_CISTPL\_CONFIG(9F)

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

<b>NAME</b>	csx_Parse_CISTPL_DATE – parse the Card Initialization Date tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_DATE(client_handle_t ch, tuple_t *tu,                              cistpl_date_t *cd);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cd</i> Pointer to a cistpl_date_t structure which contains the parsed CISTPL_DATE tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Card Initialization Date tuple, CISTPL_DATE, into a form usable by PC Card drivers.</p> <p>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.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_date_t are:</p> <pre>uint32_t    time; uint32_t    day</pre> <p>The fields are defined as follows:</p> <p><i>time</i> This field indicates the time at which the PC Card was initialized.</p> <p><i>day</i> This field indicates the date the PC Card was initialized.</p>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard, PCMCIA/JEIDA</i></p>												

## csx\_Parse\_CISTPL\_DEVICE(9F)

<b>NAME</b>	csx_Parse_CISTPL_DEVICE, csx_Parse_CISTPL_DEVICE_A, csx_Parse_CISTPL_DEVICE_OC, csx_Parse_CISTPL_DEVICE_OA – parse Device Information tuples
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F).  <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  <i>cd</i> 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.
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_device_t are:</p> <pre>uint32_t          num_devices; /* number of devices found */ cistpl_device_node_t devnode[CISTPL_DEVICE_MAX_DEVICES];</pre> <p>The structure members of cistpl_device_node_t are:</p>

## csx\_Parse\_CISTPL\_DEVICE(9F)

```

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:

## csx\_Parse\_CISTPL\_DEVICE(9F)

CISTPL\_DEVICE\_DTYPE\_NULL  
No device

CISTPL\_DEVICE\_DTYPE\_ROM  
Masked ROM

CISTPL\_DEVICE\_DTYPE\_OTPROM  
One Time Programmable ROM

CISTPL\_DEVICE\_DTYPE\_EPROM  
UV EPROM

CISTPL\_DEVICE\_DTYPE\_EEPROM  
EEPROM

CISTPL\_DEVICE\_DTYPE\_FLASH  
FLASH

CISTPL\_DEVICE\_DTYPE\_SRAM  
Static RAM

CISTPL\_DEVICE\_DTYPE\_DRAM  
Dynamic RAM

CISTPL\_DEVICE\_DTYPE\_FUNCSPEC  
Function-specific memory address range

CISTPL\_DEVICE\_DTYPE\_EXTEND  
Extended type follows

<b>RETURN VALUES</b>	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*

<b>NAME</b>	csx_Parse_CISTPL_DEVICEGEO – parse the Device Geo tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_DEVICEGEO(client_handle_t ch, tuple_t     *tp, cistpl_devicegeo_t *pt);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tp</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo tuple information upon return from this function.</p>
<b>DESCRIPTION</b>	<p>This function parses the Device Geo tuple, CISTPL_DEVICEGEO, into a form usable by PC Card drivers.</p> <p>The CISTPL_DEVICEGEO tuple describes the device geometry of common memory partitions.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_devicegeo_t are:</p> <pre>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;</pre> <p>The fields are defined as follows:</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus</pre> <p>This field indicates the card interface width in bytes for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs</pre> <p>This field indicates the minimum erase block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs</pre> <p>This field indicates the minimum read block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs</pre> <p>This field indicates the minimum write block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part</pre> <p>This field indicates the segment partition subdivisions for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil</pre> <p>This field indicates the hardware interleave</p>
<b>RETURN VALUES</b>	<p>CS_SUCCESS Successful operation.</p>

csx\_Parse\_CISTPL\_DEVICEGEO(9F)

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*

<b>NAME</b>	csx_Parse_CISTPL_DEVICEGEO_A – parse the Device Geo A tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_DEVICEGEO_A(client_handle_t ch, tuple_t     *tp, cistpl_devicegeo_t *pt);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tp</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo A tuple information upon return from this function.</p>
<b>DESCRIPTION</b>	<p>This function parses the Device Geo A tuple, CISTPL_DEVICEGEO_A, into a form usable by PC Card drivers.</p> <p>The CISTPL_DEVICEGEO_A tuple describes the device geometry of attribute memory partitions.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_devicegeo_t are:</p> <pre>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;</pre> <p>The fields are defined as follows:</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus</pre> <p>This field indicates the card interface width in bytes for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs</pre> <p>This field indicates the minimum erase block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs</pre> <p>This field indicates the minimum read block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs</pre> <p>This field indicates the minimum write block size for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part</pre> <p>This field indicates the segment partition subdivisions for the given partition.</p> <pre>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil</pre> <p>This field indicates the hardware interleave for the given partition.</p>
<b>RETURN VALUES</b>	<p>CS_SUCCESS</p> <p>Successful operation.</p>

csx\_Parse\_CISTPL\_DEVICEGEO\_A(9F)

- 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*

<b>NAME</b>	csx_Parse_CISTPL_FORMAT – parse the Data Recording Format tuple														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_FORMAT(client_handle_t ch, tuple_t *tu,                                cistpl_format_t *pt);</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_format_t structure which contains the parsed CISTPL_FORMAT tuple information upon return from this function.</p>														
<b>DESCRIPTION</b>	<p>This function parses the Data Recording Format tuple, CISTPL_FORMAT, into a form usable by PC Card drivers.</p> <p>The CISTPL_FORMAT tuple indicates the data recording format for a device partition.</p>														
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_format_t are:</p> <pre>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;</pre> <p>The fields are defined as follows:</p> <table border="0"> <tr> <td style="vertical-align: top;">type</td> <td>This field indicates the type of device:</td> </tr> <tr> <td></td> <td>TPLFMTTYPER_DISK disk-like device</td> </tr> <tr> <td></td> <td>TPLFMTTYPER_MEM memory-like device</td> </tr> <tr> <td></td> <td>TPLFMTTYPER_VS vendor-specific device</td> </tr> <tr> <td style="vertical-align: top;">edc_length</td> <td>This field indicates the error detection code length.</td> </tr> <tr> <td style="vertical-align: top;">edc_type</td> <td>This field indicates the error detection code type.</td> </tr> <tr> <td style="vertical-align: top;">offset</td> <td>This field indicates the offset of the first byte of data in this partition.</td> </tr> </table>	type	This field indicates the type of device:		TPLFMTTYPER_DISK disk-like device		TPLFMTTYPER_MEM memory-like device		TPLFMTTYPER_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.
type	This field indicates the type of device:														
	TPLFMTTYPER_DISK disk-like device														
	TPLFMTTYPER_MEM memory-like device														
	TPLFMTTYPER_VS vendor-specific device														
edc_length	This field indicates the error detection code length.														
edc_type	This field indicates the error detection code type.														
offset	This field indicates the offset of the first byte of data in this partition.														

## csx\_Parse\_CISTPL\_FORMAT(9F)

nbytes	This field indicates the number of bytes of data in this partition
dev.disk.bksize	This field indicates the block size, for disk devices.
dev.disk.nblocks	This field indicates the number of blocks, for disk devices.
dev.disk.edcloc	This field indicates the location of the error detection code, for disk devices.
dev.mem.flags	This field provides flags, for memory devices. Valid flags are: TPLFMTFLAGS_ADDR address is valid TPLFMTFLAGS_AUTO automatically map memory region
dev.mem.reserved	This field is reserved.
dev.mem.address	This field indicates the physical address, for memory devices.
dev.mem.edcloc	This field indicates the location of the error detection code, for memory devices.

### RETURN VALUES

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

### CONTEXT

This function may be called from user or kernel context.

### SEE ALSO

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

*PC Card 95 Standard, PCMCIA/JEIDA*

<b>NAME</b>	csx_Parse_CISTPL_FUNCE – parse Function Extension tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_FUNCE(client_handle_t ch, tuple_t *tu,                                cistpl_funce_t *cf, uint32_t fid);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cf</i> Pointer to a cistpl_funce_t structure which contains the parsed CISTPL_FUNCE tuple information upon return from this function.</p> <p><i>fid</i> The function ID code to which this CISTPL_FUNCE tuple refers. See csx_Parse_CISTPL_FUNCID(9F).</p>
<b>DESCRIPTION</b>	<p>This function parses the Function Extension tuple, CISTPL_FUNCE, into a form usable by PC Card drivers.</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_funce_t are:</p> <pre>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; }</pre>

## csx\_Parse\_CISTPL\_FUNCE(9F)

```
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;     /* voice sampling rates (*100) */
    uint32_t nss;     /* voice sample sizes (*10) */
    uint32_t nsc;     /* voice compression methods */
    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

csx\_Parse\_CISTPL\_FUNCE(9F)

fc

TPLFE\_UC\_STOP\_15  
1.5 stop bits supported

TPLFE\_UC\_STOP\_2  
2 stop bits supported

This identifies the modem flow control methods and is defined as follows:

TPLFE\_FC\_TX\_XONOFF  
Transmit XON/XOFF

TPLFE\_FC\_RX\_XONOFF  
Receiver XON/XOFF

TPLFE\_FC\_TX\_HW  
Transmit hardware flow control (CTS)

TPLFE\_FC\_RX\_HW  
Receiver hardware flow control (RTS)

TPLFE\_FC\_TRANS  
Transparent flow control

ms  
This identifies the modem modulation standards and is defined as follows:

TPLFE\_MS\_BELL103  
300bps

TPLFE\_MS\_V21  
300bps (V.21)

TPLFE\_MS\_V23  
600/1200bps (V.23)

TPLFE\_MS\_V22AB  
1200bps (V.22A V.22B)

TPLFE\_MS\_BELL212  
2400bps (US Bell 212)

TPLFE\_MS\_V22BIS  
2400bps (V.22bis)

TPLFE\_MS\_V26  
2400bps leased line (V.26)

TPLFE\_MS\_V26BIS  
2400bps (V.26bis)

TPLFE\_MS\_V27BIS  
4800/2400bps leased line (V.27bis)

	TPLFE_MS_V29 9600/7200/4800 leased line (V.29)
	TPLFE_MS_V32 Up to 9600bps (V.32)
	TPLFE_MS_V32BIS Up to 14400bps (V.32bis)
	TPLFE_MS_VFAST Up to 28800 V.FAST
em	This identifies modem error correction/detection protocols and is defined as follows:  TPLFE_EM_MNP MNP levels 2-4  TPLFE_EM_V42 CCITT LAPM (V.42)
dc	This identifies modem data compression protocols and is defined as follows:  TPLFE_DC_V42BI CCITT compression V.42  TPLFE_DC_MNP5 MNP compression (uses MNP 2, 3 or 4)
cm	This identifies modem command protocols and is defined as follows:  TPLFE_CM_AT1 ANSI/EIA/TIA 602 "Action" commands  TPLFE_CM_AT2 ANSI/EIA/TIA 602 "ACE/DCE IF Params"  TPLFE_CM_AT3 ANSI/EIA/TIA 602 "Ace Parameters"  TPLFE_CM_MNP_AT MNP specification AT commands  TPLFE_CM_V25BIS V.25bis calling commands  TPLFE_CM_V25A V.25bis test procedures  TPLFE_CM_DMCL DMCL command mode
ex	This identifies the modem escape mechanism and is defined as follows:

csx\_Parse\_CISTPL\_FUNCE(9F)

	<p>TPLFE_EX_BREAK BREAK support standardized</p> <p>TPLFE_EX_PLUS +++ returns to command mode</p> <p>TPLFE_EX_UD User defined escape character</p>
dy	<p>This identifies modem standardized data encryption and is a reserved field for future use and must be set to 0.</p>
ef	<p>This identifies modem miscellaneous features and is defined as follows:</p> <p>TPLFE_EF_CALLERID Caller ID is supported</p>
fm	<p>This identifies fax modulation standards and is defined as follows:</p> <p>TPLFE_FM_V21C2 300bps (V.21-C2)</p> <p>TPLFE_FM_V27TER 4800/2400bps (V.27ter)</p> <p>TPLFE_FM_V29 9600/7200/4800 leased line (V.29)</p> <p>TPLFE_FM_V17 14.4K/12K/9600/7200bps (V.17)</p> <p>TPLFE_FM_V33 4.4K/12K/9600/7200 leased line (V.33)</p>
fs	<p>This identifies the fax feature selection and is defined as follows:</p> <p>TPLFE_FS_T3 Group 2 (T.3) service class</p> <p>TPLFE_FS_T4 Group 3 (T.4) service class</p> <p>TPLFE_FS_T6 Group 4 (T.6) service class</p> <p>TPLFE_FS_ECM Error Correction Mode</p> <p>TPLFE_FS_VOICEREQ Voice requests allowed</p>

	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

csx\_Parse\_CISTPL\_FUNCE(9F)

TPLFE\_LAN\_MEDIA\_SSR\_2\_4  
Spread spectrum radio 2.4 GHz  
TPLFE\_LAN\_MEDIA\_SSR\_5\_4  
Spread spectrum radio 5.4 GHz  
TPLFE\_LAN\_MEDIA\_DIFFUSE\_IR  
Diffuse infra red  
TPLFE\_LAN\_MEDIA\_PTP\_IR  
Point to point infra red

**RETURN VALUES**

CS\_SUCCESS  
Successful operation.  
CS\_BAD\_HANDLE  
Client handle is invalid.  
CS\_UNKNOWN\_TUPLE  
Parser does not know how to parse tuple.  
CS\_NO\_CARD  
No PC Card in socket.  
CS\_NO\_CIS  
No Card Information Structure (CIS) on PC Card.  
CS\_UNSUPPORTED\_FUNCTION  
No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

csx\_GetFirstTuple(9F), csx\_GetTupleData(9F),  
csx\_Parse\_CISTPL\_FUNCID(9F), csx\_RegisterClient(9F),  
csx\_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

<b>NAME</b>	csx_Parse_CISTPL_FUNCID – parse Function Identification tuple																
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_FUNCID(client_handle_t ch, tuple_t *tu,                                 cistpl_funcid_t *cf);</pre>																
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cf</i> Pointer to a cistpl_funcid_t structure which contains the parsed CISTPL_FUNCID tuple information upon return from this function.</p>																
<b>DESCRIPTION</b>	<p>This function parses the Function Identification tuple, CISTPL_FUNCID, into a form usable by PC Card drivers.</p> <p>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_FUNCID follow this tuple (see csx_Parse_CISTPL_FUNCID(9F)).</p>																
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_funcid_t are:</p> <pre>uint32_t    function;    /* PC Card function code */ uint32_t    sysinit;    /* system initialization mask */</pre> <p>The fields are defined as follows:</p> <table border="0"> <tr> <td style="vertical-align: top;">function</td> <td>This is the function type for CISTPL_FUNCID:</td> </tr> <tr> <td></td> <td>TPLFUNC_MULTI Vendor-specific multifunction card</td> </tr> <tr> <td></td> <td>TPLFUNC_MEMORY Memory card</td> </tr> <tr> <td></td> <td>TPLFUNC_SERIAL Serial I/O port</td> </tr> <tr> <td></td> <td>TPLFUNC_PARALLEL Parallel printer port</td> </tr> <tr> <td></td> <td>TPLFUNC_FIXED Fixed disk, silicon or removable</td> </tr> <tr> <td></td> <td>TPLFUNC_VIDEO Video interface</td> </tr> <tr> <td></td> <td>TPLFUNC_LAN Local Area Network adapter</td> </tr> </table>	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
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																

## csx\_Parse\_CISTPL\_FUNCID(9F)

TPLFUNC\_AIMS  
Auto Incrementing Mass Storage

TPLFUNC\_SCSI  
SCSI bridge

TPLFUNC\_SECURITY  
Security cards

TPLFUNC\_VENDOR\_SPECIFIC  
Vendor specific

TPLFUNC\_UNKNOWN  
Unknown function(s)

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

TPLINIT\_POST  
POST should attempt configure

TPLINIT\_ROM  
Map ROM during sys init

### RETURN VALUES

CS\_SUCCESS  
Successful operation.

CS\_BAD\_HANDLE  
Client handle is invalid.

CS\_UNKNOWN\_TUPLE  
Parser does not know how to parse tuple.

CS\_NO\_CARD  
No PC Card in socket.

CS\_NO\_CIS  
No Card Information Structure (CIS) on PC Card.

CS\_UNSUPPORTED\_FUNCTION  
No PCMCIA hardware installed.

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

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

*PC Card 95 Standard, PCMCIA/JEIDA*

<b>NAME</b>	csx_Parse_CISTPL_GEOMETRY – parse the Geometry tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_GEOMETRY(client_handle_t ch, tuple_t *tu,     cistpl_geometry_t *pt);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_geometry_t structure which contains the parsed CISTPL_GEOMETRY tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Geometry tuple, CISTPL_GEOMETRY, into a form usable by PC Card drivers.</p> <p>The CISTPL_GEOMETRY tuple indicates the geometry of a disk-like device.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_geometry_t are:</p> <pre>uint32_t    spt; uint32_t    tpc; uint32_t    ncyl;</pre> <p>The fields are defined as follows:</p> <p><i>spt</i> This field indicates the number of sectors per track.</p> <p><i>tpc</i> This field indicates the number of tracks per cylinder.</p> <p><i>ncyl</i> This field indicates the number of cylinders.</p>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard</i>, PCMCIA/JEIDA</p>												

## csx\_Parse\_CISTPL\_JEDEC\_C(9F)

<b>NAME</b>	csx_Parse_CISTPL_JEDEC_C, csx_Parse_CISTPL_JEDEC_A – parse JEDEC Identifier tuples												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  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);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F).  <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  <i>cj</i> 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.												
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_jedec_t are:</p> <pre>uint32_t      nid; /* # of JEDEC identifiers present */ jedec_ident_t jid[CISTPL_JEDEC_MAX_IDENTIFIERS];</pre> <p>The structure members of jedec_ident_t are:</p> <pre>uint32_t      id; /* manufacturer id */ uint32_t      info; /* manufacturer specific info */</pre>												
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_NO_CIS</td><td>No Card Information Structure (CIS) on PC Card.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.												
<b>CONTEXT</b>	These functions may be called from user or kernel context.												

`csx_Parse_CISTPL_JEDEC_C(9F)`

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

*PC Card 95 Standard*, PCMCIA/JEIDA

## csx\_Parse\_CISTPL\_LINKTARGET(9F)

<b>NAME</b>	csx_Parse_CISTPL_LINKTARGET – parse the Link Target tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_LINKTARGET(client_handle_t ch, tuple_t     *tu, cistpl_linktarget_t *pt);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F). <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F). <i>pt</i> Pointer to a cistpl_linktarget_t structure which contains the parsed CISTPL_LINKTARGET tuple information upon return from this function.												
<b>DESCRIPTION</b>	<p>This function parses the Link Target tuple, CISTPL_LINKTARGET, into a form usable by PCCard drivers.</p> <p>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.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_linktarget_t are:</p> <pre>uint32_t length; char      tpltg_tag[CIS_MAX_TUPLE_DATA_LEN];</pre> <p>The fields are defined as follows:</p> <p><i>length</i> This field indicates the number of bytes in tpltg_tag.</p> <p><i>tpltg_tag</i> This field provides the Link Target tuple information.</p>												
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_NO_CIS</td><td>No Card Information Structure (CIS) on PC Card.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)  <i>PC Card 95 Standard</i> , <i>PCMCIA/JEIDA</i>												

csx\_Parse\_CISTPL\_LOONGLINK\_A(9F)

<b>NAME</b>	csx_Parse_CISTPL_LOONGLINK_A, csx_Parse_CISTPL_LOONGLINK_C – parse the Long Link A and C tuples
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_LOONGLINK_A(client_handle_t ch, tuple_t     *tu, cistpl_longlink_ac_t *pt);  int32_t csx_Parse_CISTPL_LOONGLINK_C(client_handle_t ch, tuple_t     *tu, cistpl_longlink_ac_t *pt);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_longlink_ac_t structure which contains the parsed CISTPL_LOONGLINK_A or CISTPL_LOONGLINK_C tuple information upon return from this function.</p>
<b>DESCRIPTION</b>	<p>This function parses the Long Link A and C tuples, CISTPL_LOONGLINK_A and CISTPL_LOONGLINK_A, into a form usable by PC Card drivers.</p> <p>The CISTPL_LOONGLINK_A and CISTPL_LOONGLINK_C tuples provide links to Attribute and Common Memory.</p>
<b>STRUCTURE LEVEL MEMBERS</b>	<p>The structure members of cistpl_longlink_ac_t are:</p> <pre>uint32_t flags; uint32_t tpl1_addr;</pre> <p>The fields are defined as follows:</p> <p><i>flags</i> This field indicates the type of memory:</p> <p style="padding-left: 40px;">CISTPL_LOONGLINK_AC_AM long link to Attribute Memory</p> <p style="padding-left: 40px;">CISTPL_LOONGLINK_AC_CM long link to Common Memory</p> <p><i>tpl1_addr</i> This field provides the offset from the beginning of the specified address space.</p>
<b>RETURN VALUES</b>	<p>CS_SUCCESS Successful operation.</p> <p>CS_BAD_HANDLE Client handle is invalid.</p> <p>CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.</p> <p>CS_NO_CARD No PC Card in socket.</p> <p>CS_NO_CIS No Card Information Structure (CIS) on PC Card.</p>

csx\_Parse\_CISTPL\_LONGLINK\_A(9F)

CS\_UNSUPPORTED\_FUNCTION            No PCMCIA hardware installed.

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

**SEE ALSO**    csx\_GetFirstTuple(9F), csx\_GetTupleData(9F), csx\_RegisterClient(9F),  
csx\_ValidateCIS(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

csx\_Parse\_CISTPL\_LOONGLINK\_MFC(9F)

<b>NAME</b>	csx_Parse_CISTPL_LOONGLINK_MFC – parse the Multi-Function tuple
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_LOONGLINK_MFC(client_handle_t ch, tuple_t     *tu, cistpl_longlink_mfc_t *pt);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F).  <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  <i>pt</i> Pointer to a cistpl_longlink_mfc_t structure which contains the parsed CISTPL_LOONGLINK_MFC tuple information upon return from this function.
<b>DESCRIPTION</b>	This function parses the Multi-Function tuple, CISTPL_LOONGLINK_MFC, into a form usable by PC Card drivers.  The CISTPL_LOONGLINK_MFC tuple describes the start of the function-specific CIS for each function on a multi-function card.
<b>STRUCTURE MEMBERS</b>	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:  <i>nfuncs</i> This field indicates the number of functions on the PC card.  <i>nregs</i> This field indicates the number of configuration register sets.  <i>function[CIS_MAX_FUNCTIONS].tas</i> This field provides the target address space for each function on the PC card. This field can be one of:  CISTPL_LOONGLINK_MFC_TAS_AM CIS in attribute memory  CISTPL_LOONGLINK_MFC_TAS_CM CIS in common memory  <i>function[CIS_MAX_FUNCTIONS].addr</i> This field provides the target address offset for each function on the PC card.
<b>RETURN VALUES</b>	CS_SUCCESS Successful operation. CS_BAD_HANDLE Client handle is invalid.

csx\_Parse\_CISTPL\_LONGLINK\_MFC(9F)

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*

<b>NAME</b>	csx_Parse_CISTPL_MANFID – parse Manufacturer Identification tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_MANFID(client_handle_t ch, tuple_t *tu,                                 cistpl_manfid_t *cm);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cm</i> Pointer to a cistpl_manfid_t structure which contains the parsed CISTPL_MANFID tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Manufacturer Identification tuple, CISTPL_MANFID, into a form usable by PC Card drivers.</p> <p>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.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_manfid_t are:</p> <pre>uint32_t  manf; /* PCMCIA assigned manufacturer code */ uint32_t  card; /* manufacturer information                  (part number and/or revision) */</pre>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard</i>, PCMCIA/JEIDA</p>												

## csx\_Parse\_CISTPL\_ORG(9F)

<b>NAME</b>	csx_Parse_CISTPL_ORG – parse the Data Organization tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_ORG(client_handle_t ch, tuple_t *tu,                              cistpl_org_t *pt);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)												
	<i>ch</i> Client handle returned from csx_RegisterClient(9F).												
	<i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).												
	<i>pt</i> Pointer to a cistpl_org_t structure which contains the parsed CISTPL_ORG tuple information upon return from this function.												
<b>DESCRIPTION</b>	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.												
<b>STRUCTURE MEMBERS</b>	The structure members of cistpl_org_t are:												
	<pre>uint32_t type; char desc[CIS_MAX_TUPLE_DATA_LEN];</pre>												
	The fields are defined as follows:												
	<i>type</i> This field indicates type of data organization.												
	<i>desc[CIS_MAX_TUPLE_DATA_LEN]</i> This field provides the text description of this organization.												
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_NO_CIS</td><td>No Card Information Structure (CIS) on PC Card.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)  <i>PC Card 95 Standard, PCMCIA/JEIDA</i>												

<b>NAME</b>	csx_Parse_CISTPL_SPCL – parse the Special Purpose tuple										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_SPCL(client_handle_t ch, tuple_t *tu,                              cistpl_spcl_t *csp);</pre>										
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>csp</i> Pointer to a cistpl_spcl_t structure which contains the parsed CISTPL_SPCL tuple information upon return from this function.</p>										
<b>DESCRIPTION</b>	<p>This function parses the Special Purpose tuple, CISTPL_SPCL, into a form usable by PC Card drivers.</p> <p>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.</p>										
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_data_t are:</p> <pre>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];</pre> <p>The fields are defined as follows:</p> <p><i>id</i> 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.</p> <p><i>seq</i> This field contains a data sequence number. CISTPL_SPCL_SEQ_END is the last tuple in sequence.</p> <p><i>bytes</i> This field contains the number of data bytes in the data[CIS_MAX_TUPLE_DATA_LEN].</p> <p><i>data</i> The data component of this tuple.</p>										
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> </table>	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_SUCCESS	Successful operation.										
CS_BAD_HANDLE	Client handle is invalid.										
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.										
CS_NO_CARD	No PC Card in socket.										
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.										

csx\_Parse\_CISTPL\_SPCL(9F)

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*

<b>NAME</b>	csx_Parse_CISTPL_SWIL – parse the Software Interleaving tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_SWIL(client_handle_t ch, tuple_t *tu,                              cistpl_swil_t *pt);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>pt</i> Pointer to a cistpl_swil_t structure which contains the parsed CISTPL_SWIL tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Software Interleaving tuple, CISTPL_SWIL, into a form usable by PC Card drivers.</p> <p>The CISTPL_SWIL tuple provides the software interleaving of data within a partition on the card.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_swil_t are:</p> <pre>uint32_t    intrlv;</pre> <p>The fields are defined as follows:</p> <p><i>intrlv</i> This field provides the software interleaving for a partition.</p>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard</i>, PCMCIA/JEIDA</p>												

## csx\_Parse\_CISTPL\_VERS\_1(9F)

<b>NAME</b>	csx_Parse_CISTPL_VERS_1 – parse Level-1 Version/Product Information tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_VERS_1(client_handle_t ch, tuple_t *tu,                                 cistpl_vers_1_t *cv1);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F).  <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  <i>cv1</i> Pointer to a cistpl_vers_1_t structure which contains the parsed CISTPL_VERS_1 tuple information upon return from this function.												
<b>DESCRIPTION</b>	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.												
<b>STRUCTURE MEMBERS</b>	The structure members of cistpl_vers_1_t are: <pre>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 */</pre>												
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_NO_CIS</td><td>No Card Information Structure (CIS) on PC Card.</td></tr><tr><td>CS_UNSUPPORTED_FUNCTION</td><td>No PCMCIA hardware installed.</td></tr></table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)  <i>PC Card 95 Standard, PCMCIA/JEIDA</i>												

<b>NAME</b>	csx_Parse_CISTPL_VERS_2 – parse Level-2 Version and Information tuple												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_Parse_CISTPL_VERS_2(client_handle_t ch, tuple_t *tu,                                 cistpl_vers_2_t *cv2);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</p> <p><i>cv2</i> Pointer to a cistpl_vers_2_t structure which contains the parsed CISTPL_VERS_2 tuple information upon return from this function.</p>												
<b>DESCRIPTION</b>	<p>This function parses the Level-2 Version and Information tuple, CISTPL_VERS_2, into a form usable by PC Card drivers.</p> <p>The CISTPL_VERS_2 tuple is used to describe the card Level-2 information which has the logical organization of the card's data.</p>												
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cistpl_vers_2_t are:</p> <pre>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 */</pre>												
<b>RETURN VALUES</b>	<table border="0"> <tr> <td>CS_SUCCESS</td> <td>Successful operation.</td> </tr> <tr> <td>CS_BAD_HANDLE</td> <td>Client handle is invalid.</td> </tr> <tr> <td>CS_UNKNOWN_TUPLE</td> <td>Parser does not know how to parse tuple.</td> </tr> <tr> <td>CS_NO_CARD</td> <td>No PC Card in socket.</td> </tr> <tr> <td>CS_NO_CIS</td> <td>No Card Information Structure (CIS) on PC Card.</td> </tr> <tr> <td>CS_UNSUPPORTED_FUNCTION</td> <td>No PCMCIA hardware installed.</td> </tr> </table>	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.
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.												
<b>CONTEXT</b>	This function may be called from user or kernel context.												
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</p> <p><i>PC Card 95 Standard</i>, PCMCIA/JEIDA</p>												

## csx\_ParseTuple(9F)

<b>NAME</b>	csx_ParseTuple – generic tuple parser										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_ParseTuple</b>(client_handle_t <i>ch</i>, tuple_t *<i>tu</i>, cisparsed_t     *<i>cp</i>, cisdata_t <i>cd</i>);</pre>										
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI) <i>ch</i> Client handle returned from csx_RegisterClient(9F). <i>tu</i> Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F). <i>cp</i> Pointer to a cisparsed_t structure that unifies all tuple parsing structures. <i>cd</i> Extended tuple data for some tuples.										
<b>DESCRIPTION</b>	This function is the generic tuple parser entry point.										
<b>STRUCTURE MEMBERS</b>	The structure members of cisparsed_t are: <pre>typedef union cisparsed_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; } cisparsed_t;</pre>										
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_UNKNOWN_TUPLE</td><td>Parser does not know how to parse tuple.</td></tr><tr><td>CS_NO_CARD</td><td>No PC Card in socket.</td></tr><tr><td>CS_BAD_CIS</td><td>Generic parser error.</td></tr></table>	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_SUCCESS	Successful operation.										
CS_BAD_HANDLE	Client handle is invalid.										
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.										
CS_NO_CARD	No PC Card in socket.										
CS_BAD_CIS	Generic parser error.										

csx\_ParseTuple(9F)

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*

## csx\_Put8(9F)

<b>NAME</b>	csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  void <b>csx_Put8</b>(acc_handle_t handle, uint32_t offset, uint8_t value); void <b>csx_Put16</b>(acc_handle_t handle, uint32_t offset, uint16_t value); void <b>csx_Put32</b>(acc_handle_t handle, uint32_t offset, uint32_t value); void <b>csx_Put64</b>(acc_handle_t handle, uint32_t offset, uint64_t value);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI) <table><tr><td><i>handle</i></td><td>The access handle returned from <code>csx_RequestIO(9F)</code>, <code>csx_RequestWindow(9F)</code>, or <code>csx_DupHandle(9F)</code>.</td></tr><tr><td><i>offset</i></td><td>The offset in bytes from the base of the mapped resource.</td></tr><tr><td><i>value</i></td><td>The data to be written to the device.</td></tr></table>	<i>handle</i>	The access handle returned from <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> , or <code>csx_DupHandle(9F)</code> .	<i>offset</i>	The offset in bytes from the base of the mapped resource.	<i>value</i>	The data to be written to the device.
<i>handle</i>	The access handle returned from <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> , or <code>csx_DupHandle(9F)</code> .						
<i>offset</i>	The offset in bytes from the base of the mapped resource.						
<i>value</i>	The data to be written to the device.						
<b>DESCRIPTION</b>	<p>These functions generate a write of various sizes to the mapped memory or device register.</p> <p>The <code>csx_Put8()</code>, <code>csx_Put16()</code>, <code>csx_Put32()</code>, and <code>csx_Put64()</code> functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, <i>handle</i>, at an offset in bytes represented by the <i>offset</i>, <i>offset</i>.</p> <p>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.</p>						
<b>CONTEXT</b>	These functions may be called from user, kernel, or interrupt context.						
<b>SEE ALSO</b>	<code>csx_DupHandle(9F)</code> , <code>csx_Get8(9F)</code> , <code>csx_GetMappedAddr(9F)</code> , <code>csx_RepGet8(9F)</code> , <code>csx_RepPut8(9F)</code> , <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> <i>PC Card 95 Standard</i> , <i>PCMCIA/JEIDA</i>						

<b>NAME</b>	csx_RegisterClient – register a client
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_RegisterClient</b>(client_handle_t *ch, client_reg_t *cr);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i>            Pointer to a client_handle_t structure.</p> <p><i>mc</i>            Pointer to a client_reg_t structure.</p>
<b>DESCRIPTION</b>	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.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of client_reg_t are:</p> <pre>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];</pre> <p>The fields are defined as follows:</p> <p><b>Attributes</b></p> <p>This field is bit-mapped and defined as follows:</p> <p><b>INFO_MEM_CLIENT</b> Memory client device driver.</p> <p><b>INFO_MTD_CLIENT</b> Memory Technology Driver client.</p> <p><b>INFO_IO_CLIENT</b> IO client device driver.</p> <p><b>INFO_CARD_SHARE</b> Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events.</p> <p><b>INFO_CARD_EXCL</b> Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events.</p> <p><b>INFO_MEM_CLIENT</b> <b>INFO_MTD_CLIENT</b> <b>INFO_IO_CLIENT</b></p> <p>These bits are mutually exclusive (that is, only one bit may be set), but one of the bits must be set.</p>

## csx\_RegisterClient(9F)

INFO\_CARD\_SHARE

INFO\_CARD\_EXCL

If either of these bits is set, the client will receive a CS\_EVENT\_REGISTRATION\_COMPLETE event when Card Services has completed its internal client registration processing and after a successful call to csx\_RequestSocketMask(9F).

Also, if either of these bits is set, and if a card of the type that the client can control is currently inserted in the socket (and after a successful call to csx\_RequestSocketMask(9F)), the client will receive an artificial CS\_EVENT\_CARD\_INSERTION event.

### Event Mask

This field is bit-mapped and specifies the client's global event mask. Card Services performs event notification based on this field. See csx\_event\_handler(9E) for valid event definitions and for additional information about handling events.

### event\_callback\_args

The event\_callback\_args\_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.

## csx\_RegisterClient(9F)

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

## csx\_ReleaseConfiguration(9F)

<b>NAME</b>	csx_ReleaseConfiguration – release PC Card and socket configuration
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_ReleaseConfiguration(client_handle_t ch,                                 release_config_t *rc);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>rc</i> Pointer to a release_config_t structure.</p>
<b>DESCRIPTION</b>	<p>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).</p> <p>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).</p> <p>Card Services is prohibited from resetting the PC Card and is not required to cycle power through zero (0) volts.</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of release_config_t are:</p> <pre>uint32_t Socket; /* socket number */</pre> <p>The <i>Socket</i> field is not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p>
<b>RETURN VALUES</b>	<p><b>CS_SUCCESS</b> Successful operation.</p> <p><b>CS_BAD_HANDLE</b> Client handle is invalid or csx_RequestConfiguration(9F) not done.</p> <p><b>CS_BAD_SOCKET</b> Error getting or setting socket hardware parameters.</p> <p><b>CS_NO_CARD</b> No PC card in socket.</p> <p><b>CS_UNSUPPORTED_FUNCTION</b> No PCMCIA hardware installed.</p>

`csx_ReleaseConfiguration(9F)`

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

**SEE ALSO** `csx_RegisterClient(9F)`, `csx_RequestConfiguration(9F)`,  
`csx_RequestIO(9F)`, `csx_RequestIRQ(9F)`, `csx_RequestWindow(9F)`

*PC Card 95 Standard*, PCMCIA/JEIDA

## csx\_RepGet8(9F)

<b>NAME</b>	csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  void <b>csx_RepGet8</b>(acc_handle_t handle, uint8_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepGet16</b>(acc_handle_t handle, uint16_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepGet32</b>(acc_handle_t handle, uint32_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepGet64</b>(acc_handle_t handle, uint64_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)
	<i>handle</i> The access handle returned from <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> , or <code>csx_DupHandle(9F)</code> .
	<i>hostaddr</i> Source host address.
	<i>offset</i> The offset in bytes from the base of the mapped resource.
	<i>repcount</i> Number of data accesses to perform.
	<i>flags</i> Device address flags.
<b>DESCRIPTION</b>	<p>These functions generate multiple reads of various sizes from the mapped memory or device register.</p> <p>The <code>csx_RepGet8()</code>, <code>csx_RepGet16()</code>, <code>csx_RepGet32()</code>, and <code>csx_RepGet64()</code> functions generate <i>repcount</i> reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, <i>handle</i>, at an offset in bytes represented by the offset, <i>offset</i>. The data read is stored consecutively into the buffer pointed to by the host address pointer, <i>hostaddr</i>.</p> <p>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.</p> <p>When the <i>flags</i> argument is set to <code>CS_DEV_AUTOINCR</code>, these functions increment the device offset, <i>offset</i>, after each datum read operation. However, when the <i>flags</i> argument is set to <code>CS_DEV_NO_AUTOINCR</code>, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.</p>
<b>CONTEXT</b>	These functions may be called from user, kernel, or interrupt context.

csx\_RepGet8(9F)

**SEE ALSO** | csx\_DupHandle(9F), csx\_Get8(9F), csx\_GetMappedAddr(9F), csx\_Put8(9F),  
csx\_RepPut8(9F), csx\_RequestIO(9F), csx\_RequestWindow(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*

## csx\_RepPut8(9F)

<b>NAME</b>	csx_RepPut8, csx_RepPut16, csx_RepPut32, csx_RepPut64 – write repetitively to the device register
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  void <b>csx_RepPut8</b>(acc_handle_t handle, uint8_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepPut16</b>(acc_handle_t handle, uint16_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepPut32</b>(acc_handle_t handle, uint32_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);  void <b>csx_RepPut64</b>(acc_handle_t handle, uint64_t *hostaddr, uint32_t     offset, uint32_t repcount, uint32_t flags);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)
<i>handle</i>	The access handle returned from <code>csx_RequestIO(9F)</code> , <code>csx_RequestWindow(9F)</code> , or <code>csx_DupHandle(9F)</code> .
<i>hostaddr</i>	Source host address.
<i>offset</i>	The offset in bytes from the base of the mapped resource.
<i>repcount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags.
<b>DESCRIPTION</b>	<p>These functions generate multiple writes of various sizes to the mapped memory or device register.</p> <p>The <code>csx_RepPut8()</code>, <code>csx_RepPut16()</code>, <code>csx_RepPut32()</code>, and <code>csx_RepPut64()</code> functions generate <i>repcount</i> writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, <i>handle</i>, at an offset in bytes represented by the offset, <i>offset</i>. The data written is read consecutively from the buffer pointed to by the host address pointer, <i>hostaddr</i>.</p> <p>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.</p> <p>When the <i>flags</i> argument is set to <code>CS_DEV_AUTOINCR</code>, these functions increment the device offset, <i>offset</i>, after each datum write operation. However, when the <i>flags</i> argument is set to <code>CS_DEV_NO_AUTOINCR</code>, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.</p>
<b>CONTEXT</b>	These functions may be called from user, kernel, or interrupt context.

csx\_RepPut8(9F)

**SEE ALSO** | csx\_DupHandle(9F), csx\_Get8(9F), csx\_GetMappedAddr(9F), csx\_Put8(9F),  
csx\_RepGet8(9F), csx\_RequestIO(9F), csx\_RequestWindow(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*

## csx\_RequestConfiguration(9F)

<b>NAME</b>	csx_RequestConfiguration – configure the PC Card and socket
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_RequestConfiguration(client_handle_t ch, config_req_t     *cr);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI) <i>ch</i> Client handle returned from csx_RegisterClient(9F). <i>cr</i> Pointer to a config_req_t structure.
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of config_req_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> 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:</p> <p><b>CONF_ENABLE_IRQ_STEERING</b> 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()</p>

## csx\_RequestConfiguration(9F)

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 (CSR) present

#### CONFIG\_PINREPL\_REG\_PRESENT

Pin Replacement Register (PRR) present

#### CONFIG\_COPY\_REG\_PRESENT

Socket and Copy Register (SCR) present

#### CONFIG\_ESR\_REG\_PRESENT

Extended Status Register (ESR) present

## csx\_RequestConfiguration(9F)

Status, Pin, Copy, ExtendedStatus

These fields represent the initial values that should be written to those registers if they are present, as indicated by the `Present` field.

The `Pin` field is also used to inform Card Services which pins in the PC Card's PRR (Pin Replacement Register) are valid. Only those bits which are set are considered valid. This affects how status is returned by the `csx_GetStatus(9F)` function. If a particular signal is valid in the PRR, both the *mask* (STATUS) bit and the *change* (EVENT) bit must be set in the `Pin` field. The following PRR bit definitions are provided for client use:

PRR_WP_STATUS	WRITE PROTECT mask
PRR_READY_STATUS	READY mask
PRR_BVD2_STATUS	BVD2 mask
PRR_BVD1_STATUS	BVD1 mask
PRR_WP_EVENT	WRITE PROTECT changed
PRR_READY_EVENT	READY changed
PRR_BVD2_EVENT	BVD2 changed
PRR_BVD1_EVENT	BVD1 changed

ConfigIndex

This field is the value written to the COR (Configuration Option Register) for the configuration index required by the PC Card. Only the least significant six bits of the `ConfigIndex` field are significant; the upper two (2) bits are ignored. The interrupt type in the COR is always set to *level* mode by Card Services.

### RETURN VALUES

CS\_SUCCESS

Successful operation.

CS\_BAD\_HANDLE

Client handle is invalid or `csx_RequestConfiguration()` not done.

CS\_BAD\_SOCKET

Error in getting or setting socket hardware parameters.

CS\_BAD\_VCC

Requested `Vcc` is not available on socket.

CS\_BAD\_VPP

Requested `Vpp` is not available on socket.

CS\_NO\_CARD

No PC Card in socket.

CS\_BAD\_TYPE

I/O and memory interface not supported on socket.

CS\_CONFIGURATION\_LOCKED

`csx_RequestConfiguration()` already done.

csx\_RequestConfiguration(9F)

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*

## csx\_RequestIO(9F)

<b>NAME</b>	csx_RequestIO, csx_ReleaseIO – request or release I/O resources for the client
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir); int32_t csx_ReleaseIO(client_handle_t ch, io_req_t *ir);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>ir</i> Pointer to an io_req_t structure.</p>
<b>DESCRIPTION</b>	<p>The functions csx_RequestIO() and csx_ReleaseIO() request or release, respectively, I/O resources for the client.</p> <p>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.</p> <p>csx_RequestIO() fails if it has already been called without a corresponding csx_ReleaseIO().</p> <p>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.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of io_req_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p>Socket</p> <p>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p>

```
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.
```

## csx\_RequestIO(9F)

### 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

`csx_RequestIO(9F)`

**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.

<b>NAME</b>	csx_RequestIRQ, csx_ReleaseIRQ – request or release IRQ resource
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_RequestIRQ(client_handle_t ch, irq_req_t *ir); int32_t csx_ReleaseIRQ(client_handle_t ch, irq_req_t *ir);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>ir</i> Pointer to an irq_req_t structure.</p>
<b>DESCRIPTION</b>	<p>The function csx_RequestIRQ() requests an IRQ resource and registers the client's IRQ handler with Card Services.</p> <p>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.</p> <p>The function csx_ReleaseIRQ() releases a previously requested IRQ resource.</p> <p>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.</p> <p>csx_RequestIRQ() fails if it has already been called without a corresponding csx_ReleaseIRQ().</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of irq_req_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> This field is bit-mapped. It specifies details about the type of IRQ desired by the client. The following bits are defined:</p>

## csx\_RequestIRQ(9F)

`IRQ_TYPE_EXCLUSIVE`

IRQ is exclusive to this socket. This bit must be set. It indicates that the system IRQ is dedicated to this PC Card.

`irq_handler`

The client IRQ callback handler entry point is passed in the `irq_handler` field.

`irq_handler_arg`

The client can use the `irq_handler_arg` field to pass client-specific data to the client IRQ callback handler.

`iblk_cookie`

`idev_cookie`

These fields must be used by the client to set up mutexes that are used in the client's IRQ callback handler.

For a specific `csx_ReleaseIRQ()` call, the values in the `irq_req_t` structure must be the same as those returned from the previous `csx_RequestIRQ()` call; otherwise, `CS_BAD_ARGS` is returned and no changes are made to Card Services resources or the socket and adapter hardware.

## RETURN VALUES

`CS_SUCCESS`

Successful operation.

`CS_BAD_ARGS`

IRQ description does not match allocation.

`CS_BAD_ATTRIBUTE`

`IRQ_TYPE_EXCLUSIVE` not set, or an unsupported or reserved bit is set.

`CS_BAD_HANDLE`

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

`CS_BAD_IRQ`

Unable to allocate IRQ resources.

`CS_IN_USE`

`csx_RequestIRQ()` already done or a previous `csx_RequestIRQ()` has not been done for a corresponding `csx_ReleaseIRQ()`.

`CS_CONFIGURATION_LOCKED`

`csx_RequestConfiguration(9F)` already done or `csx_ReleaseConfiguration(9F)` has not been done.

`CS_NO_CARD`

No PC Card in socket.

`CS_UNSUPPORTED_FUNCTION`

No PCMCIA hardware installed.

## CONTEXT

These functions may be called from user or kernel context.

## SEE ALSO

`csx_ReleaseConfiguration(9F)`, `csx_RequestConfiguration(9F)`

csx\_RequestIRQ(9F)

*PC Card Card 95 Standard, PCMCIA/JEIDA*

## csx\_RequestSocketMask(9F)

<b>NAME</b>	csx_RequestSocketMask, csx_ReleaseSocketMask – set or clear the client’s client event mask						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  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);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)  <i>ch</i> Client handle returned from csx_RegisterClient(9F). <i>sm</i> Pointer to a request_socket_mask_t structure. <i>rm</i> Pointer to a release_socket_mask_t structure.						
<b>DESCRIPTION</b>	<p>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.</p> <p>csx_RequestSocketMask() must be used before calling csx_GetEventMask(9F) or csx_SetEventMask(9F) for the client event mask for this socket.</p> <p>The function csx_ReleaseSocketMask() clears the client’s client event mask.</p>						
<b>STRUCTURE MEMBERS</b>	<p>The structure members of request_socket_mask_t are:</p> <pre>uint32_t Socket; /* socket number */ uint32_t EventMask; /* event mask to set or return */</pre> <p>The structure members of release_socket_mask_t are:</p> <pre>uint32_t Socket; /* socket number */</pre> <p>The fields are defined as follows:</p> <table><tr><td>Socket</td><td>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</td></tr><tr><td>EventMask</td><td>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.</td></tr></table>	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.		
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.						
<b>RETURN VALUES</b>	<table><tr><td>CS_SUCCESS</td><td>Successful operation.</td></tr><tr><td>CS_BAD_HANDLE</td><td>Client handle is invalid.</td></tr><tr><td>CS_IN_USE</td><td>csx_ReleaseSocketMask() has not been done.</td></tr></table>	CS_SUCCESS	Successful operation.	CS_BAD_HANDLE	Client handle is invalid.	CS_IN_USE	csx_ReleaseSocketMask() has not been done.
CS_SUCCESS	Successful operation.						
CS_BAD_HANDLE	Client handle is invalid.						
CS_IN_USE	csx_ReleaseSocketMask() has not been done.						

csx\_RequestSocketMask(9F)

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

## csx\_RequestWindow(9F)

<b>NAME</b>	csx_RequestWindow, csx_ReleaseWindow – request or release window resources
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_RequestWindow(client_handle_t ch, window_handle_t     *wh, win_req_t *wr);  int32_t csx_ReleaseWindow(window_handle_t wh);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI) <i>ch</i> Client handle returned from csx_RegisterClient(9F). <i>wh</i> Pointer to a window_handle_t structure. <i>wr</i> Pointer to a win_req_t structure.
<b>DESCRIPTION</b>	<p>The function csx_RequestWindow() requests a block of system address space be assigned to a PC Card in a socket.</p> <p>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.</p> <p>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.</p> <p>The PC Card Attribute or Common Memory offset for this window is set by csx_MapMemPage(9F).</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of win_req_t are:</p> <pre>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 offset */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Attributes</b> This field is bit-mapped. It is defined as follows:</p>

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).

## csx\_RequestWindow(9F)

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 nsec
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.

## csx\_RequestWindow(9F)

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*

<b>NAME</b>	csx_ResetFunction – reset a function on a PC card	
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_ResetFunction(client_handle_t ch, reset_function_t     *rf);</pre>	
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI Specific (Solaris DDI)	
	<i>ch</i>	Client handle returned from csx_RegisterClient(9F).
	<i>rf</i>	Pointer to a reset_function_t structure.
<b>DESCRIPTION</b>	csx_ResetFunction() requests that the specified function on the PC card initiate a reset operation.	
<b>STRUCTURE MEMBERS</b>	<p>The structure members of reset_function_t are:</p> <pre>uint32_t    Socket;        /* socket number */ uint32_t    Attributes;    /* reset attributes */</pre> <p>The fields are defined as follows:</p> <p>Socket            Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p>Attributes        Must be 0.</p>	
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	This function may be called from user or kernel context.	
<b>SEE ALSO</b>	csx_event_handler(9E), csx_RegisterClient(9F)	
	<i>PC Card 95 Standard</i> , PCMCIA/JEIDA	
<b>NOTES</b>	csx_ResetFunction() has not been implemented in this release and always returns CS_IN_USE.	

## csx\_SetEventMask(9F)

<b>NAME</b>	csx_SetEventMask, csx_GetEventMask – set or return the client event mask for the client
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_SetEventMask(client_handle_t ch, sockevent_t *se); int32_t csx_GetEventMask(client_handle_t ch, sockevent_t *se);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>se</i> Pointer to a sockevent_t structure</p>
<b>DESCRIPTION</b>	<p>The function csx_SetEventMask() sets the client or global event mask for the client.</p> <p>The function csx_GetEventMask() returns the client or global event mask for the client.</p> <p>csx_RequestSocketMask(9F) must be called before calling csx_SetEventMask() for the client event mask for this socket.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of sockevent_t are:</p> <pre>uint32_t  uint32_t      /* attribute flags for call */ uint32_t  EventMask;   /* event mask to set or return */ uint32_t  Socket;      /* socket number if necessary */</pre> <p>The fields are defined as follows:</p> <p><b>Attributes</b></p> <p>This is a bit-mapped field that identifies the type of event mask to be returned. The field is defined as follows:</p> <p>CONF_EVENT_MASK_GLOBAL Client's global event mask. If set, the client's global event mask is returned.</p> <p>CONF_EVENT_MASK_CLIENT Client's local event mask. If set, the client's local event mask is returned.</p> <p><b>EventMask</b></p> <p>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.</p> <p><b>Socket</b></p> <p>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p>
<b>RETURN VALUES</b>	<p>CS_SUCCESS Successful operation.</p> <p>CS_BAD_HANDLE Client handle is invalid.</p>

`csx_SetEventMask(9F)`

`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

## csx\_SetHandleOffset(9F)

<b>NAME</b>	csx_SetHandleOffset – set current access handle offset				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t <b>csx_SetHandleOffset</b>(acc_handle_t <i>handle</i>, uint32_t <i>offset</i>);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)				
<b>PARAMETERS</b>	<table><tr><td><i>handle</i></td><td>Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).</td></tr><tr><td><i>offset</i></td><td>New access handle offset.</td></tr></table>	<i>handle</i>	Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).	<i>offset</i>	New access handle offset.
<i>handle</i>	Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).				
<i>offset</i>	New access handle offset.				
<b>DESCRIPTION</b>	This function sets the current offset for the access handle, <i>handle</i> , to <i>offset</i> .				
<b>RETURN VALUES</b>	CS_SUCCESS Successful operation.				
<b>CONTEXT</b>	This function may be called from user or kernel context.				
<b>SEE ALSO</b>	csx_GetHandleOffset(9F), csx_RequestIO(9F), csx_RequestIRQ(9F) <i>PC Card 95 Standard</i> , <i>PCMCIA/JEIDA</i>				

<b>NAME</b>	csx_ValidateCIS – validate the Card Information Structure (CIS)
<b>SYNOPSIS</b>	<pre>#include &lt;sys/pccard.h&gt;  int32_t csx_ValidateCIS(client_handle_t ch, cisinfo_t *ci);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI Specific (Solaris DDI)</p> <p><i>ch</i> Client handle returned from csx_RegisterClient(9F).</p> <p><i>ci</i> Pointer to a cisinfo_t structure.</p>
<b>DESCRIPTION</b>	This function validates the Card Information Structure (CIS) on the PC Card in the specified socket.
<b>STRUCTURE MEMBERS</b>	<p>The structure members of cisinfo_t are:</p> <pre>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 */</pre> <p>The fields are defined as follows:</p> <p><b>Socket</b> Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</p> <p><b>Chains</b> This field returns the number of valid tuple chains located in the CIS. If 0 is returned, the CIS is not valid.</p> <p><b>Tuples</b> 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.</p>
<b>RETURN VALUES</b>	<p><b>CS_SUCCESS</b> Successful operation.</p> <p><b>CS_NO_CIS</b> No CIS on PC Card or CIS is invalid.</p> <p><b>CS_NO_CARD</b> No PC Card in socket.</p> <p><b>CS_UNSUPPORTED_FUNCTION</b> No PCMCIA hardware installed.</p>
<b>CONTEXT</b>	This function may be called from user or kernel context.
<b>SEE ALSO</b>	<p>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_ParseTuple(9F), csx_RegisterClient(9F)</p> <p><i>PC Card 95 Standard</i>, PCMCIA/JEIDA</p>

## datamsq(9F)

<b>NAME</b>	datamsq – test whether a message is a data message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  int datamsq(unsigned char type);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>type</i> The type of message to be tested. The <i>db_type</i> field of the <i>datab(9S)</i> structure contains the message type. This field may be accessed through the message block using <i>mp-&gt;b_datap-&gt;db_type</i> .
<b>DESCRIPTION</b>	<i>datamsq()</i> tests the type of message to determine if it is a data message type ( <i>M_DATA</i> , <i>M_DELAY</i> , <i>M_PROTO</i> , or <i>M_PCPROTO</i> ).
<b>RETURN VALUES</b>	<i>datamsq</i> returns 1 if the message is a data message 0 otherwise.
<b>CONTEXT</b>	<i>datamsq()</i> can be called from user or interrupt context.
<b>EXAMPLES</b>	<b>EXAMPLE 1</b> The <i>put(9E)</i> routine enqueues all data messages for handling by the <i>srv(9E)</i> (service) routine. All non-data messages are handled in the <i>put(9E)</i> routine. <pre>1 xxxput(q, mp) 2     queue_t *q; 3     mblk_t *mp; 4 { 5     if (datamsq(mp-&gt;b_datap-&gt;db_type)) { 6         putq(q, mp); 7         return; 8     } 9     switch (mp-&gt;b_datap-&gt;db_type) { 10    case M_FLUSH: 11        ... 12    }</pre>
<b>SEE ALSO</b>	<i>put(9E)</i> , <i>srv(9E)</i> , <i>alloca(9F)</i> , <i>datab(9S)</i> , <i>msgb(9S)</i>  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>

<b>NAME</b>	ddi_add_intr, ddi_get_iblock_cookie, ddi_remove_intr – hardware interrupt handling routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p>For ddi_get_iblock_cookie():</p> <p><i>dip</i>                    Pointer to dev_info structure.</p> <p><i>inumber</i>                Interrupt number.</p> <p><i>iblock_cookiep</i>        Pointer to an interrupt block cookie.</p> <p>For ddi_add_intr():</p> <p><i>dip</i>                    Pointer to dev_info structure.</p> <p><i>inumber</i>                Interrupt number.</p> <p><i>iblock_cookiep</i>        Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.</p> <p><i>idevice_cookiep</i>        Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.</p> <p><i>int_handler</i>            Pointer to interrupt handler.</p> <p><i>int_handler_arg</i>        Argument for interrupt handler.</p> <p>For ddi_remove_intr():</p> <p><i>dip</i>                    Pointer to dev_info structure.</p> <p><i>inumber</i>                Interrupt number.</p> <p><i>iblock_cookie</i>        Block cookie which identifies the interrupt handler to be removed.</p>
<b>DESCRIPTION</b>	

`ddi_add_intr(9F)`

**`ddi_get_iblock_cookie()`** `ddi_get_iblock_cookie()` retrieves the interrupt block cookie associated with a particular interrupt specification. This routine should be called before `ddi_add_intr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the interrupt routine. The interrupt number *inumber* determines for which interrupt specification to retrieve the cookie. *inumber* is associated with information provided either by the device (see `sbus(4)`) or the hardware configuration file (see `sysbus(4)`, `isa(4)`, `eisa(4)`, and `driver.conf(4)`). If only one interrupt is associated with the device, *inumber* should be 0.

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

**`ddi_remove_intr()`** `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 <code>ddi_remove_intr()</code> returns. <code>ddi_remove_intr()</code> 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 <code>ddi_remove_intr()</code> or deadlock may result.						
<b>For All Three Functions:</b>	For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See <code>sysbus(4)</code> , <code>isa(4)</code> , <code>eisa(4)</code> , and <code>sbus(4)</code> for details.						
<b>RETURN VALUES</b>	<p><code>ddi_add_intr()</code> and <code>ddi_get_iblock_cookie()</code> return:</p> <table border="0"> <tr> <td><code>DDI_SUCCESS</code></td> <td>On success.</td> </tr> <tr> <td><code>DDI_INTR_NOTFOUND</code></td> <td>On failure to find the interrupt.</td> </tr> <tr> <td><code>DDI_FAILURE</code></td> <td>On failure. <code>DDI_FAILURE</code> can also be returned on failure to find interrupt (<code>sun4u</code>).</td> </tr> </table>	<code>DDI_SUCCESS</code>	On success.	<code>DDI_INTR_NOTFOUND</code>	On failure to find the interrupt.	<code>DDI_FAILURE</code>	On failure. <code>DDI_FAILURE</code> can also be returned on failure to find interrupt ( <code>sun4u</code> ).
<code>DDI_SUCCESS</code>	On success.						
<code>DDI_INTR_NOTFOUND</code>	On failure to find the interrupt.						
<code>DDI_FAILURE</code>	On failure. <code>DDI_FAILURE</code> can also be returned on failure to find interrupt ( <code>sun4u</code> ).						
<b>CONTEXT</b>	<code>ddi_add_intr()</code> , <code>ddi_remove_intr()</code> , and <code>ddi_get_iblock_cookie()</code> can be called from user or kernel context.						
<b>SEE ALSO</b>	<p><code>driver.conf(4)</code>, <code>eisa(4)</code>, <code>isa(4)</code>, <code>sbus(4)</code>, <code>sysbus(4)</code>, <code>attach(9E)</code>, <code>detach(9E)</code>, <code>ddi_intr_hilevel(9F)</code>, <code>mutex(9F)</code>, <code>mutex_init(9F)</code>, <code>rw_init(9F)</code>, <code>rwlock(9F)</code>, <code>ddi_idevice_cookie(9S)</code></p> <p><i>Writing Device Drivers</i></p>						
<b>NOTES</b>	<p><code>ddi_get_iblock_cookie()</code> must not be called <i>after</i> the driver adds an interrupt handler for the interrupt specification corresponding to <i>inumber</i>.</p> <p>All consumers of these interfaces, checking return codes, should verify <code>return_code != DDI_SUCCESS</code>. Checking for specific failure codes can result in inconsistent behaviors among platforms.</p>						
<b>BUGS</b>	The <code>idevice_cookiep</code> should really point to a data structure that is specific to the bus architecture that the device operates on. Currently the SBus and PCI buses are supported and a single data structure is used to describe both.						

## ddi\_add\_softintr(9F)

<b>NAME</b>	ddi_add_softintr, ddi_get_soft_iblock_cookie, ddi_remove_softintr, ddi_trigger_softintr – software interrupt handling routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p>ddi_get_soft_iblock_cookie()</p> <p><i>dip</i> Pointer to a dev_info structure.</p> <p><i>preference</i> The type of soft interrupt to retrieve the cookie for.</p> <p><i>iblock_cookiep</i> Pointer to a location to store the interrupt block cookie.</p> <p>ddi_add_softintr()</p> <p><i>dip</i> Pointer to dev_info structure.</p> <p><i>preference</i> A hint value describing the type of soft interrupt to generate.</p> <p><i>idp</i> Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.</p> <p><i>iblock_cookiep</i> Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.</p> <p><i>idevice_cookiep</i> Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).</p> <p><i>int_handler</i> Pointer to interrupt handler.</p> <p><i>int_handler_arg</i> Argument for interrupt handler.</p> <p>ddi_remove_softintr()</p> <p><i>id</i> The identifier specifying which soft interrupt handler to remove.</p> <p>ddi_trigger_softintr()</p> <p><i>id</i> The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.</p>

**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`.

## ddi\_add\_softintr(9F)

The routine *int\_handler*, with its argument *int\_handler\_arg*, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine *int\_handler* checks this flag, reachable through *int\_handler\_arg*, to determine if it should claim the interrupt and do its work.

The interrupt handler must return `DDI_INTR_CLAIMED` if the interrupt was claimed, `DDI_INTR_UNCLAIMED` otherwise.

If successful, `ddi_add_softintr()` will return `DDI_SUCCESS`; if the interrupt information cannot be found, it will return `DDI_FAILURE`.

For `ddi_remove_softintr()`:

`ddi_remove_softintr()` removes a soft interrupt from the system. The soft interrupt identifier *id*, which was returned from a call to `ddi_add_softintr()`, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For `ddi_trigger_softintr()`:

`ddi_trigger_softintr()` triggers a soft interrupt. The soft interrupt identifier *id* is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using `ddi_add_softintr()`.

### RETURN VALUES

`ddi_add_softintr()` and `ddi_get_soft_iblock_cookie()` return:

`DDI_SUCCESS`                      on success

`DDI_FAILURE`                      on failure

### CONTEXT

These functions can be called from user or kernel context. `ddi_trigger_softintr()` may be called from high-level interrupt context as well.

### EXAMPLES

**EXAMPLE 1** device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See `ddi_intr_hilevel(9F)`.

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

In the example, the high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt handler. If the soft interrupt handler is not currently running, the high-level interrupt routine triggers a soft interrupt so the soft interrupt handler can process the data. Once running, the soft interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data shared between the high-level interrupt handler and the soft interrupt handler. The low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
    . . .
    ddi_softintr_t      id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t           high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t           low_mutex;
    int                softint_running;
    . . .
};
struct xxstate *xsp;
static uint_t xxsoftintr(caddr_t);
static uint_t xxhighintr(caddr_t);
. . .
```

**EXAMPLE 2** sample attach() routine

The following code fragment would usually appear in the driver's attach(9E) routine. `ddi_add_intr(9F)` is used to add the high-level interrupt handler and `ddi_add_softintr()` is used to add the low-level interrupt routine.

```
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    . . .
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber,
        &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
        (void *)xsp->high_iblock_cookie);

    /* add high-level routine - xxhighintr() */
    if (ddi_add_intr(dip, inumber, NULL, NULL,
        xxhighintr, (caddr_t) xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }
}
```

## ddi\_add\_softintr(9F)

### EXAMPLE 2 sample attach() routine (Continued)

```
    }

    /* get soft iblock cookie */
    if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
        &xsp->low_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize low-level mutex */
    mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
        (void *)xsp->low_iblock_cookie);

    /* add low level routine - xxsoftintr() */
    if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
        NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }

    . . .
}
```

### EXAMPLE 3 High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, `ddi_trigger_softintr()` is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```
static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    . . .
    mutex_enter(&xsp->high_mutex);
    /*
     * Verify this device generated the interrupt
     * and disable the device interrupt.
     * Enqueue data for xxsoftintr() processing.
     */

    /* is xxsoftintr() already running ? */
    if (xsp->softint_running)
        need_softint = 0;
    else
        need_softint = 1;
    mutex_exit(&xsp->high_mutex);

    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
```

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

```

        ddi_trigger_softintr(xsp->id);
        . . .
        return (DDI_INTR_CLAIMED);
    }

static uint_t
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    . . .
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);

    /* verify there is work to do */
    if (work_queue_empty || xsp->softint_running) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }

    xsp->softint_running = 1;

    while ( data on queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));

        /* de-queue data */

        mutex_exit(&xsp->high_mutex);

        /* Process data on queue */

        mutex_enter(&xsp->high_mutex);
    }

    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);

    return (DDI_INTR_CLAIMED);
}

```

**SEE ALSO** ddi\_add\_intr(9F), ddi\_in\_panic(9F), ddi\_intr\_hilevel(9F),  
ddi\_remove\_intr(9F), mutex\_init(9F)

*Writing Device Drivers*

**NOTES** ddi\_add\_softintr() may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for *int\_handler\_arg* in each of the calls to ddi\_add\_softintr(). Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For

## ddi\_add\_softintr(9F)

example, the argument could be a pointer to the device's soft state structure, which could contain a 'which\_service' field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_trigger_softintr()`.

<b>NAME</b>	ddi_binding_name, ddi_get_name – return driver binding name
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  char *ddi_binding_name(dev_info_t *dip); char *ddi_get_name(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> A pointer to the device's dev_info structure.
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	ddi_binding_name() and ddi_get_name() return the name used to bind a driver to a device.
<b>CONTEXT</b>	ddi_binding_name() and ddi_get_name() can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	ddi_node_name(9F) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	The name returned by ddi_binding_name() and ddi_get_name() is read-only.

## ddi\_btop(9F)

<b>NAME</b>	ddi_btop, ddi_btopr, ddi_ptob – page size conversions
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	<p>This set of routines use the parent nexus driver to perform conversions in page size units.</p> <p>ddi_btop() converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.</p> <p>ddi_btopr() converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.</p> <p>ddi_ptob() converts the given number of pages to the number of bytes that it corresponds to.</p> <p>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.</p>
<b>RETURN VALUES</b>	ddi_btop() and ddi_btopr() return the number of corresponding pages. ddi_ptob() returns the corresponding number of bytes. There are no error return values.
<b>CONTEXT</b>	This function can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Find the size (in bytes) of one page</p> <pre>pagesize = ddi_ptob(dip, 1L);</pre>
<b>SEE ALSO</b>	btop(9F), btopr(9F), ptob(9F) <i>Writing Device Drivers</i>

ddi\_check\_acc\_handle(9F)

<b>NAME</b>	ddi_check_acc_handle, ddi_check_dma_handle – Check data access and DMA handles
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_check_acc_handle(ddi_acc_handle_t acc_handle ); int ddi_check_dma_handle(ddi_dma_handle_t dma_handle );</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>acc_handle</i> Data access handle obtained from a previous call to ddi_regs_map_setup(9F), ddi_dma_mem_alloc(9F), or similar function.</p> <p><i>dma_handle</i> DMA handle obtained from a previous call to ddi_dma_setup(9F) or one of its derivatives.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p> <p>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.</p> <p><b>Note</b> – 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 <i>does</i> exist with respect to communication using that handle.</p>

## ddi\_check\_acc\_handle(9F)

**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)`

<b>NAME</b>	ddi_copyin – copy data to a driver buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_copyin</b>(const void *buf, void *driverbuf, size_t cn, int flags);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>buf</i>                      Source address from which data is transferred.</p> <p><i>driverbuf</i>                Driver destination address to which data is transferred.</p> <p><i>cn</i>                         Number of bytes transferred.</p> <p><i>flags</i>                     Set of flag bits that provide address space information about <i>buf</i>.</p>
<b>DESCRIPTION</b>	<p>This routine is designed for use in driver <code>ioctl(9E)</code> routines for drivers that support layered ioctls. <code>ddi_copyin()</code> copies data from a source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.</p> <p>The <i>flags</i> argument determines the address space information about <i>buf</i>. If the <code>FKIOCTL</code> flag is set, this indicates that <i>buf</i> is a kernel address, and <code>ddi_copyin()</code> behaves like <code>bcopy(9F)</code>. Otherwise, <i>buf</i> is interpreted as a user buffer address, and <code>ddi_copyin()</code> behaves like <code>copyin(9F)</code>.</p> <p>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.</p>
<b>RETURN VALUES</b>	<p><code>ddi_copyin()</code> returns 0, indicating a successful copy. It returns -1 if one of the following occurs:</p> <ul style="list-style-type: none"> <li>■ Paging fault; the driver tried to access a page of memory for which it did not have read or write access.</li> <li>■ Invalid user address, such as a user area or stack area.</li> <li>■ Invalid address that would have resulted in data being copied into the user block.</li> <li>■ Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.</li> </ul> <p>If -1 is returned to the caller, driver entry point routines should return <code>EFAULT</code>.</p>
<b>CONTEXT</b>	<code>ddi_copyin()</code> can be called from user or kernel context only.

## ddi\_copyin(9F)

### 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
20     instance = getminor(dev);
21     sp = ddi_get_soft_state(statep, instance);
22     if (sp == NULL)
23         return (ENXIO);
24     rp = sp->regsp;
25     switch (cmd) {
26     case XX_GETREGS: /* copy data to temp. regs. buf */
27         if (ddi_copyin(arg, &reg_buf,
28             sizeof (struct device), mode) != 0) {
29             return (EFAULT);
30         }
31         mutex_enter(&sp->reg_mutex);
32         /*
33          * Copy data from temporary device register
34          * buffer to device registers.
35          * e.g. rp->control = reg_buf.control;
36          */
37         mutex_exit(&sp->reg_mutex);
38
39         break;
40     }
41 }
```

**EXAMPLE 1** ddi\_copyin() example (Continued)

**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*.

## ddi\_copyout(9F)

<b>NAME</b>	ddi_copyout – copy data from a driver								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_copyout</b>(const void *driverbuf, void *buf, size_t cn, int     flags);</pre>								
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).								
<b>PARAMETERS</b>	<table><tr><td><i>driverbuf</i></td><td>Source address in the driver from which the data is transferred.</td></tr><tr><td><i>buf</i></td><td>Destination address to which the data is transferred.</td></tr><tr><td><i>cn</i></td><td>Number of bytes to copy.</td></tr><tr><td><i>flags</i></td><td>Set of flag bits that provide address space information about <i>buf</i>.</td></tr></table>	<i>driverbuf</i>	Source address in the driver from which the data is transferred.	<i>buf</i>	Destination address to which the data is transferred.	<i>cn</i>	Number of bytes to copy.	<i>flags</i>	Set of flag bits that provide address space information about <i>buf</i> .
<i>driverbuf</i>	Source address in the driver from which the data is transferred.								
<i>buf</i>	Destination address to which the data is transferred.								
<i>cn</i>	Number of bytes to copy.								
<i>flags</i>	Set of flag bits that provide address space information about <i>buf</i> .								
<b>DESCRIPTION</b>	<p>This routine is designed for use in driver <code>ioctl(9E)</code> routines for drivers that support layered <code>ioctls</code>. <code>ddi_copyout()</code> copies data from a driver buffer to a destination address, <i>buf</i>.</p> <p>The <i>flags</i> argument determines the address space information about <i>buf</i>. If the <code>FKIOCTL</code> flag is set, this indicates that <i>buf</i> is a kernel address, and <code>ddi_copyout()</code> behaves like <code>bcopy(9F)</code>. Otherwise, <i>buf</i> is interpreted as a user buffer address, and <code>ddi_copyout()</code> behaves like <code>copyout(9F)</code>.</p> <p>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.</p>								
<b>RETURN VALUES</b>	<p>Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if one of the following occurs:</p> <ul style="list-style-type: none"><li>■ Paging fault; the driver tried to access a page of memory for which it did not have read or write access.</li><li>■ Invalid user address, such as a user area or stack area.</li><li>■ Invalid address that would have resulted in data being copied into the user block.</li><li>■ Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.</li></ul> <p>If -1 is returned to the caller, driver entry point routines should return <code>EFAULT</code>.</p>								
<b>CONTEXT</b>	<code>ddi_copyout()</code> 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 };

12 static void *statep; /* for soft state routines */

13 xxioctl(dev_t dev, int cmd, int arg, int mode,
14         cred_t *cred_p, int *rval_p)
15 {
16     struct device_state *sp;
17     volatile struct device *rp;
18     struct device reg_buf; /* temporary buffer for registers */
19     int instance;

20     instance = getminor(dev);
21     sp = ddi_get_soft_state(statep, instance);
22     if (sp == NULL)
23         return (ENXIO);
24     rp = sp->regsp;
25     . . .
26     switch (cmd) {
27
28     case XX_GETREGS: /* copy registers to arg */
29         mutex_enter(&sp->reg_mutex);
30         /*
31          * Copy data from device registers to
32          * temporary device register buffer
33          * e.g. reg_buf.control = rp->control;
34          */
35         mutex_exit(&sp->reg_mutex);
36         if (ddi_copyout(&reg_buf, arg,
37             sizeof (struct device), mode) != 0) {
38             return (EFAULT);
39         }
40     }
41     break;
42 }
43 }

```

## ddi\_copyout(9F)

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

**SEE ALSO** `ioctl(9E)`, `bcopy(9F)`, `copyin(9F)`, `copyout(9F)`, `ddi_copyin(9F)`, `uiomove(9F)`

*Writing Device Drivers*

**NOTES** The value of the *flags* argument to `ddi_copyout()` should be passed through directly from the *mode* argument of `ioctl()` untranslated.

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

`ddi_copyout()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*.

<b>NAME</b>	ddi_create_minor_node – create a minor node for this device																										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stat.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_create_minor_node(dev_info_t *dip, char *name, int spec_type,     minor_t minor_num, char *node_type, int flag);</pre>																										
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i>                    A pointer to the device's dev_info structure.</p> <p><i>name</i>                    The name of this particular minor device.</p> <p><i>spec_type</i>                S_IFCHR or S_IFBLK for character or block minor devices respectively.</p> <p><i>minor_num</i>                The minor number for this particular minor device.</p> <p><i>node_type</i>                Any string that uniquely identifies the type of node. The following predefined node types are provided with this release:</p> <table border="0" style="margin-left: 2em;"> <tr><td>DDI_NT_SERIAL</td><td>For serial ports</td></tr> <tr><td>DDI_NT_SERIAL_MB</td><td>For on board serial ports</td></tr> <tr><td>DDI_NT_SERIAL_DO</td><td>For dial out ports</td></tr> <tr><td>DDI_NT_SERIAL_MB_DO</td><td>For on board dial out ports</td></tr> <tr><td>DDI_NT_BLOCK</td><td>For hard disks</td></tr> <tr><td>DDI_NT_BLOCK_CHAN</td><td>For hard disks with channel or target numbers</td></tr> <tr><td>DDI_NT_CD</td><td>For CDROM drives</td></tr> <tr><td>DDI_NT_CD_CHAN</td><td>For CDROM drives with channel or target numbers</td></tr> <tr><td>DDI_NT_FD</td><td>For floppy disks</td></tr> <tr><td>DDI_NT_TAPE</td><td>For tape drives</td></tr> <tr><td>DDI_NT_NET</td><td>For DLPI style 1 or style 2 network devices</td></tr> <tr><td>DDI_NT_DISPLAY</td><td>For display devices</td></tr> <tr><td>DDI_PSEUDO</td><td>For pseudo devices</td></tr> </table> <p><i>flag</i>                    If the device is a clone device then this flag is set to CLONE_DEV else it is set to 0.</p>	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
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																										
<b>DESCRIPTION</b>	<p>ddi_create_minor_node() provides the necessary information to enable the system to create the /dev and /devices hierarchies. The <i>name</i> 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 <i>spec_type</i> specifies whether this is</p>																										

## ddi\_create\_minor\_node(9F)

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.

<b>NAME</b>	ddi_device_copy – copy data from one device register to another device register																								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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 bytcount, uint_t dev_datsz);</pre>																								
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <table border="0"> <tr> <td><i>src_handle</i></td> <td>The data access handle of the source device.</td> </tr> <tr> <td><i>src_addr</i></td> <td>Base data source address.</td> </tr> <tr> <td><i>src_advcnt</i></td> <td>Number of <i>dev_datsz</i> units to advance on every access.</td> </tr> <tr> <td><i>dest_handle</i></td> <td>The data access handle of the destination device.</td> </tr> <tr> <td><i>dest_addr</i></td> <td>Base data destination address.</td> </tr> <tr> <td><i>dest_advcnt</i></td> <td>Number of <i>dev_datsz</i> units to advance on every access.</td> </tr> <tr> <td><i>bytcount</i></td> <td>Number of bytes to transfer.</td> </tr> <tr> <td><i>dev_datsz</i></td> <td>The size of each data word. Possible values are defined as:</td> </tr> <tr> <td></td> <td>DDI_DATA_SZ01_ACC      1 byte data size</td> </tr> <tr> <td></td> <td>DDI_DATA_SZ02_ACC      2 bytes data size</td> </tr> <tr> <td></td> <td>DDI_DATA_SZ04_ACC      4 bytes data size</td> </tr> <tr> <td></td> <td>DDI_DATA_SZ08_ACC      8 bytes data size</td> </tr> </table>	<i>src_handle</i>	The data access handle of the source device.	<i>src_addr</i>	Base data source address.	<i>src_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.	<i>dest_handle</i>	The data access handle of the destination device.	<i>dest_addr</i>	Base data destination address.	<i>dest_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.	<i>bytcount</i>	Number of bytes to transfer.	<i>dev_datsz</i>	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
<i>src_handle</i>	The data access handle of the source device.																								
<i>src_addr</i>	Base data source address.																								
<i>src_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.																								
<i>dest_handle</i>	The data access handle of the destination device.																								
<i>dest_addr</i>	Base data destination address.																								
<i>dest_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.																								
<i>bytcount</i>	Number of bytes to transfer.																								
<i>dev_datsz</i>	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																								
<b>DESCRIPTION</b>	<p>ddi_device_copy() copies <i>bytcount</i> bytes from the source address, <i>src_addr</i>, to the destination address, <i>dest_addr</i>. The attributes encoded in the access handles, <i>src_handle</i> and <i>dest_handle</i>, govern how data is actually copied from the source to the destination. Only matching data sizes between the source and destination are supported.</p> <p>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.</p> <p>The <i>src_advcnt</i> and <i>dest_advcnt</i> arguments specifies the number of <i>dev_datsz</i> 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.</p> <p>The <i>dev_datsz</i> argument determines the size of the data word on each access. The data size must be the same between the source and destination.</p>																								
<b>RETURN VALUES</b>	<p>ddi_device_copy() returns:</p> <table border="0"> <tr> <td>DDI_SUCCESS</td> <td>Successfully transferred the data.</td> </tr> </table>	DDI_SUCCESS	Successfully transferred the data.																						
DDI_SUCCESS	Successfully transferred the data.																								

## ddi\_device\_copy(9F)

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*

<b>NAME</b>	ddi_device_zero – zero fill the device								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_device_zero(ddi_acc_handle_t handle, caddr_t dev_addr,     size_t bytecount, ssize_t dev_advcnt, uint_t dev_datsz);</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</p> <p><i>dev_addr</i>           Beginning of the device address.</p> <p><i>bytecount</i>          Number of bytes to zero.</p> <p><i>dev_advcnt</i>         Number of <i>dev_datsz</i> units to advance on every access.</p> <p><i>dev_datsz</i>          The size of each data word. Possible values are defined as:</p> <table border="0" style="margin-left: 40px;"> <tr> <td>DDI_DATA_SZ01_ACC</td> <td>1 byte data size</td> </tr> <tr> <td>DDI_DATA_SZ02_ACC</td> <td>2 bytes data size</td> </tr> <tr> <td>DDI_DATA_SZ04_ACC</td> <td>4 bytes data size</td> </tr> <tr> <td>DDI_DATA_SZ08_ACC</td> <td>8 bytes data size</td> </tr> </table>	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
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								
<b>DESCRIPTION</b>	<p>ddi_device_zero() function fills the given, <i>bytecount</i>, number of byte of zeroes to the device register or memory.</p> <p>The <i>dev_advcnt</i> argument determines the value of the device address, <i>dev_addr</i>, on each access. A value of 0 will use the same device address, <i>dev_addr</i>, 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 <i>dev_datsz</i> units.</p> <p>The <i>dev_datsz</i> argument determines the size of data word on each access.</p>								
<b>RETURN VALUES</b>	<p>ddi_device_zero() returns:</p> <p>DDI_SUCCESS        Successfully zeroed the data.</p> <p>DDI_FAILURE        The byte count is not a multiple of <i>dev_datsz</i>.</p>								
<b>CONTEXT</b>	ddi_device_zero() can be called from user, kernel, or interrupt context.								
<b>SEE ALSO</b>	<p>ddi_regs_map_free(9F), ddi_regs_map_setup(9F)</p> <p><i>Writing Device Drivers</i></p>								

## ddi\_devid\_compare(9F)

<b>NAME</b>	ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register, ddi_devid_sizeof, ddi_devid_str_decode, ddi_devid_str_encode, ddi_devid_str_free, ddi_devid_unregister, ddi_devid_valid – kernel interfaces for device ids																						
<b>SYNOPSIS</b>	<pre>int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);  size_t ddi_devid_sizeof(ddi_devid_t devid);  int ddi_devid_init(dev_info_t *dip, ushort_t devid_type, ushort_t     nbytes, void *id, ddi_devid_t *retdevid);  void ddi_devid_free(ddi_devid_t devid);  int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);  int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid, char     **retminor_name);  int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);  int ddi_devid_str_free(char *devidstr);  void ddi_devid_unregister(dev_info_t *dip);  int ddi_devid_valid(ddi_devid_t devid);</pre>																						
<b>PARAMETERS</b>	<table><tr><td><i>devid</i></td><td>The device id address.</td></tr><tr><td><i>devidstr</i></td><td>The <i>devid</i> and <i>minor_name</i> represented as a string.</td></tr><tr><td><i>devid1</i></td><td>The first of two device id addresses to be compared calling <code>ddi_devid_compare()</code>.</td></tr><tr><td><i>devid2</i></td><td>The second of two device id addresses to be compared calling <code>ddi_devid_compare()</code>.</td></tr><tr><td><i>dip</i></td><td>A <code>dev_info</code> pointer, which identifies the device.</td></tr><tr><td><i>devid_type</i></td><td>The following device id types may be accepted by the <code>ddi_devid_init()</code> function: <table><tr><td>DEVID_SCSI3_WWN</td><td>World Wide Name associated with SCSI-3 devices.</td></tr><tr><td>DEVID_SCSI_SERIAL</td><td>Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.</td></tr><tr><td>DEVID_ENCAP</td><td>Device ID of another device. This is for layered device driver usage.</td></tr><tr><td>DEVID_FAB</td><td>Fabricated device ID.</td></tr></table></td></tr><tr><td><i>minor_name</i></td><td>The minor name to be encoded.</td></tr></table>	<i>devid</i>	The device id address.	<i>devidstr</i>	The <i>devid</i> and <i>minor_name</i> represented as a string.	<i>devid1</i>	The first of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .	<i>devid2</i>	The second of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .	<i>dip</i>	A <code>dev_info</code> pointer, which identifies the device.	<i>devid_type</i>	The following device id types may be accepted by the <code>ddi_devid_init()</code> function: <table><tr><td>DEVID_SCSI3_WWN</td><td>World Wide Name associated with SCSI-3 devices.</td></tr><tr><td>DEVID_SCSI_SERIAL</td><td>Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.</td></tr><tr><td>DEVID_ENCAP</td><td>Device ID of another device. This is for layered device driver usage.</td></tr><tr><td>DEVID_FAB</td><td>Fabricated device ID.</td></tr></table>	DEVID_SCSI3_WWN	World Wide Name associated with SCSI-3 devices.	DEVID_SCSI_SERIAL	Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.	DEVID_ENCAP	Device ID of another device. This is for layered device driver usage.	DEVID_FAB	Fabricated device ID.	<i>minor_name</i>	The minor name to be encoded.
<i>devid</i>	The device id address.																						
<i>devidstr</i>	The <i>devid</i> and <i>minor_name</i> represented as a string.																						
<i>devid1</i>	The first of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .																						
<i>devid2</i>	The second of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .																						
<i>dip</i>	A <code>dev_info</code> pointer, which identifies the device.																						
<i>devid_type</i>	The following device id types may be accepted by the <code>ddi_devid_init()</code> function: <table><tr><td>DEVID_SCSI3_WWN</td><td>World Wide Name associated with SCSI-3 devices.</td></tr><tr><td>DEVID_SCSI_SERIAL</td><td>Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.</td></tr><tr><td>DEVID_ENCAP</td><td>Device ID of another device. This is for layered device driver usage.</td></tr><tr><td>DEVID_FAB</td><td>Fabricated device ID.</td></tr></table>	DEVID_SCSI3_WWN	World Wide Name associated with SCSI-3 devices.	DEVID_SCSI_SERIAL	Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.	DEVID_ENCAP	Device ID of another device. This is for layered device driver usage.	DEVID_FAB	Fabricated device ID.														
DEVID_SCSI3_WWN	World Wide Name associated with SCSI-3 devices.																						
DEVID_SCSI_SERIAL	Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.																						
DEVID_ENCAP	Device ID of another device. This is for layered device driver usage.																						
DEVID_FAB	Fabricated device ID.																						
<i>minor_name</i>	The minor name to be encoded.																						

**INTERFACE  
LEVEL  
DESCRIPTION**

*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()`.

Solaris DDI specific (Solaris DDI).

The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device's physical connection or its logical device name or number.

`ddi_devid_compare()` compares two device IDs byte-by-byte and determines both equality and sort order.

`ddi_devid_sizeof()` returns the number of bytes allocated for the passed in device ID (*devid*).

`ddi_devid_init()` allocates memory and initializes the opaque device ID structure. This function does not store the *devid*. If the device id is not derived from the device's firmware, it is the driver's responsibility to store the *devid* on some reliable store. When a *devid\_type* of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (*id*) must be passed in (*nbytes*).

When the *devid\_type* `DEVID_FAB` is used, the array of bytes (*id*) must be NULL and the length (*nbytes*) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine's host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

`ddi_devid_free()` frees the memory allocated for the returned *devid* by the `ddi_devid_init()` and `devid_str_decode()` functions.

`ddi_devid_register()` registers the device ID address (*devid*) with the DDI framework, associating it with the `dev_info` passed in (*dip*). The drivers must register device IDs at attach time. See `attach(9E)`.

`ddi_devid_unregister()` removes the device ID address from the `dev_info` passed in (*dip*). Drivers must use this function to unregister the device ID when devices are being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_devid_free()` function. See `detach(9E)`.

`ddi_devid_valid()` validates the device ID (*devid*) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

## ddi\_devid\_compare(9F)

The `ddi_devid_str_encode()` function encodes a *devid* and *minor\_name* into a null-terminated ASCII string, returning a pointer to that string. If both a *devid* and a *minor\_name* are non-null, then a slash (/) is used to separate the *devid* from the *minor\_name* in the encoded string. If *minor\_name* is null, then only the *devid* is encoded. If the *devid* is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine *devid* equality. The returned string must be freed by calling `devid_str_free()`.

The `ddi_devid_str_decode()` function takes a string previously produced by the `devid_str_encode(3DEVID)` or `ddi_devid_str_encode()` function and decodes the contained device ID and *minor\_name*, allocating and returning pointers to the extracted parts through the *retdevid* and *retminor\_name* arguments. If the special *devidstr* `id0` was specified then the returned device ID and minor name will both be null. A non-null returned *devid* must be freed by the caller through the `ddi_devid_free()` function. A non-null returned minor name must be freed by calling `ddi_devid_str_free()`.

The `ddi_devid_str_free()` function is used to free all strings returned by the `ddi_devid` functions (the `ddi_devid_str_encode()` function return value and the returned *retminor\_name* argument).

### RETURN VALUES

`ddi_devid_init()` returns the following values:

- DDI\_SUCCESS      Success.
- DDI\_FAILURE      Out of memory. An invalid *devid\_type* was passed in.

`ddi_devid_valid()` returns the following values:

- DDI\_SUCCESS      Valid device ID.
- DDI\_FAILURE      Invalid device ID.

`ddi_devid_register()` returns the following values:

- DDI\_SUCCESS      Success.
- DDI\_FAILURE      Failure. The device ID is already registered or the device ID is invalid.

`ddi_devid_valid()` returns the following values:

- DDI\_SUCCESS      Valid device ID.
- DDI\_FAILURE      Invalid device ID.

`ddi_devid_compare()` returns the following values:

- 1              The first device ID is less than the second device ID.
- 0                The first device ID is equal to the second device ID.
- 1                The first device ID is greater than the second device ID.

ddi\_devid\_compare(9F)

`ddi_devid_sizeof()` returns the size of the *devid* in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.

`ddi_devid_str_encode()` returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid *devid*. If the return value is non-null then the caller must free the returned string by using the `devid_str_free()` function.

`ddi_devid_str_decode()` returns the following values:

DDI\_SUCCESS

Success.

DDI\_FAILURE

Failure; the *devidstr* string was not valid.

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

**SEE ALSO** `devid_get(3DEVID)`, `libdevid(3LIB)`, `attributes(5)`, `attach(9E)`, `detach(9E)`, `kmem_free(9F)`

*Writing Device Drivers*

## ddi\_dev\_is\_needed(9F)

<b>NAME</b>	ddi_dev_is_needed – inform the system that a device’s component is required
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dev_is_needed(dev_info_t *dip, int component, int level);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)  <i>dip</i> Pointer to the device’s dev_info structure. <i>component</i> Component of the driver which is needed. <i>level</i> Power level at which the component is needed.
<b>DESCRIPTION</b>	<p>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).</p> <p>The ddi_dev_is_needed() function informs the system that a device component is needed at the specified power level. The <i>level</i> argument must be non-zero.</p> <p>This function sets a <i>component</i> to the required level and sets all devices which depend on this to their normal power levels. If <i>component</i> 0 of a device using original Power Management interfaces (calls pm_create_components(9F)) is at power level 0, the ddi_dev_is_needed() call will result in component 0 being returned to normal power and the device being resumed via attach(9E) before ddi_dev_is_needed() returns.</p> <p>The state of the device should be examined before each physical access. The ddi_dev_is_needed() function should be called to set a <i>component</i> 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.</p> <p>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().</p>
<b>RETURN VALUES</b>	The ddi_dev_is_needed() function returns:  DDI_SUCCESS        Power successfully set to the requested level. DDI_FAILURE        An error occurred.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> disk driver code</p> <p>A hypothetical disk driver might include this code:</p> <pre>static int xxdisk_spun_down(struct xxstate *xsp) {     return (xsp-&gt;power_level[DISK_COMPONENT] &lt; POWER_SPUN_UP); } static int xxdisk_strategy(struct buf *bp)</pre>

**EXAMPLE 1** disk driver code (Continued)

```

{
. . .

mutex_enter(&xxstate_lock);
/*
 * Since we have to drop the mutex, we have to do this in a loop
 * in case we get preempted and the device gets taken away from
 * us again
 */
while (device_spun_down(sp)) {
    mutex_exit(&xxstate_lock);
    if (ddi_dev_is_needed(xsp->mydip,
        XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
        bioerror(bp, EIO);
        biodone(bp);
        return (0);
    }
    mutex_enter(&xxstate_lock);
}
xsp->device_busy++;
mutex_exit(&xxstate_lock);

. . .
}

```

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

**ATTRIBUTES** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Obsolete

**SEE ALSO** [pm\(7D\)](#), [pm-components\(9P\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [power\(9E\)](#), [pm\\_busy\\_components\(9F\)](#), [pm\\_create\\_components\(9F\)](#), [pm\\_destroy\\_components\(9F\)](#), [pm\\_idle\\_component\(9F\)](#)

*Writing Device Drivers*

## ddi\_dev\_is\_sid(9F)

<b>NAME</b>	ddi_dev_is_sid – tell whether a device is self-identifying
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dev_is_sid(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> A pointer to the device's dev_info structure.
<b>DESCRIPTION</b>	ddi_dev_is_sid() tells the caller whether the device described by <i>dip</i> 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).
<b>RETURN VALUES</b>	DDI_SUCCESS     Device is self-identifying. DDI_FAILURE     Device is not self-identifying.
<b>CONTEXT</b>	ddi_dev_is_sid() can be called from user or interrupt context.
<b>EXAMPLES</b>	<pre>1  ... 2  int 3  bz_probe(dev_info_t *dip) 4  { 5      ... 6      if (ddi_dev_is_sid(dip) == DDI_SUCCESS) { 7          /* 8           * This is the self-identifying version (OpenBoot). 9           * No need to probe for it because we know it is there. 10         * The existence of dip &amp;&amp; ddi_dev_is_sid() proves this. 11         */ 12         return (DDI_PROBE_DONTCARE); 13     } 14     /* 15     * Not a self-identifying variant of the device. Now we have to 16     * do some work to see whether it is really attached to the 17     * system. 18     */ 19     ...</pre>
<b>SEE ALSO</b>	probe(9E) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_dev_nintrs – return the number of interrupt specifications a device has
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dev_nintrs</b>(dev_info_t *dip, int *resultp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_dev_nintrs() returns the number of interrupt specifications a device has in *resultp.
<b>RETURN VALUES</b>	<p>ddi_dev_nintrs() returns:</p> <p>DDI_SUCCESS      A successful return. The number of interrupt specifications that the device has is set in <i>resultp</i>.</p> <p>DDI_FAILURE      The device has no interrupt specifications.</p>
<b>CONTEXT</b>	ddi_dev_nintrs() can be called from user or interrupt context.
<b>SEE ALSO</b>	isa(4), sbus(4), ddi_add_intr(9F), ddi_dev_nregs(9F), ddi_dev_regsize(9F) <i>Writing Device Drivers</i>

## ddi\_dev\_nregs(9F)

<b>NAME</b>	ddi_dev_nregs – return the number of register sets a device has				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dev_nregs(dev_info_t *dip, int *resultp);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).				
<b>PARAMETERS</b>	<table><tr><td><i>dip</i></td><td>A pointer to the device's dev_info structure.</td></tr><tr><td><i>resultp</i></td><td>Pointer to an integer that holds the number of register sets on return.</td></tr></table>	<i>dip</i>	A pointer to the device's dev_info structure.	<i>resultp</i>	Pointer to an integer that holds the number of register sets on return.
<i>dip</i>	A pointer to the device's dev_info structure.				
<i>resultp</i>	Pointer to an integer that holds the number of register sets on return.				
<b>DESCRIPTION</b>	The function ddi_dev_nregs() returns the number of sets of registers the device has.				
<b>RETURN VALUES</b>	ddi_dev_nregs() returns: <table><tr><td>DDI_SUCCESS</td><td>A successful return. The number of register sets is returned in <i>resultp</i>.</td></tr><tr><td>DDI_FAILURE</td><td>The device has no registers.</td></tr></table>	DDI_SUCCESS	A successful return. The number of register sets is returned in <i>resultp</i> .	DDI_FAILURE	The device has no registers.
DDI_SUCCESS	A successful return. The number of register sets is returned in <i>resultp</i> .				
DDI_FAILURE	The device has no registers.				
<b>CONTEXT</b>	ddi_dev_nregs() can be called from user or interrupt context.				
<b>SEE ALSO</b>	ddi_dev_nintrs(9F), ddi_dev_regsize(9F) <i>Writing Device Drivers</i>				

<b>NAME</b>	ddi_dev_resize – return the size of a device’s register						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dev_resize</b>(dev_info_t *dip, uint_t rnumber, off_t *resultp);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>PARAMETERS</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>dip</i></td> <td>A pointer to the device’s dev_info structure.</td> </tr> <tr> <td style="padding-right: 20px;"><i>rnumber</i></td> <td>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).</td> </tr> <tr> <td style="padding-right: 20px;"><i>resultp</i></td> <td>Pointer to an integer that holds the size, in bytes, of the described register (if it exists).</td> </tr> </table>	<i>dip</i>	A pointer to the device’s dev_info structure.	<i>rnumber</i>	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).	<i>resultp</i>	Pointer to an integer that holds the size, in bytes, of the described register (if it exists).
<i>dip</i>	A pointer to the device’s dev_info structure.						
<i>rnumber</i>	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).						
<i>resultp</i>	Pointer to an integer that holds the size, in bytes, of the described register (if it exists).						
<b>DESCRIPTION</b>	ddi_dev_resize() returns the size, in bytes, of the device register specified by <i>dip</i> and <i>rnumber</i> . This is useful when, for example, one of the registers is a frame buffer with a varying size known only to its proms.						
<b>RETURN VALUES</b>	<p>ddi_dev_resize() returns:</p> <table border="0"> <tr> <td style="padding-right: 20px;">DDI_SUCCESS</td> <td>A successful return. The size, in bytes, of the specified register, is set in <i>resultp</i>.</td> </tr> <tr> <td style="padding-right: 20px;">DDI_FAILURE</td> <td>An invalid (nonexistent) register number was specified.</td> </tr> </table>	DDI_SUCCESS	A successful return. The size, in bytes, of the specified register, is set in <i>resultp</i> .	DDI_FAILURE	An invalid (nonexistent) register number was specified.		
DDI_SUCCESS	A successful return. The size, in bytes, of the specified register, is set in <i>resultp</i> .						
DDI_FAILURE	An invalid (nonexistent) register number was specified.						
<b>CONTEXT</b>	ddi_dev_resize() can be called from user or interrupt context.						
<b>SEE ALSO</b>	ddi_dev_nintrs(9F), ddi_dev_nregs(9F)						
	<i>Writing Device Drivers</i>						

## ddi\_dev\_report\_fault(9F)

<b>NAME</b>	ddi_dev_report_fault – Report a hardware failure								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_dev_report_fault (dev_info_t *dip, ddi_fault_impact_t     impact, ddi_fault_location_t location, const char *message );</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)  <table><tr><td><i>dip</i></td><td>Pointer to the driver's <code>dev_info</code> structure to which the fault report relates. (Normally the caller's own <code>dev_info</code> pointer).</td></tr><tr><td><i>impact</i></td><td>One of a set of enumerated values indicating the impact of the fault on the device's ability to provide normal service.</td></tr><tr><td><i>location</i></td><td>One of a set of enumerated values indicating the location of the fault, relative to the hardware controlled by the driver specified by <code>dip</code>.</td></tr><tr><td><i>message</i></td><td>Text of the message describing the fault being reported.</td></tr></table>	<i>dip</i>	Pointer to the driver's <code>dev_info</code> structure to which the fault report relates. (Normally the caller's own <code>dev_info</code> pointer).	<i>impact</i>	One of a set of enumerated values indicating the impact of the fault on the device's ability to provide normal service.	<i>location</i>	One of a set of enumerated values indicating the location of the fault, relative to the hardware controlled by the driver specified by <code>dip</code> .	<i>message</i>	Text of the message describing the fault being reported.
<i>dip</i>	Pointer to the driver's <code>dev_info</code> structure to which the fault report relates. (Normally the caller's own <code>dev_info</code> pointer).								
<i>impact</i>	One of a set of enumerated values indicating the impact of the fault on the device's ability to provide normal service.								
<i>location</i>	One of a set of enumerated values indicating the location of the fault, relative to the hardware controlled by the driver specified by <code>dip</code> .								
<i>message</i>	Text of the message describing the fault being reported.								
<b>DESCRIPTION</b>	<p>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.</p> <p>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 <code>impact</code> parameter:</p> <p><code>DDI_SERVICE_LOST</code> Indicates a total loss of service. The driver is unable to implement the normal functions of its hardware.</p> <p><code>DDI_SERVICE_DEGRADED</code> 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.</p> <p><code>DDI_SERVICE_UNAFFECTED</code> 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.</p> <p><code>DDI_SERVICE_RESTORED</code> 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.</p> <p>The <code>location</code> parameter should be one of the following values:</p>								

**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(9F)

```
        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)`

<b>NAME</b>	ddi_dma_addr_bind_handle – binds an address to a DMA handle																		
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>																		
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p> <p><i>as</i>                 A pointer to an address space structure. This parameter should be set to NULL, which implies kernel address space.</p> <p><i>addr</i>               Virtual address of the memory object.</p> <p><i>len</i>                Length of the memory object in bytes.</p> <p><i>flags</i>              Valid flags include:</p> <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_WRITE</td> <td>Transfer direction is from memory to I/O.</td> </tr> <tr> <td>DDI_DMA_READ</td> <td>Transfer direction is from I/O to memory.</td> </tr> <tr> <td>DDI_DMA_RDWR</td> <td>Both read and write.</td> </tr> <tr> <td>DDI_DMA_REDZONE</td> <td>Establish an MMU redzone at end of the object.</td> </tr> <tr> <td>DDI_DMA_PARTIAL</td> <td>Partial resource allocation.</td> </tr> <tr> <td>DDI_DMA_CONSISTENT</td> <td>Nonsequential, random, and small block transfers.</td> </tr> <tr> <td>DDI_DMA_STREAMING</td> <td>Sequential, unidirectional, block-sized, and block-aligned transfers.</td> </tr> </table> <p><i>callback</i>           The address of a function to call back later if resources are not currently available. The following special function addresses may also be used.</p> <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table> <p><i>arg</i>                Argument to be passed to the callback function, <i>callback</i>, if such a function is specified.</p>	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.	DDI_DMA_SLEEP	Wait until resources are available.	DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.
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.																		
DDI_DMA_SLEEP	Wait until resources are available.																		
DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.																		

## ddi\_dma\_addr\_bind\_handle(9F)

*cookiep* A pointer to the first `ddi_dma_cookie(9S)` structure.

*ccountp* Upon a successful return, *ccountp* points to a value representing the number of cookies for this DMA object.

### DESCRIPTION

`ddi_dma_addr_bind_handle()` allocates DMA resources for a memory object such that a device can perform DMA to or from the object. DMA resources are allocated considering the device's DMA attributes as expressed by `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`).

`ddi_dma_addr_bind_handle()` fills in the first DMA cookie pointed to by *cookiep* with the appropriate address, length, and bus type. *\*ccountp* is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved by calling `ddi_dma_nextcookie(9F)` the number of times specified by *\*ccountp*-1.

When a DMA transfer completes, the driver frees up system DMA resources by calling `ddi_dma_unbind_handle(9F)`.

The *flags* argument contains information for mapping routines.

`DDI_DMA_WRITE`, `DDI_DMA_READ`, `DDI_DMA_RDWR`

These flags describe the intended direction of the DMA transfer.

`DDI_DMA_STREAMING`

This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the `minxfer` and `burstsizes` fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`) is used to allocate the most effective hardware support for large transfers.

`DDI_DMA_CONSISTENT`

This flag should be set if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using `DDI_DMA_CONSISTENT`.

`DDI_DMA_REDZONE`

If this flag is set, the system attempts to establish a protected red zone after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support a red zone.

`DDI_DMA_PARTIAL`

Setting this flag indicates the caller can accept resources for part of the object. That is, if the size of the object exceeds the resources available, only resources for a portion of the object are allocated. The system indicates this condition by returning status `DDI_DMA_PARTIAL_MAP`. At a later point, the caller can use `ddi_dma_getwin(9F)` to change the valid portion of the object for which resources are allocated. If resources were allocated for only part of the object, `ddi_dma_addr_bind_handle()` returns resources for the first DMA window.

## ddi\_dma\_addr\_bind\_handle(9F)

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 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:

<code>DDI_DMA_MAPPED</code>	Successfully allocated resources for the entire object.
<code>DDI_DMA_PARTIAL_MAP</code>	Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the <code>DDI_DMA_PARTIAL</code> flag in <i>flags</i> .
<code>DDI_DMA_INUSE</code>	Another I/O transaction is using the DMA handle.
<code>DDI_DMA_NORESOURCES</code>	No resources are available at the present time.
<code>DDI_DMA_NOMAPPING</code>	The object cannot be reached by the device requesting the resources.
<code>DDI_DMA_TOOBIG</code>	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*

## ddi\_dma\_addr\_bind\_handle(9F)

**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)`.

ddi\_dma\_addr\_setup(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 PARAMETERS**

This interface is obsolete. ddi\_dma\_addr\_bind\_handle(9F) should be used instead.

*dip* A pointer to the device's dev\_info structure.

*as* A pointer to an address space structure. Should be set to NULL, which implies kernel address space.

*addr* Virtual address of the memory object.

*len* Length of the memory object in bytes.

*flags* Flags that would go into the ddi\_dma\_req structure (see ddi\_dma\_req(9S)).

*waitfp* The address of a function to call back later if resources aren't available now. The special function addresses DDI\_DMA\_SLEEP and DDI\_DMA\_DONTWAIT (see ddi\_dma\_req(9S)) are taken to mean, respectively, wait until resources are available or, do not wait at all and do not schedule a callback.

*arg* Argument to be passed to a callback function, if such a function is specified.

*lim* A pointer to a DMA limits structure for this device (see ddi\_dma\_lim\_sparc(9S) or ddi\_dma\_lim\_x86(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

*handlep* Pointer to a DMA handle. See ddi\_dma\_setup(9F) for a discussion of handle.

**DESCRIPTION** ddi\_dma\_addr\_setup() is an interface to ddi\_dma\_setup(9F). It uses its arguments to construct an appropriate ddi\_dma\_req structure and calls ddi\_dma\_setup(9F) with it.

**RETURN VALUES** See ddi\_dma\_setup(9F) for the possible return values for this function.

**CONTEXT** ddi\_dma\_addr\_setup() can be called from user or interrupt context, except when *waitfp* is set to DDI\_DMA\_SLEEP, in which case it can be called from user context only.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

ddi\_dma\_addr\_setup(9F)

**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*

<b>NAME</b>	ddi_dma_alloc_handle – allocate DMA handle				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i> Pointer to the device's dev_info structure.</p> <p><i>attr</i> Pointer to a DMA attribute structure for this device (see ddi_dma_attr(9S)).</p> <p><i>callback</i> The address of a function to call back later if resources aren't available now. The following special function addresses may also be used.</p> <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 20px;">DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table> <p><i>arg</i> Argument to be passed to a callback function, if such a function is specified.</p> <p><i>handlep</i> Pointer to the DMA handle to be initialized.</p>	DDI_DMA_SLEEP	Wait until resources are available.	DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.
DDI_DMA_SLEEP	Wait until resources are available.				
DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.				
<b>DESCRIPTION</b>	<p>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 <i>dip</i> 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 <i>handlep</i>. 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.</p> <p>The callback function, <i>callback</i>, indicates how a caller wants to handle the possibility of resources not being available. If <i>callback</i> is set to DDI_DMA_DONTWAIT, then the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If <i>callback</i> is set to DDI_DMA_SLEEP, then the caller wishes to have the the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be a function to call at a later time when resources may become available. When the specified function is called, it is passed <i>arg</i> 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.</p>				

## ddi\_dma\_alloc\_handle(9F)

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 <code>ddi_dma_attr(9S)</code> 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*

<b>NAME</b>	ddi_dma_buf_bind_handle – binds a system buffer to a DMA handle																		
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>																		
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p> <p><i>bp</i>                 A pointer to a system buffer structure (see buf(9S)).</p> <p><i>flags</i>             Valid flags include:</p> <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_WRITE</td> <td>Transfer direction is from memory to I/O</td> </tr> <tr> <td>DDI_DMA_READ</td> <td>Transfer direction is from I/O to memory</td> </tr> <tr> <td>DDI_DMA_RDWR</td> <td>Both read and write</td> </tr> <tr> <td>DDI_DMA_REDZONE</td> <td>Establish an MMU redzone at end of the object.</td> </tr> <tr> <td>DDI_DMA_PARTIAL</td> <td>Partial resource allocation</td> </tr> <tr> <td>DDI_DMA_CONSISTENT</td> <td>Nonsequential, random, and small block transfers.</td> </tr> <tr> <td>DDI_DMA_STREAMING</td> <td>Sequential, unidirectional, block-sized, and block-aligned transfers.</td> </tr> </table> <p><i>callback</i>         The address of a function to call back later if resources are not available now. The following special function addresses may also be used.</p> <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table> <p><i>arg</i>                Argument to be passed to the callback function, <i>callback</i>, if such a function is specified.</p> <p><i>cookiep</i>           A pointer to the first ddi_dma_cookie(9S) structure.</p> <p><i>ccountp</i>           Upon a successful return, <i>ccountp</i> points to a value representing the number of cookies for this DMA object.</p>	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.	DDI_DMA_SLEEP	Wait until resources are available.	DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.
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.																		
DDI_DMA_SLEEP	Wait until resources are available.																		
DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.																		

## ddi\_dma\_buf\_bind\_handle(9F)

### DESCRIPTION

`ddi_dma_buf_bind_handle()` allocates DMA resources for a system buffer such that a device can perform DMA to or from the buffer. DMA resources are allocated considering the device's DMA attributes as expressed by `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`).

`ddi_dma_buf_bind_handle()` fills in the first DMA cookie pointed to by *cookiep* with the appropriate address, length, and bus type. *\*countp* is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved by calling `ddi_dma_nextcookie(9F)` *\*countp-1* times.

When a DMA transfer completes, the driver should free up system DMA resources by calling `ddi_dma_unbind_handle(9F)`.

The *flags* argument contains information for mapping routines.

DDI\_DMA\_WRITE, DDI\_DMA\_READ, DDI\_DMA\_RDWR

These flags describe the intended direction of the DMA transfer.

DDI\_DMA\_STREAMING

This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the *minxfer* and *burstsizes* fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`) is used to allocate the most effective hardware support for large transfers.

DDI\_DMA\_CONSISTENT

This flag should be set if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using DDI\_DMA\_CONSISTENT.

DDI\_DMA\_REDZONE

If this flag is set, the system attempts to establish a protected red zone after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support a red zone.

DDI\_DMA\_PARTIAL

Setting this flag indicates the caller can accept resources for part of the object. That is, if the size of the object exceeds the resources available, only resources for a portion of the object are allocated. The system indicates this condition returning status DDI\_DMA\_PARTIAL\_MAP. At a later point, the caller can use `ddi_dma_getwin(9F)` to change the valid portion of the object for which resources are allocated. If resources were allocated for only part of the object, `ddi_dma_addr_bind_handle()` returns resources for the first DMA window. Even when DDI\_DMA\_PARTIAL is set, the system may decide to allocate resources for the entire object (less overhead) in which case DDI\_DMA\_MAPPED is returned.

The callback function, *callback*, indicates how a caller wants to handle the possibility of resources not being available. If *callback* is set to DDI\_DMA\_DONTWAIT, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If

## ddi\_dma\_buf\_bind\_handle(9F)

*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 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:

<code>DDI_DMA_MAPPED</code>	Successfully allocated resources for the entire object.
<code>DDI_DMA_PARTIAL_MAP</code>	Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the <code>DDI_DMA_PARTIAL</code> flag in <i>flags</i> .
<code>DDI_DMA_INUSE</code>	Another I/O transaction is using the DMA handle.
<code>DDI_DMA_NORESOURCES</code>	No resources are available at the present time.
<code>DDI_DMA_NOMAPPING</code>	The object cannot be reached by the device requesting the resources.
<code>DDI_DMA_TOOBIG</code>	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*

## ddi\_dma\_buf\_bind\_handle(9F)

**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)`.

ddi\_dma\_buf\_setup(9F)

**NAME** ddi\_dma\_buf\_setup – easier DMA setup for use with buffer structures

**SYNOPSIS**

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

int ddi_dma_buf_setup(dev_info_t *dip, struct buf *bp, uint_t flags,
    int (*waitfp) (caddr_t), caddr_t arg, ddi_dma_lim_t *lim,
    ddi_dma_handle_t *handlep);
```

**INTERFACE LEVEL PARAMETERS**

This interface is obsolete. ddi\_dma\_buf\_bind\_handle(9F) should be used instead.

*dip* A pointer to the device's dev\_info structure.

*bp* A pointer to a system buffer structure (see buf(9S)).

*flags* Flags that go into a ddi\_dma\_req structure (see ddi\_dma\_req(9S)).

*waitfp* The address of a function to call back later if resources aren't available now. The special function addresses DDI\_DMA\_SLEEP and DDI\_DMA\_DONTWAIT (see ddi\_dma\_req(9S)) are taken to mean, respectively, wait until resources are available, or do not wait at all and do not schedule a callback.

*arg* Argument to be passed to a callback function, if such a function is specified.

*lim* A pointer to a DMA limits structure for this device (see ddi\_dma\_lim\_sparc(9S) or ddi\_dma\_lim\_x86(9S)). If this pointer is NULL, a default set of DMA limits is assumed.

*handlep* Pointer to a DMA handle. See ddi\_dma\_setup(9F) for a discussion of handle.

**DESCRIPTION** ddi\_dma\_buf\_setup() is an interface to ddi\_dma\_setup(9F). It uses its arguments to construct an appropriate ddi\_dma\_req structure and calls ddi\_dma\_setup() with it.

**RETURN VALUES** See ddi\_dma\_setup(9F) for the possible return values for this function.

**CONTEXT** ddi\_dma\_buf\_setup() can be called from user or interrupt context, except when *waitfp* is set to DDI\_DMA\_SLEEP, in which case it can be called from user context only.

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

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)

ddi\_dma\_buf\_setup(9F)

*Writing Device Drivers*

<b>NAME</b>	ddi_dma_burstsizes – find out the allowed burst sizes for a DMA mapping
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dma_burstsizes</b>(ddi_dma_handle_t <i>handle</i>);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>handle</i>                    A DMA handle that was filled in by a successful call to <code>ddi_dma_setup(9F)</code>.</p>
<b>DESCRIPTION</b>	<p><code>ddi_dma_burstsizes()</code> returns the allowed burst sizes for a DMA mapping. This value is derived from the <code>dlim_burstsizes</code> member of the <code>ddi_dma_lim_sparc(9S)</code> structure, but it shows the allowable burstsizes <i>after</i> imposing on it the limitations of other device layers in addition to device's own limitations.</p>
<b>RETURN VALUES</b>	<p><code>ddi_dma_burstsizes()</code> returns a binary encoded value of the allowable DMA burst sizes. See <code>ddi_dma_lim_sparc(9S)</code> for a discussion of DMA burst sizes.</p>
<b>CONTEXT</b>	This function can be called from user or interrupt context.
<b>SEE ALSO</b>	<p><code>ddi_dma_devalign(9F)</code>, <code>ddi_dma_setup(9F)</code>, <code>ddi_dma_lim_sparc(9S)</code>, <code>ddi_dma_req(9S)</code></p> <p><i>Writing Device Drivers</i></p>

## ddi\_dma\_coff(9F)

<b>NAME</b>	ddi_dma_coff – convert a DMA cookie to an offset within a DMA handle
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_coff(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep,                 off_t *offp);</pre>
<b>INTERFACE LEVEL</b>	Solaris SPARC DDI (Solaris SPARC DDI).
<b>PARAMETERS</b>	<p><i>handle</i>            The <i>handle</i> filled in by a call to ddi_dma_setup(9F).</p> <p><i>cookiep</i>           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.</p> <p><i>offp</i>               A pointer to an offset to be filled in.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_coff() converts the values in DMA cookie pointed to by <i>cookiep</i> to an offset (in bytes) from the beginning of the object that the DMA <i>handle</i> has mapped.</p> <p>ddi_dma_coff() allows a driver to update a DMA cookie with values it reads from its device's DMA engine after a transfer completes and convert that value into an offset into the object that is mapped for DMA.</p>
<b>RETURN VALUES</b>	<p>ddi_dma_coff() returns:</p> <p>DDI_SUCCESS        Successfully filled in <i>offp</i>.</p> <p>DDI_FAILURE        Failed to successfully fill in <i>offp</i>.</p>
<b>CONTEXT</b>	ddi_dma_coff() can be called from user or interrupt context.
<b>SEE ALSO</b>	ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S) <i>Writing Device Drivers</i>

**NAME** ddi\_dma\_curwin – report current DMA window offset and size

**SYNOPSIS**

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

int ddi_dma_curwin(ddi_dma_handle_t handle, off_t *offp, uint_t
    *lenp);
```

**INTERFACE LEVEL** This interface is obsolete. ddi\_dma\_getwin(9F) should be used instead.

**PARAMETERS**

*handle* The DMA handle filled in by a call to ddi\_dma\_setup(9F).

*offp* A pointer to a value which will be filled in with the current offset from the beginning of the object that is mapped for DMA.

*lenp* A pointer to a value which will be filled in with the size, in bytes, of the current window onto the object that is mapped for DMA.

**DESCRIPTION** ddi\_dma\_curwin() reports the current DMA window offset and size. If a DMA mapping allows partial mapping, that is if the DDI\_DMA\_PARTIAL flag in the ddi\_dma\_req(9S) structure is set, its current (effective) DMA window offset and size can be obtained by a call to ddi\_dma\_curwin().

**RETURN VALUES** ddi\_dma\_curwin() returns:

DDI\_SUCCESS The current length and offset can be established.

DDI\_FAILURE Otherwise.

**CONTEXT** ddi\_dma\_curwin() can be called from user or interrupt context.

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

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*

## ddi\_dma\_devalign(9F)

<b>NAME</b>	ddi_dma_devalign – find DMA mapping alignment and minimum transfer size						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dma_devalign</b>(ddi_dma_handle_t <i>handle</i>, uint_t *<i>alignment</i>, uint_t *<i>minxfr</i>);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>PARAMETERS</b>	<table><tr><td><i>handle</i></td><td>The DMA handle filled in by a successful call to ddi_dma_setup(9F).</td></tr><tr><td><i>alignment</i></td><td>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.</td></tr><tr><td><i>minxfr</i></td><td>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.</td></tr></table>	<i>handle</i>	The DMA handle filled in by a successful call to ddi_dma_setup(9F).	<i>alignment</i>	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.	<i>minxfr</i>	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.
<i>handle</i>	The DMA handle filled in by a successful call to ddi_dma_setup(9F).						
<i>alignment</i>	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.						
<i>minxfr</i>	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.						
<b>DESCRIPTION</b>	ddi_dma_devalign() determines after a successful DMA mapping (see ddi_dma_setup(9F)) the minimum required data alignment and minimum DMA transfer size.						
<b>RETURN VALUES</b>	ddi_dma_devalign() returns: DDI_SUCCESS    The <i>alignment</i> and <i>minxfr</i> values have been filled. DDI_FAILURE    The handle was illegal.						
<b>CONTEXT</b>	ddi_dma_devalign() can be called from user or interrupt context.						
<b>SEE ALSO</b>	ddi_dma_setup(9F), ddi_iomin(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)  <i>Writing Device Drivers</i>						

<b>NAME</b>	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																				
<b>SYNOPSIS</b>	<pre>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 *dmaereq, 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);</pre>																				
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI). The ddi_dmae_getlim() interface, described below, is obsolete. Use ddi_dmae_getattr(), also described below, to replace it.																				
<b>PARAMETERS</b>	<table border="0"> <tr> <td style="vertical-align: top;"><i>dip</i></td> <td>A dev_info pointer that identifies the device.</td> </tr> <tr> <td style="vertical-align: top;"><i>chnl</i></td> <td>A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.</td> </tr> <tr> <td style="vertical-align: top;"><i>callback</i></td> <td>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used: <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table> </td> </tr> <tr> <td style="vertical-align: top;"><i>arg</i></td> <td>Argument to be passed to the callback function, if specified.</td> </tr> <tr> <td style="vertical-align: top;"><i>dmaereq</i></td> <td>A pointer to a DMA engine request structure. See ddi_dmae_req(9S).</td> </tr> <tr> <td style="vertical-align: top;"><i>cookiep</i></td> <td>A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.</td> </tr> <tr> <td style="vertical-align: top;"><i>countp</i></td> <td>A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.</td> </tr> <tr> <td style="vertical-align: top;"><i>limitsp</i></td> <td>A pointer to a DMA limit structure. See ddi_dma_lim_x86(9S).</td> </tr> </table>	<i>dip</i>	A dev_info pointer that identifies the device.	<i>chnl</i>	A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.	<i>callback</i>	The address of a function to call back later if resources are not currently available. The following special function addresses may also be used: <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table>	DDI_DMA_SLEEP	Wait until resources are available.	DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.	<i>arg</i>	Argument to be passed to the callback function, if specified.	<i>dmaereq</i>	A pointer to a DMA engine request structure. See ddi_dmae_req(9S).	<i>cookiep</i>	A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.	<i>countp</i>	A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.	<i>limitsp</i>	A pointer to a DMA limit structure. See ddi_dma_lim_x86(9S).
<i>dip</i>	A dev_info pointer that identifies the device.																				
<i>chnl</i>	A DMA channel number. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.																				
<i>callback</i>	The address of a function to call back later if resources are not currently available. The following special function addresses may also be used: <table border="0" style="margin-left: 2em;"> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table>	DDI_DMA_SLEEP	Wait until resources are available.	DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.																
DDI_DMA_SLEEP	Wait until resources are available.																				
DDI_DMA_DONTWAIT	Do not wait until resources are available and do not schedule a callback.																				
<i>arg</i>	Argument to be passed to the callback function, if specified.																				
<i>dmaereq</i>	A pointer to a DMA engine request structure. See ddi_dmae_req(9S).																				
<i>cookiep</i>	A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.																				
<i>countp</i>	A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.																				
<i>limitsp</i>	A pointer to a DMA limit structure. See ddi_dma_lim_x86(9S).																				

ddi\_dmae(9F)

	<p><i>attrp</i>                      A pointer to a DMA attribute structure. See <code>ddi_dma_attr(9S)</code>.</p>
<b>DESCRIPTION</b>	<p>There are three possible ways that a device can perform DMA engine functions:</p> <p><b>Bus master DMA</b> 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 <code>ddi_dma_segtocookie(9F)</code>. See <code>ddi_dma_cookie(9S)</code> for a description of a DMA cookie.</p> <p><b>Third-party DMA</b> 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 <code>ddi_dmae_1stparty()</code>, 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.</p> <p><b>First-party DMA</b> 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 <code>ddi_dmae_1stparty()</code> function may be used to perform whatever configuration is necessary to enable this mode.</p>
<b>ddi_dmae_alloc()</b>	<p>The <code>ddi_dmae_alloc()</code> function is used to acquire a DMA channel of the system DMA engine. <code>ddi_dmae_alloc()</code> 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 <code>ddi_dmae_release()</code> after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See <code>detach(9E)</code>. No other driver may acquire the DMA channel until it is released.</p> <p>If the requested channel is not immediately available, the value of <i>callback</i> determines what action will be taken. If the value of <i>callback</i> is <code>DDI_DMA_DONTWAIT</code>, <code>ddi_dmae_alloc()</code> will return immediately. The value <code>DDI_DMA_SLEEP</code> 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, <code>ddi_dmae_alloc()</code> returns immediately, and when resources might have become available, the callback function is called (with the argument <i>arg</i>) 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 <code>DDI_DMA_CALLBACK_DONE</code>. If it tries to allocate the channel but fails to do so, it must return the value <code>DDI_DMA_CALLBACK_RUNOUT</code>. In this case, the callback function is put back on a list to be called again later.</p>

- ddi\_dmae\_prog()** The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for *dmaereqp* if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through *dmaereqp*. The fields of that structure are documented in `ddi_dmae_req(9S)`.
- Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.
- ddi\_dmae\_disable()** The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device's DMA service requests.
- ddi\_dmae\_enable()** The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.
- ddi\_dmae\_stop()** The `ddi_dmae_stop()` function disables the channel and terminates any active operation.
- ddi\_dmae\_getcnt()** The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets *\*countp* to the number of bytes remaining to be transferred. The channel is assumed to be stopped.
- ddi\_dmae\_1stparty()** In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the system's DMA engine to operate in a "slave" ("cascade") mode.
- When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.
- ddi\_dmae\_getlim()** Note that this function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.
- The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by *limitsp*, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver

## ddi\_dmae(9F)

should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine's scatter/gather list to determine whether DMA chaining will be used. See `ddi_dma_lim_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 <code>ddi_dmae_alloc()</code> if the requested resources are not available and the value of <i>dmae_waitfp</i> is not <code>DDI_DMA_SLEEP</code> .

**CONTEXT** If `ddi_dmae_alloc()` is called from interrupt context, then its *dmae\_waitfp* argument and the callback function must not have the value `DDI_DMA_SLEEP`. Otherwise, all these routines may be called from user or interrupt context.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	x86

**SEE ALSO** `eisa(4)`, `isa(4)`, `attributes(5)`, `ddi_dma_buf_setup(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_nextseg(9F)`, `ddi_dma_nextwin(9F)`, `ddi_dma_segtocookie(9F)`, `ddi_dma_setup(9F)`, `ddi_dma_attr(9S)`, `ddi_dma_cookie(9S)`, `ddi_dma_lim_x86(9S)`, `ddi_dma_req(9S)`, `ddi_dmae_req(9S)`

**NAME** ddi\_dma\_free – release system DMA resources

**SYNOPSIS**

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

int ddi_dma_free(ddi_dma_handle_t handle);
```

**INTERFACE 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** ddi\_dma\_free() releases system DMA resources set up by ddi\_dma\_setup(9F). When a DMA transfer completes, the driver should free up system DMA resources established by a call to ddi\_dma\_setup(9F). This is done by a call to ddi\_dma\_free(). ddi\_dma\_free() does an implicit ddi\_dma\_sync(9F) for you so any further synchronization steps are not necessary.

**RETURN VALUES** ddi\_dma\_free() returns:

DDI\_SUCCESS Successfully released resources

DDI\_FAILURE Failed to free resources

**CONTEXT** ddi\_dma\_free() can be called from user or interrupt context.

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

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*

## ddi\_dma\_free\_handle(9F)

<b>NAME</b>	ddi_dma_free_handle – free DMA handle
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_dma_free_handle</b>(ddi_dma_handle_t *handle) ;</pre>
<b>PARAMETERS</b>	<i>handle</i> A pointer to the DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_dma_free_handle() destroys the DMA handle pointed to by <i>handle</i> . 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.
<b>CONTEXT</b>	ddi_dma_free_handle() can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	ddi_dma_alloc_handle(9F), ddi_dma_unbind_handle(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_dma_get_attr – get the device DMA attribute structure from a DMA handle
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dma_get_attr</b>(ddi_dma_handle_t <i>handle</i>, ddi_dma_attr_t     *<i>attrp</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI)</p> <p><i>handle</i> The handle filled in by a call to ddi_dma_alloc_handle(9F).</p> <p><i>attrp</i> Pointer to a buffer suitable for holding a DMA attribute structure. See ddi_dma_attr(9S).</p>
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	<p>DDI_SUCCESS Successfully passed back attribute structure in buffer pointed to by <i>attrp</i>.</p> <p>DDI_DMA_BADATTR A valid attribute structure could not be passed back.</p>
<b>CONTEXT</b>	ddi_dma_get_attr() can be called from any context.
<b>SEE ALSO</b>	ddi_dma_alloc_handle(9F), ddi_dma_attr(9S)

## ddi\_dma\_getwin(9F)

<b>NAME</b>	ddi_dma_getwin – activate a new DMA window												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <table><tr><td><i>handle</i></td><td>The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</td></tr><tr><td><i>win</i></td><td>Number of the window to activate.</td></tr><tr><td><i>offp</i></td><td>Pointer to an offset. Upon a successful return, <i>offp</i> will contain the new offset indicating the beginning of the window within the object.</td></tr><tr><td><i>lenp</i></td><td>Upon a successful return, <i>lenp</i> will contain the size, in bytes, of the current window.</td></tr><tr><td><i>cookiep</i></td><td>A pointer to the first ddi_dma_cookie(9S) structure.</td></tr><tr><td><i>ccountp</i></td><td>Upon a successful return, <i>ccountp</i> will contain the number of cookies for this DMA window.</td></tr></table>	<i>handle</i>	The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).	<i>win</i>	Number of the window to activate.	<i>offp</i>	Pointer to an offset. Upon a successful return, <i>offp</i> will contain the new offset indicating the beginning of the window within the object.	<i>lenp</i>	Upon a successful return, <i>lenp</i> will contain the size, in bytes, of the current window.	<i>cookiep</i>	A pointer to the first ddi_dma_cookie(9S) structure.	<i>ccountp</i>	Upon a successful return, <i>ccountp</i> will contain the number of cookies for this DMA window.
<i>handle</i>	The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).												
<i>win</i>	Number of the window to activate.												
<i>offp</i>	Pointer to an offset. Upon a successful return, <i>offp</i> will contain the new offset indicating the beginning of the window within the object.												
<i>lenp</i>	Upon a successful return, <i>lenp</i> will contain the size, in bytes, of the current window.												
<i>cookiep</i>	A pointer to the first ddi_dma_cookie(9S) structure.												
<i>ccountp</i>	Upon a successful return, <i>ccountp</i> will contain the number of cookies for this DMA window.												
<b>DESCRIPTION</b>	<p>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().</p> <p>The caller must first determine the number of DMA windows, <i>N</i>, using ddi_dma_numwin(9F). ddi_dma_getwin() takes a DMA window number from the range [0..<i>N</i>-1] as the parameter <i>win</i> and makes it the current DMA window.</p> <p>ddi_dma_getwin() fills in the first DMA cookie pointed to by <i>cookiep</i> with the appropriate address, length, and bus type. <i>*ccountp</i> is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved using ddi_dma_nextcookie(9F).</p> <p>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).</p> <p>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.</p>												
<b>RETURN VALUES</b>	ddi_dma_getwin() returns:												

ddi\_dma\_getwin(9F)

DDI\_SUCCESS Resources for the specified DMA window are allocated.

DDI\_FAILURE *win* is not a valid window index.

**CONTEXT** ddi\_dma\_getwin() can be called from user, kernel, or interrupt context.

**SEE ALSO** ddi\_dma\_addr\_bind\_handle(9F), ddi\_dma\_alloc\_handle(9F),  
ddi\_dma\_buf\_bind\_handle(9F), ddi\_dma\_nextcookie(9F),  
ddi\_dma\_numwin(9F), ddi\_dma\_sync(9F), ddi\_dma\_unbind\_handle(9F),  
ddi\_dma\_cookie(9S)

*Writing Device Drivers*

## ddi\_dma\_htoc(9F)

**NAME** ddi\_dma\_htoc – convert a DMA handle to a DMA address cookie

**SYNOPSIS**

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

int ddi_dma_htoc(ddi_dma_handle_t handle, off_t off,
                ddi_dma_cookie_t *cookiep);
```

**INTERFACE LEVEL** This interface is obsolete. ddi\_dma\_addr\_bind\_handle(9F) or ddi\_dma\_buf\_bind\_handle(9F) should be used instead.

**PARAMETERS**

*handle* The handle filled in by a call to ddi\_dma\_setup(9F).

*off* An offset into the object that *handle* maps.

*cookiep* A pointer to a ddi\_dma\_cookie(9S) structure.

**DESCRIPTION** ddi\_dma\_htoc() takes a DMA handle (established by ddi\_dma\_setup(9F)), and fills in the cookie pointed to by *cookiep* with the appropriate address, length, and bus type to be used to program the DMA engine.

**RETURN VALUES** ddi\_dma\_htoc() returns:

DDI\_SUCCESS Successfully filled in the cookie pointed to by *cookiep*.

DDI\_FAILURE Failed to successfully fill in the cookie.

**CONTEXT** ddi\_dma\_htoc() can be called from user or interrupt context.

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

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*

<b>NAME</b>	ddi_dma_mem_alloc – allocate memory for DMA transfer				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p> <p><i>length</i>            The length in bytes of the desired allocation.</p> <p><i>accattrp</i>          Pointer to a device access attribute structure of this device (see ddi_device_acc_attr(9S)).</p> <p><i>flags</i>             Data transfer mode flags. Possible values are:</p> <table border="0" style="margin-left: 2em;"> <tr> <td style="padding-right: 1em;">DDI_DMA_STREAMING</td> <td>Sequential, unidirectional, block-sized, and block-aligned transfers.</td> </tr> <tr> <td style="padding-right: 1em;">DDI_DMA_CONSISTENT</td> <td>Nonsequential transfers of small objects.</td> </tr> </table> <p><i>waitfp</i>            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, <i>arg</i> 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.</p>	DDI_DMA_STREAMING	Sequential, unidirectional, block-sized, and block-aligned transfers.	DDI_DMA_CONSISTENT	Nonsequential transfers of small objects.
DDI_DMA_STREAMING	Sequential, unidirectional, block-sized, and block-aligned transfers.				
DDI_DMA_CONSISTENT	Nonsequential transfers of small objects.				

## ddi\_dma\_mem\_alloc(9F)

	The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.
<i>arg</i>	Argument to be passed to the callback function, if such a function is specified.
<i>kaddrp</i>	On successful return, <i>kaddrp</i> points to the allocated memory.
<i>real_length</i>	The amount of memory, in bytes, allocated. Alignment and padding requirements may require <code>ddi_dma_mem_alloc()</code> to allocate more memory than requested in <i>length</i> .
<i>handlep</i>	Pointer to a data access handle.

### DESCRIPTION

`ddi_dma_mem_alloc()` allocates memory for DMA transfers to or from a device. The allocation will obey the alignment, padding constraints and device granularity as specified by the DMA attributes (see `ddi_dma_attr(9S)`) passed to `ddi_dma_alloc_handle(9F)` and the more restrictive attributes imposed by the system.

*flags* should be set to `DDI_DMA_STREAMING` if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the `minxfer` and `burstsizes` fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`) will be used to allocate the most effective hardware support for large transfers. For example, if an I/O transfer can be sped up by using an I/O cache, which has a minimum transfer of one cache line, `ddi_dma_mem_alloc()` will align the memory at a cache line boundary and it will round up *real\_length* to a multiple of the cache line size.

*flags* should be set to `DDI_DMA_CONSISTENT` if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using `DDI_DMA_CONSISTENT`.

The device access attributes are specified in the location pointed by the *accattrp* argument (see `ddi_device_acc_attr(9S)`).

The data access handle is returned in *handlep*. *handlep* is opaque – drivers may not attempt to interpret its value. To access the data content, the driver must invoke `ddi_get8(9F)` or `ddi_put8(9F)` (depending on the data transfer direction) with the data access handle.

DMA resources must be established before performing a DMA transfer by passing *kaddrp* and *real\_length* as returned from `ddi_dma_mem_alloc()` and the flag `DDI_DMA_STREAMING` or `DDI_DMA_CONSISTENT` to `ddi_dma_addr_bind_handle(9F)`. In addition, to ensure the consistency of a memory object shared between the CPU and the device after a DMA transfer, explicit synchronization steps using `ddi_dma_sync(9F)` or `ddi_dma_unbind_handle(9F)` are required.

<b>RETURN VALUES</b>	<p>ddi_dma_mem_alloc() returns:</p> <p>DDI_SUCCESS      Memory successfully allocated.</p> <p>DDI_FAILURE      Memory allocation failed.</p>
<b>CONTEXT</b>	ddi_dma_mem_alloc() can be called from user or interrupt context, except when <i>waitfp</i> is set to DDI_DMA_SLEEP, in which case it can be called from user context only.
<b>SEE ALSO</b>	<p>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)</p> <p><i>Writing Device Drivers</i></p>
<b>WARNINGS</b>	If DDI_NEVERSWAP_ACC is specified, memory can be used for any purpose; but if either endian mode is specified, you must use ddi_get/put* and never anything else.

## ddi\_dma\_mem\_free(9F)

<b>NAME</b>	ddi_dma_mem_free – free previously allocated memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_dma_mem_free</b>(ddi_acc_handle_t *handlep);</pre>
<b>PARAMETERS</b>	<i>handlep</i> Pointer to the data access handle previously allocated by a call to ddi_dma_mem_alloc(9F).
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_dma_mem_free() deallocates the memory acquired by ddi_dma_mem_alloc(9F). In addition, it destroys the data access handle <i>handlep</i> associated with the memory.
<b>CONTEXT</b>	ddi_dma_mem_free() can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	ddi_dma_mem_alloc(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_dma_movwin – shift current DMA window
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_movwin(ddi_dma_handle_t handle, off_t *offp, uint_t     *lenp, ddi_dma_cookie_t *cookiep);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>This interface is obsolete. ddi_dma_getwin(9F) should be used instead.</p> <p><i>handle</i>            The DMA handle filled in by a call to ddi_dma_setup(9F).</p> <p><i>offp</i>              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.</p> <p><i>lenp</i>              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.</p> <p><i>cookiep</i>          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.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_movwin() shifts the current DMA window. If a DMA request allows the system to allocate resources for less than the entire object by setting the DDI_DMA_PARTIAL flag in the ddi_dma_req(9S) structure, the current DMA window can be shifted by a call to ddi_dma_movwin().</p> <p>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.</p> <p>ddi_dma_movwin() takes care of underlying resource synchronizations required to shift the window. However, if you want to <i>access</i> the data prior to or after moving the window, further synchronizations using ddi_dma_sync(9F) are required.</p> <p>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.</p>
<b>RETURN VALUES</b>	<p>ddi_dma_movwin() returns:</p> <p>DDI_SUCCESS      The current length and offset are legal and have been set.</p> <p>DDI_FAILURE      Otherwise.</p>

## ddi\_dma\_movwin(9F)

**CONTEXT** | `ddi_dma_movwin()` can be called from user or interrupt context.

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

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.

<b>NAME</b>	ddi_dma_nextcookie – retrieve subsequent DMA cookie
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_dma_nextcookie</b>(ddi_dma_handle_t <i>handle</i>, ddi_dma_cookie_t     *<i>cookiep</i>);</pre>
<b>PARAMETERS</b>	<p><i>handle</i>            The handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p> <p><i>cookiep</i>           A pointer to a ddi_dma_cookie(9S) structure.</p>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	<p>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 <i>cookiep</i>. The ddi_dma_cookie(9S) structure must be allocated prior to calling ddi_dma_nextcookie().</p> <p>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, <i>N</i>, is larger than 1, ddi_dma_nextcookie() must be called <i>N</i>-1 times to retrieve all DMA cookies.</p>
<b>CONTEXT</b>	ddi_dma_nextcookie() can be called from user, kernel, or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> process a scatter-gather list of I/O requests</p> <p>This example demonstrates the use of ddi_dma_nextcookie() to process a scatter-gather list of I/O requests.</p> <pre>/* 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, &amp;dmacookie, &amp;ccount);  if (status == DDI_DMA_MAPPED) {      /* program DMA engine with first cookie */      while (--ccount &gt; 0) {         ddi_dma_nextcookie(handle, &amp;dmacookie);         /* program DMA engine with next cookie */     } } . . .</pre>

ddi\_dma\_nextcookie(9F)

**EXAMPLE 1** process a scatter-gather list of I/O requests     *(Continued)*

**SEE ALSO**   ddi\_dma\_addr\_bind\_handle(9F), ddi\_dma\_alloc\_handle(9F),  
ddi\_dma\_buf\_bind\_handle(9F), ddi\_dma\_unbind\_handle(9F),  
ddi\_dma\_cookie(9S)

*Writing Device Drivers*

<b>NAME</b>	ddi_dma_nextseg – get next DMA segment
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_nextseg(ddi_dma_win_t win, ddi_dma_seg_t seg,     ddi_dma_seg_t *nseg);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>This interface is obsolete. ddi_dma_nextcookie(9F) should be used instead.</p> <p><i>win</i>                    A DMA window.</p> <p><i>seg</i>                    The current DMA segment or NULL.</p> <p><i>nseg</i>                   A pointer to the next DMA segment to be filled in. If <i>seg</i> is NULL, a pointer to the first segment within the specified window is returned.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_nextseg() gets the next DMA segment within the specified window <i>win</i>. If the current segment is NULL, the first DMA segment within the window is returned.</p> <p>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.</p> <p>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.</p>
<b>RETURN VALUES</b>	<p>ddi_dma_nextseg() returns:</p> <p>DDI_SUCCESS            Successfully filled in the next segment pointer.</p> <p>DDI_DMA_DONE            There is no next segment. The current segment is the final segment within the specified window.</p> <p>DDI_DMA_STALE            <i>win</i> does not refer to the currently active window.</p>
<b>CONTEXT</b>	ddi_dma_nextseg() can be called from user or interrupt context.
<b>EXAMPLES</b>	For an example, see ddi_dma_segtocookie(9F).
<b>ATTRIBUTES</b>	See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

ddi\_dma\_nextseg(9F)

**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*

<b>NAME</b>	ddi_dma_nextwin – get next DMA window
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_nextwin(ddi_dma_handle_t handle, ddi_dma_win_t win,     ddi_dma_win_t *nwin);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>This interface is obsolete. ddi_dma_getwin(9F) should be used instead.</p> <p><i>handle</i>                    A DMA handle.</p> <p><i>win</i>                        The current DMA window or NULL.</p> <p><i>nwin</i>                      A pointer to the next DMA window to be filled in. If <i>win</i> is NULL, a pointer to the first window within the object is returned.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_nextwin() shifts the current DMA window <i>win</i> within the object referred to by <i>handle</i> to the next DMA window <i>nwin</i>. 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.</p> <p>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).</p> <p>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 <i>severe</i> error to call ddi_dma_nextwin() before any transfers into the current window are complete.</p> <p>ddi_dma_nextwin() takes care of underlying memory synchronizations required to shift the window. However, if you want to access the data before or after moving the window, further synchronizations using ddi_dma_sync(9F) are required.</p>
<b>RETURN VALUES</b>	<p>ddi_dma_nextwin() returns:</p> <p>DDI_SUCCESS            Successfully filled in the next window pointer.</p> <p>DDI_DMA_DONE          There is no next window. The current window is the final window within the specified object.</p> <p>DDI_DMA_STALE        <i>win</i> does not refer to the currently active window.</p>
<b>CONTEXT</b>	ddi_dma_nextwin() can be called from user or interrupt context.

ddi\_dma\_nextwin(9F)

**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*

<b>NAME</b>	ddi_dma_numwin – retrieve number of DMA windows
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_numwin(ddi_dma_handle_t handle, uint_t *nwinp);</pre>
<b>PARAMETERS</b>	<p><i>handle</i>            The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p> <p><i>nwinp</i>            Upon a successful return, <i>nwinp</i> will contain the number of DMA windows for this object.</p>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_dma_numwin() returns the number of DMA windows for a DMA object if partial resource allocation was permitted.
<b>RETURN VALUES</b>	<p>ddi_dma_numwin() returns:</p> <p>DDI_SUCCESS            Successfully filled in the number of DMA windows.</p> <p>DDI_FAILURE            DMA windows are not activated.</p>
<b>CONTEXT</b>	ddi_dma_numwin() can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	<p>ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F), ddi_dma_buf_bind_handle(9F), ddi_dma_unbind_handle(9F)</p> <p><i>Writing Device Drivers</i></p>

## ddi\_dma\_segtocookie(9F)

<b>NAME</b>	ddi_dma_segtocookie – convert a DMA segment to a DMA address cookie								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_segtocookie(ddi_dma_seg_t seg, off_t *offp, off_t *lenp,     ddi_dma_cookie_t *cookiep);</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	This interface is obsolete. ddi_dma_nextcookie(9F) should be used instead.  <table><tr><td><i>seg</i></td><td>A DMA segment.</td></tr><tr><td><i>offp</i></td><td>A pointer to an <i>off_t</i>. Upon a successful return, it is filled in with the offset. This segment is addressing within the object.</td></tr><tr><td><i>lenp</i></td><td>The byte length. This segment is addressing within the object.</td></tr><tr><td><i>cookiep</i></td><td>A pointer to a DMA cookie (see ddi_dma_cookie(9S)).</td></tr></table>	<i>seg</i>	A DMA segment.	<i>offp</i>	A pointer to an <i>off_t</i> . Upon a successful return, it is filled in with the offset. This segment is addressing within the object.	<i>lenp</i>	The byte length. This segment is addressing within the object.	<i>cookiep</i>	A pointer to a DMA cookie (see ddi_dma_cookie(9S)).
<i>seg</i>	A DMA segment.								
<i>offp</i>	A pointer to an <i>off_t</i> . Upon a successful return, it is filled in with the offset. This segment is addressing within the object.								
<i>lenp</i>	The byte length. This segment is addressing within the object.								
<i>cookiep</i>	A pointer to a DMA cookie (see ddi_dma_cookie(9S)).								
<b>DESCRIPTION</b>	ddi_dma_segtocookie() takes a DMA segment and fills in the cookie pointed to by <i>cookiep</i> with the appropriate address, length, and bus type to be used to program the DMA engine. ddi_dma_segtocookie() also fills in <i>*offp</i> and <i>*lenp</i> , which specify the range within the object.								
<b>RETURN VALUES</b>	ddi_dma_segtocookie() returns:  DDI_SUCCESS                      Successfully filled in all values. DDI_FAILURE                      Failed to successfully fill in all values.								
<b>CONTEXT</b>	ddi_dma_segtocookie() can be called from user or interrupt context.								
<b>EXAMPLES</b>	<b>EXAMPLE 1</b> ddi_dma_segtocookie() example <pre>for (win = NULL; (retw = ddi_dma_nextwin(handle, win, &amp;nwin)) !=     DDI_DMA_DONE; win = nwin) {     if (retw != DDI_SUCCESS) {         /* do error handling */     } else {         for (seg = NULL; (rets = ddi_dma_nextseg(nwin, seg, &amp;nseg)) !=             DDI_DMA_DONE; seg = nseg) {             if (rets != DDI_SUCCESS) {                  /* do error handling */             } else {                 ddi_dma_segtocookie(nseg, &amp;off, &amp;len, &amp;cookie);                  /* program DMA engine */             }         }     } }</pre>								

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_nextcookie(9F)`, `ddi_dma_nextseg(9F)`,  
`ddi_dma_nextwin(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_cookie(9S)`

*Writing Device Drivers*

## ddi\_dma\_set\_sbus64(9F)

<b>NAME</b>	ddi_dma_set_sbus64 – allow 64-bit transfers on SBus				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_set_sbus64(ddi_dma_handle_t handle, uint_t burstsizes);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>                    The handle filled in by a call to ddi_dma_alloc_handle(9F).</p> <p><i>burstsizes</i>                The possible burst sizes the device's DMA engine can accept in 64-bit mode.</p>				
<b>DESCRIPTION</b>	<p>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.</p> <p><i>burstsizes</i> 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.</p> <p>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.</p>				
<b>RETURN VALUES</b>	<p>ddi_dma_set_sbus64 () returns:</p> <p>DDI_SUCCESS      Successfully set the SBus to 64-bit mode.</p> <p>DDI_FAILURE      64-bit mode could not be set.</p>				
<b>CONTEXT</b>	ddi_dma_set_sbus64 () can be called from user, kernel, or interrupt context.				
<b>ATTRIBUTES</b>	See attributes(5) for descriptions of the following attributes:				
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">ATTRIBUTE TYPE</th> <th style="text-align: center;">ATTRIBUTE VALUE</th> </tr> </thead> <tbody> <tr> <td>Architecture</td> <td>SBus</td> </tr> </tbody> </table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Architecture	SBus
ATTRIBUTE TYPE	ATTRIBUTE VALUE				
Architecture	SBus				
<b>SEE ALSO</b>	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)				
<b>NOTES</b>	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.				

<b>NAME</b>	ddi_dma_setup – setup DMA resources
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dma_setup</b>(dev_info_t *dip, ddi_dma_req_t *dmareqp,     ddi_dma_handle_t *handlep);</pre>
<b>INTERFACE LEVEL</b>	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.
<b>PARAMETERS</b>	<p><i>dip</i>                    A pointer to the device's dev_info structure.</p> <p><i>dmareqp</i>                A pointer to a DMA request structure (see ddi_dma_req(9S)).</p> <p><i>handlep</i>                A pointer to a DMA handle to be filled in. See below for a discussion of a handle. If <i>handlep</i> 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.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_setup() allocates resources for a memory object such that a device can perform DMA to or from that object.</p> <p>A call to ddi_dma_setup() informs the system that device referred to by <i>dip</i> 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 <i>dmareqp</i>.</p> <p>A successful call to ddi_dma_setup() fills in the value pointed to by <i>handlep</i>. 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.</p> <p>Again a DMA handle is opaque—drivers may <i>not</i> 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.</p> <p>When DMA transfer completes, the driver should free up the the allocated DMA resources by calling ddi_dma_free().</p>
<b>RETURN VALUES</b>	<p>ddi_dma_setup() returns:</p> <p>DDI_DMA_MAPPED            Successfully allocated resources for the object. In the case of an <i>advisory</i> call, this indicates that the request is legal.</p> <p>DDI_DMA_PARTIAL_MAP        Successfully allocated resources for a <i>part</i> of the object. This is acceptable when partial transfers are allowed</p>

## ddi\_dma\_setup(9F)

using a flag setting in the `ddi_dma_req` structure (see `ddi_dma_req(9S)` and `ddi_dma_movwin(9F)`).

DDI_DMA_NORESOURCES	When no resources are available.
DDI_DMA_NOMAPPING	The object cannot be reached by the device requesting the resources.
DDI_DMA_TOOBIG	The object is too big and exceeds the available resources. The maximum size varies depending on machine and configuration.

**CONTEXT** `ddi_dma_setup()` can be called from user or interrupt context, except when the `dmar_fp` member of the `ddi_dma_req` structure pointed to by `dmareqp` is set to `DDI_DMA_SLEEP`, in which case it can be called from user context only.

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

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.

<b>NAME</b>	ddi_dma_sync – synchronize CPU and I/O views of memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_dma_sync(ddi_dma_handle_t handle, off_t offset, size_t length,                 uint_t type);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The <i>handle</i> filled in by a call to ddi_dma_alloc_handle(9F).</p> <p><i>offset</i>            The offset into the object described by the <i>handle</i>.</p> <p><i>length</i>            The length, in bytes, of the area to synchronize. When <i>length</i> is zero, the entire range starting from <i>offset</i> to the end of the object has the requested operation applied to it.</p> <p><i>type</i>              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.</p>
<b>DESCRIPTION</b>	<p>ddi_dma_sync() is used to selectively synchronize either a DMA device's or a CPU's view of a memory object that has DMA resources allocated for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.</p> <p>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 <i>did not</i> 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)).</p> <p>This cannot be stated too strongly. If a consistent view of the memory object must be ensured between the time DMA resources are allocated for the object and the time they are deallocated, you <i>must</i> call ddi_dma_sync() to ensure that either a CPU or a DMA device has such a consistent view.</p> <p>What to set <i>type</i> 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 <i>written</i> to the memory object, and you are going to <i>read</i> (with a CPU) the object (using an extant virtual address mapping that you have to</p>

## ddi\_dma\_sync(9F)

the memory object), use `DDI_DMA_SYNC_FORCPU`. This ensures that a CPU's view of the memory object includes any changes made to the object by the device's DMA engine. If you are only interested in the kernel's view (kernel-space part of the CPU's view) you may use `DDI_DMA_SYNC_FORKERNEL`. This gives a hint to the system—that is, if it is more economical to synchronize the kernel's view only, then do so; otherwise, synchronize for CPU.

### RETURN VALUES

`ddi_dma_sync()` returns:

`DDI_SUCCESS` Caches are successfully flushed.  
`DDI_FAILURE` The address range to be flushed is out of the address range established by `ddi_dma_addr_bind_handle(9F)` or `ddi_dma_buf_bind_handle(9F)`.

### CONTEXT

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

### SEE ALSO

`ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`,  
`ddi_dma_buf_bind_handle(9F)`, `ddi_dma_mem_alloc(9F)`,  
`ddi_dma_unbind_handle(9F)`

*Writing Device Drivers*

<b>NAME</b>	ddi_dma_unbind_handle – unbinds the address in a DMA handle
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_dma_unbind_handle</b>(ddi_dma_handle_t <i>handle</i>);</pre>
<b>PARAMETERS</b>	<p><i>handle</i>                    The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).</p>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	<p>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.</p>
<b>RETURN VALUES</b>	<p>DDI_SUCCESS            on success</p> <p>DDI_FAILURE            on failure</p>
<b>CONTEXT</b>	ddi_dma_unbind_handle() can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	<p>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)</p> <p><i>Writing Device Drivers</i></p>

## ddi\_driver\_major(9F)

<b>NAME</b>	ddi_driver_major – return driver’s major device number
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  major_t ddi_driver_major(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>DESCRIPTION</b>	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.
<b>PARAMETERS</b>	<i>dip</i> A pointer to the device’s dev_info structure.
<b>RETURN VALUES</b>	ddi_driver_major() returns the major number of the driver bound to a device, if any, or DDI_MAJOR_T_NONE otherwise.
<b>CONTEXT</b>	ddi_driver_major() can be called from kernel or interrupt context.
<b>SEE ALSO</b>	ddi_driver_name(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_driver_name – return normalized driver name
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  const char *<b>ddi_driver_name</b>(dev_info_t *devi);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<b>dip</b> A pointer to the device's dev_info structure.
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	ddi_driver_name() returns the actual name of the driver bound to a device.
<b>CONTEXT</b>	ddi_driver_name() can be called from kernel, or interrupt context.
<b>SEE ALSO</b>	ddi_get_name(9F) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	The name returned by ddi_driver_name() is read-only.

## ddi\_enter\_critical(9F)

<b>NAME</b>	ddi_enter_critical, ddi_exit_critical – enter and exit a critical region of control
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  unsigned int <b>ddi_enter_critical</b>(void);  void <b>ddi_exit_critical</b>(unsignedint <i>ddic</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>ddic</i> The returned value from the call to ddi_enter_critical() must be passed to ddi_exit_critical().
<b>DESCRIPTION</b>	<p>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 <i>must</i> 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()).</p> <p>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().</p>
<b>RETURN VALUES</b>	ddi_enter_critical() returns an opaque unsigned integer which must be used in the subsequent call to ddi_exit_critical().
<b>CONTEXT</b>	This function can be called from user or interrupt context.
<b>WARNINGS</b>	<p>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 <i>can</i> do is guarantee that the currently executing thread will not be preempted.</p> <p>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.</p>
<b>SEE ALSO</b>	mutex(9F)  <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_ffs, ddi_fls – find first (last) bit set in a long integer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_ffs(long mask); int ddi_fls(long mask);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>mask</i> A 32-bit argument value to search through.
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	<p>0            No bits are set in mask.</p> <p><i>N</i>            Bit <i>N</i> 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.</p>
<b>CONTEXT</b>	This function can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

## ddi\_get8(9F)

<b>NAME</b>	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						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <i>handle</i> The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).  <i>dev_addr</i> Base device address.						
<b>DESCRIPTION</b>	<p>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, <i>dev_addr</i>.</p> <p>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.</p> <p>For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).</p>						
<b>RETURN VALUES</b>	These functions return the value read from the mapped address.						
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.						
<b>SEE ALSO</b>	ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F)						
<b>NOTES</b>	The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:						
	<table border="1"><thead><tr><th>Previous Name</th><th>New Name</th></tr></thead><tbody><tr><td>ddi_getb</td><td>ddi_get8</td></tr><tr><td>ddi_getw</td><td>ddi_get16</td></tr></tbody></table>	Previous Name	New Name	ddi_getb	ddi_get8	ddi_getw	ddi_get16
Previous Name	New Name						
ddi_getb	ddi_get8						
ddi_getw	ddi_get16						

ddi\_get8(9F)

Previous Name	New Name
ddi_get1	ddi_get32
ddi_get11	ddi_get64

## ddi\_get\_cred(9F)

<b>NAME</b>	ddi_get_cred – returns a pointer to the credential structure of the caller
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  cred_t *ddi_get_cred(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_get_cred() returns a pointer to the user credential structure of the caller.
<b>RETURN VALUES</b>	ddi_get_cred() returns a pointer to the caller's credential structure.
<b>CONTEXT</b>	ddi_get_cred() can be called from user context only.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

<b>NAME</b>	ddi_get_devstate – Check device state
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  ddi_devstate_t ddi_get_devstate(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<i>dip</i> Pointer to the device's dev_info structure
<b>DESCRIPTION</b>	The ddi_get_devstate() function returns a value indicating the state of the device specified by <i>dip</i> , 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.
<b>RETURN VALUES</b>	<p>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.</p> <p>DDI_DEVSTATE_DOWN The device is online but unusable due to a fault.</p> <p>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.</p> <p>DDI_DEVSTATE_DEGRADED The device is online but only able to provide a partial or degraded service, due to a fault.</p> <p>DDI_DEVSTATE_UP The device is online and fully operational.</p>
<b>CONTEXT</b>	The ddi_get_devstate() function may be called from user, kernel, or interrupt context.
<b>NOTES</b>	<p>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).</p> <p>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.</p>
<b>SEE ALSO</b>	attach(9E), ioctl(9E), open(9E), read(9E), strategy(9E), write(9E), ddi_dev_report_fault(9F)

## ddi\_get\_driver\_private(9F)

<b>NAME</b>	ddi_get_driver_private, ddi_set_driver_private – get or set the address of the device's private data area
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_set_driver_private</b>(dev_info_t *dip, caddr_t data); caddr_t <b>ddi_get_driver_private</b>(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p>ddi_get_driver_private()</p> <p><i>dip</i>        Pointer to device information structure to get from.</p> <p>ddi_set_driver_private()</p> <p><i>dip</i>        Pointer to device information structure to set.</p> <p><i>data</i>       Data area address to set.</p>
<b>DESCRIPTION</b>	<p>ddi_get_driver_private() returns the address of the device's private data area from the device information structure pointed to by <i>dip</i>.</p> <p>ddi_set_driver_private() sets the address of the device's private data area in the device information structure pointed to by <i>dip</i> with the value of <i>data</i>.</p>
<b>RETURN VALUES</b>	ddi_get_driver_private() returns the contents of devi_driver_data. If ddi_set_driver_private() has not been previously called with <i>dip</i> , an unpredictable value is returned.
<b>CONTEXT</b>	These functions can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

<b>NAME</b>	ddi_getminor – get kernel internal minor number from an external dev_t				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/mkdev.h&gt; #include &lt;sys/ddi.h&gt;  minor_t ddi_getminor(dev_t dev);</pre>				
<b>INTERFACE LEVEL</b>	This interface is obsolete. getminor(9F) should be used instead.				
<b>PARAMETERS</b>	<p>The following parameters are supported:</p> <p><i>dev</i> Device number.</p>				
<b>DESCRIPTION</b>	<p>ddi_getminor() 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.</p> <p>For certain bus types, you can call this DDI function from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details.</p>				
<b>CONTEXT</b>	ddi_getminor() can be called from user context only.				
<b>RETURN VALUES</b>	The minor number or EMINOR_UNKNOWN if the minor number of the device is invalid.				
<b>ATTRIBUTES</b>	See attributes(5) for a description of the following attributes:				
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">ATTRIBUTE TYPE</th> <th style="text-align: center;">ATTRIBUTE VALUE</th> </tr> </thead> <tbody> <tr> <td>Stability Level</td> <td>Obsolete</td> </tr> </tbody> </table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Stability Level	Obsolete
ATTRIBUTE TYPE	ATTRIBUTE VALUE				
Stability Level	Obsolete				
<b>SEE ALSO</b>	attributes(5), getmajor(9F), getminor(9F), makedevice(9F)  <i>Writing Device Drivers</i>				
<b>WARNINGS</b>	Validity checking is performed. If <i>dev</i> is invalid, EMINOR_UNKNOWN is returned. This behavior differs from getminor(9F).				

## ddi\_get\_instance(9F)

<b>NAME</b>	ddi_get_instance – get device instance number
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_get_instance(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> Pointer to dev_info structure.
<b>DESCRIPTION</b>	<p>ddi_get_instance() returns the instance number of the device corresponding to <i>dip</i>.</p> <p>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.</p> <p>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 <code>f000</code> has been assigned an instance number of 0 and device <code>f001</code> has been assigned an instance number of 1, if <code>f000</code> is removed, <code>f001</code> will continue to be associated with instance number 1 (even though <code>f001</code> is now the only device of its type on the system).</p>
<b>RETURN VALUES</b>	ddi_get_instance() returns the instance number of the device corresponding to <i>dip</i> .
<b>CONTEXT</b>	ddi_get_instance() can be called from user or interrupt context.
<b>SEE ALSO</b>	path_to_inst(4) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_get_kt_did – get identifier of current thread
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  kt_did_t ddi_get_kt_did(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>DESCRIPTION</b>	The ddi_get_kt_did() function returns a unique 64-bit identifier for the currently running thread.
<b>CONTEXT</b>	This routine can be called from user, kernel, or interrupt context. This routine cannot be called from a high-level interrupt context.
<b>RETURN VALUES</b>	ddi_get_kt_did() always returns the identifier for the current thread. There are no error conditions.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>
<b>NOTES</b>	<p>The value returned by this function can also be seen in adb or mdb as the did field displayed when using the thread macro.</p> <p>This interface is intended for tracing and debugging purposes.</p>

## ddi\_get\_lbolt(9F)

<b>NAME</b>	ddi_get_lbolt – returns the value of lbolt
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  clock_t ddi_get_lbolt(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_get_lbolt() returns the value of lbolt where lbolt is an integer that represents the number of clock ticks since the last system reboot. This value is used as a counter or timer inside the system kernel. The tick frequency can be determined by using drv_usec2hz(9F) which converts microseconds into clock ticks.
<b>RETURN VALUES</b>	ddi_get_lbolt() returns the value of lbolt.
<b>CONTEXT</b>	This routine can be called from any context.
<b>SEE ALSO</b>	ddi_get_time(9F), drv_getparm(9F), drv_usec2hz(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	ddi_get_parent – find the parent of a device information structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  dev_info_t *<b>ddi_get_parent</b>(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> Pointer to a device information structure.
<b>DESCRIPTION</b>	ddi_get_parent() returns a pointer to the device information structure which is the parent of the one pointed to by <i>dip</i> .
<b>RETURN VALUES</b>	ddi_get_parent() returns a pointer to a device information structure.
<b>CONTEXT</b>	ddi_get_parent() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

## ddi\_get\_pid(9F)

<b>NAME</b>	ddi_get_pid – returns the process ID
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  pid_t ddi_get_pid(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	ddi_get_pid() returns the process ID.
<b>CONTEXT</b>	This routine can be called from user context only.
<b>SEE ALSO</b>	drv_getparm(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	ddi_get_time – returns the current time in seconds
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  time_t ddi_get_time(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	ddi_get_time() returns the time in seconds.
<b>CONTEXT</b>	This routine can be called from any context.
<b>SEE ALSO</b>	ddi_get_lbolt(9F), drv_getparm(9F), drv_usec2ohz(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

## ddi\_in\_panic(9F)

<b>NAME</b>	ddi_in_panic – determine if system is in panic state
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_in_panic(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	<p>Drivers controlling devices on which the system may write a kernel crash dump in the event of a panic can call <code>ddi_in_panic()</code> to determine if the system is panicking.</p> <p>When the system is panicking, the calls of functions scheduled by <code>timeout(9F)</code> and <code>ddi_trigger_softintr(9F)</code> will never occur. Neither can <code>delay(9F)</code> be relied upon, since it is implemented via <code>timeout(9F)</code>.</p> <p>Drivers that need to enforce a time delay such as SCSI bus reset delay time must busy-wait when the system is panicking.</p>
<b>RETURN VALUES</b>	<code>ddi_in_panic()</code> returns 1 if the system is in panic, or 0 otherwise.
<b>CONTEXT</b>	<code>ddi_in_panic()</code> may be called from any context.
<b>SEE ALSO</b>	<code>dump(9E)</code> , <code>delay(9F)</code> , <code>ddi_trigger_softintr(9F)</code> , <code>timeout(9F)</code> <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_intr_hilevel – indicate interrupt handler type
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_intr_hilevel(dev_info_t *dip, uint_t inumber);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i>                    Pointer to dev_info structure.</p> <p><i>inumber</i>                Interrupt number.</p>
<b>DESCRIPTION</b>	<p>ddi_intr_hilevel() returns non-zero if the specified interrupt is a "high level" interrupt.</p> <p>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.</p> <p>In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable.</p> <p>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.</p> <p>ddi_intr_hilevel() can be used before calling ddi_add_intr() to decide which type of interrupt handler should be used. Most device drivers are designed with the knowledge that the devices they support will always generate low level interrupts, however some devices, for example those using SBus or VME bus level 6 or 7 interrupts must use this test because on some machines those interrupts are high level (above the scheduler level) and on other machines they are not.</p>
<b>RETURN VALUES</b>	non-zero                indicates a high-level interrupt.
<b>CONTEXT</b>	These functions can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>ddi_add_intr(9F), mutex(9F)</p> <p><i>Writing Device Drivers</i></p>

## ddi\_io\_get8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>handle</i> Data access handle returned from setup calls, such as ddi_regs_map_setup(9F). <i>dev_addr</i> Device address.
<b>DESCRIPTION</b>	<p>These routines generate a read of various sizes from the device address, <i>dev_addr</i>, 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, <i>dev_addr</i>.</p> <p>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.</p>
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	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)
<b>NOTES</b>	<p>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.</p> <p>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:</p>

Previous Name	New Name
ddi_io_getb	ddi_io_get8
ddi_io_getw	ddi_io_get16
ddi_io_getl	ddi_io_get32

<b>NAME</b>	ddi_iomin – find minimum alignment and transfer size for DMA						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_iomin(dev_info_t *dip, int initial, int streaming);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>PARAMETERS</b>	<table> <tr> <td><i>dip</i></td> <td>A pointer to the device's dev_info structure.</td> </tr> <tr> <td><i>initial</i></td> <td>The initial minimum DMA transfer size in bytes. This may be zero or an appropriate dlim_minxfer value for device's ddi_dma_lim structure (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). This value must be a power of two.</td> </tr> <tr> <td><i>streaming</i></td> <td>This argument, if non-zero, indicates that the returned value should be modified to account for <i>streaming</i> mode accesses (see ddi_dma_req(9S) for a discussion of streaming versus non-streaming access mode).</td> </tr> </table>	<i>dip</i>	A pointer to the device's dev_info structure.	<i>initial</i>	The initial minimum DMA transfer size in bytes. This may be zero or an appropriate dlim_minxfer value for device's ddi_dma_lim structure (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). This value must be a power of two.	<i>streaming</i>	This argument, if non-zero, indicates that the returned value should be modified to account for <i>streaming</i> mode accesses (see ddi_dma_req(9S) for a discussion of streaming versus non-streaming access mode).
<i>dip</i>	A pointer to the device's dev_info structure.						
<i>initial</i>	The initial minimum DMA transfer size in bytes. This may be zero or an appropriate dlim_minxfer value for device's ddi_dma_lim structure (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). This value must be a power of two.						
<i>streaming</i>	This argument, if non-zero, indicates that the returned value should be modified to account for <i>streaming</i> mode accesses (see ddi_dma_req(9S) for a discussion of streaming versus non-streaming access mode).						
<b>DESCRIPTION</b>	ddi_iomin(), finds out the minimum DMA transfer size for the device pointed to by <i>dip</i> . 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.						
<b>RETURN VALUES</b>	ddi_iomin() returns the minimum DMA transfer size for the calling device, or it returns zero, which means that you cannot get there from here.						
<b>CONTEXT</b>	This function can be called from user or interrupt context.						
<b>SEE ALSO</b>	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)  <i>Writing Device Drivers</i>						

## ddi\_iopb\_alloc(9F)

<b>NAME</b>	ddi_iopb_alloc, ddi_iopb_free – allocate and free non-sequentially accessed memory								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>								
<b>INTERFACE LEVEL</b>	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().								
<b>PARAMETERS</b>									
<b>ddi_iopb_alloc()</b>	<table><tr><td><i>dip</i></td><td>A pointer to the device's dev_info structure.</td></tr><tr><td><i>limits</i></td><td>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.</td></tr><tr><td><i>length</i></td><td>The length in bytes of the desired allocation.</td></tr><tr><td><i>iopbp</i></td><td>A pointer to a caddr_t. On a successful return, *iopbp points to the allocated storage.</td></tr></table>	<i>dip</i>	A pointer to the device's dev_info structure.	<i>limits</i>	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.	<i>length</i>	The length in bytes of the desired allocation.	<i>iopbp</i>	A pointer to a caddr_t. On a successful return, *iopbp points to the allocated storage.
<i>dip</i>	A pointer to the device's dev_info structure.								
<i>limits</i>	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.								
<i>length</i>	The length in bytes of the desired allocation.								
<i>iopbp</i>	A pointer to a caddr_t. On a successful return, *iopbp points to the allocated storage.								
<b>ddi_iopb_free()</b>	<table><tr><td><i>iopb</i></td><td>The iopb returned from a successful call to ddi_iopb_alloc().</td></tr></table>	<i>iopb</i>	The iopb returned from a successful call to ddi_iopb_alloc().						
<i>iopb</i>	The iopb returned from a successful call to ddi_iopb_alloc().								
<b>DESCRIPTION</b>	<p>ddi_iopb_alloc() allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using ddi_dma_sync(9F) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as <i>consistent</i> access. The allocation will obey the alignment and padding constraints as specified in the <i>limits</i> argument and other limits imposed by the system.</p> <p>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().</p> <p>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.</p> <p>ddi_iopb_free() frees up memory allocated by ddi_iopb_alloc().</p>								
<b>RETURN VALUES</b>	ddi_iopb_alloc() returns: <table><tr><td>DDI_SUCCESS</td><td>Memory successfully allocated.</td></tr><tr><td>DDI_FAILURE</td><td>Allocation failed.</td></tr></table>	DDI_SUCCESS	Memory successfully allocated.	DDI_FAILURE	Allocation failed.				
DDI_SUCCESS	Memory successfully allocated.								
DDI_FAILURE	Allocation failed.								
<b>CONTEXT</b>	These functions can be called from user 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)`, `ddi_dma_free(9F)`, `ddi_dma_mem_alloc(9F)`, `ddi_dma_mem_free(9F)`, `ddi_dma_setup(9F)`, `ddi_dma_sync(9F)`, `ddi_mem_alloc(9F)`, `ddi_dma_lim_sparc(9S)`, `ddi_dma_lim_x86(9S)`, `ddi_dma_req(9S)`

*Writing Device Drivers*

**NOTES** This function uses scarce system resources. Use it selectively.

## ddi\_io\_put8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>handle</i> Data access handle returned from setup calls, such as ddi_regs_map_setup(9F). <i>dev_addr</i> Base device address. <i>value</i> Data to be written to the device.
<b>DESCRIPTION</b>	<p>These routines generate a write of various sizes to the device address, <i>dev_addr</i>, 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, <i>dev_addr</i>.</p> <p>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.</p>
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	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)
<b>NOTES</b>	<p>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.</p> <p>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:</p>

Previous Name	New Name
ddi_io_putb	ddi_io_put8

ddi\_io\_put8(9F)

Previous Name	New Name
ddi_io_putw	ddi_io_put16
ddi_io_putl	ddi_io_put32

## ddi\_io\_rep\_get8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,     uint8_t *dev_addr, , size_t recount);  void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,     uint16_t *dev_addr, , size_t recount);  void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,     uint32_t *dev_addr, , size_t recount);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <i>handle</i> The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).  <i>host_addr</i> Base host address.  <i>dev_addr</i> Base device address.  <i>recount</i> Number of data accesses to perform.
<b>DESCRIPTION</b>	<p>These routines generate multiple reads from the device address, <i>dev_addr</i>, in I/O space. <i>recount</i> data is copied from the device address, <i>dev_addr</i>, to the host address, <i>host_addr</i>. For each input datum, the <code>ddi_io_rep_get8()</code>, <code>ddi_io_rep_get16()</code>, and <code>ddi_io_rep_get32()</code> functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address. <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	<code>isa(4)</code> , <code>ddi_io_get8(9F)</code> , <code>ddi_io_put8(9F)</code> , <code>ddi_io_rep_put8(9F)</code> , <code>ddi_regs_map_free(9F)</code> , <code>ddi_regs_map_setup(9F)</code> , <code>ddi_device_acc_attr(9S)</code>
<b>NOTES</b>	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 <code>isa(4)</code> ) 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.

## ddi\_io\_rep\_get8(9F)

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

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

## ddi\_io\_rep\_put8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,     uint8_t *dev_addr, size_t repcount);  void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,     uint16_t *dev_addr, size_t repcount);  void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,     uint32_t *dev_addr, size_t repcount);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>handle</i> Data access handle returned from setup calls, such as ddi_regs_map_setup(9F). <i>host_addr</i> Base host address. <i>dev_addr</i> Base device address. <i>repcount</i> Number of data accesses to perform.
<b>DESCRIPTION</b>	<p>These routines generate multiple writes to the device address, <i>dev_address</i>, in I/O space. <i>repcount</i> data is copied from the host address, <i>host_addr</i>, to the device address, <i>dev_addr</i>. 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. <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	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)
<b>NOTES</b>	<p>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.</p> <p>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:</p>

ddi\_io\_rep\_put8(9F)

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

## ddi\_log\_sysevent(9F)

<b>NAME</b>	ddi_log_sysevent – log system event for drivers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).
	<p><i>dip</i> A pointer to the <code>dev_info</code> node for this driver.</p> <p><i>vendor</i> 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 <code>DDI_VENDOR_SUNW</code>.</p> <p><i>class</i> A pointer to a string defining the event class.</p> <p><i>subclass</i> A pointer to a string defining the event subclass.</p> <p><i>attr_list</i> A pointer to an <code>nvlist_t</code>, listing the name-value attributes associated with the event or NULL if there are no such attributes for this event.</p> <p><i>eidp</i> The address of a <code>sysevent_id_t</code> 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 <code>sysevent_id_t</code>.</p> <p><i>sleep_flag</i> Indicates how a caller wants to handle the possibility of resources not being available. If <i>sleep_flag</i> is <code>DDI_NOSLEEP</code>, the caller does not care if the allocation fails or the queue is full and can handle a failure appropriately. If <i>sleep_flag</i> is <code>DDI_SLEEP</code>, the caller wishes to have the allocation and queuing routines wait for resources to become available.</p>
<b>DESCRIPTION</b>	<p><code>ddi_log_sysevent()</code> causes a system event, of the specified class and subclass, to be generated on behalf of the driver and queued for delivery to <code>syseventd</code>, the user-land <code>sysevent</code> daemon.</p> <p>The publisher string for the event is constructed using the vendor name and driver name, with the format:</p> <pre>"&lt;vendor&gt;:kern:&lt;driver-name&gt;"</pre> <p>The two fields of <code>eidp</code>, <code>eid_seq</code> and <code>eid_ts</code>, are sufficient to uniquely identify an event.</p>
<b>STRUCTURE MEMBERS</b>	<p>The structure members of <code>sysevent_id_t</code> are:</p> <pre>uint64_t  eid_seq;      /* sysevent sequence number */ hrtime_t  eid_ts;      /* sysevent timestamp */</pre>
<b>RETURN VALUES</b>	<code>ddi_log_sysevent()</code> 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

## ddi\_log\_sysevent(9F)

DATA\_TYPE\_INT32\_ARRAY

DATA\_TYPE\_UINT32\_ARRAY

DATA\_TYPE\_INT64\_ARRAY

DATA\_TYPE\_UINT64\_ARRAY

**CONTEXT** `ddi_log_sysevent()` can be called from user or interrupt context, except when *sleep\_flag* is `DDI_SLEEP`, in which case it can be called from user context only.

**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;
```

**EXAMPLE 3** Use Timeout to Handle nvlist and System Event Resource Allocation Failures *(Continued)*

```

err = nvlist_alloc(&xx->xx_ev_attrlist, NV_UNIQUE_NAME_TYPE, 0);
if (err != 0)
    return (1);
err = nvlist_add_uint32(&xx->xx_ev_attrlist,
    xx->xx_ev_name, xx->xx_ev_value);
if (err != 0) {
    nvlist_free(xx->xx_ev_attrlist);
    return(1);
}

err = ddi_log_sysevent(xx->xx_dip, DDI_VENDOR_SUNW,
    xx->xx_ev_class, xx->xx_ev_sbclass,
    xx->xx_ev_attrlist, NULL, DDI_NOSLEEP);
nvlist_free(xx->xx_ev_attrlist);
if (err == DDI_SUCCESS || err == DDI_ETRANSPORT) {
    if (err == DDI_ETRANSPORT)
        cmn_err(CE_WARN, "cannot log system event\n");
    return (0);
}
return (1);
}

```

**SEE ALSO** syseventd(1M), attributes(5), nvlist\_add\_boolean(9F), nvlist\_alloc(9F)

*Writing Device Drivers*

## ddi\_mapdev(9F)

<b>NAME</b>	ddi_mapdev – create driver-controlled mapping of device																								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_mapdev(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 *cred,                struct ddi_mapdev_ctl *ctl, ddi_mapdev_handle_t *handlep, void                *devprivate);</pre>																								
<b>INTERFACE LEVEL PARAMETERS</b>	<p>This interface is obsolete. devmap_setup(9F) should be used instead.</p> <table><tr><td><i>dev</i></td><td>The device whose memory is to be mapped.</td></tr><tr><td><i>offset</i></td><td>The offset within device memory at which the mapping begins.</td></tr><tr><td><i>as</i></td><td>An opaque pointer to the user address space into which the device memory should be mapped.</td></tr><tr><td><i>addrp</i></td><td>Pointer to the starting address within the user address space to which the device memory should be mapped.</td></tr><tr><td><i>len</i></td><td>Length (in bytes) of the memory to be mapped.</td></tr><tr><td><i>prot</i></td><td>A bit field that specifies the protections.</td></tr><tr><td><i>maxprot</i></td><td>Maximum protection flag possible for attempted mapping.</td></tr><tr><td><i>flags</i></td><td>Flags indicating type of mapping.</td></tr><tr><td><i>cred</i></td><td>Pointer to the user credentials structure.</td></tr><tr><td><i>ctl</i></td><td>A pointer to a ddi_mapdev_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.</td></tr><tr><td><i>handlep</i></td><td>An opaque pointer to a device mapping handle. A handle to the new device mapping is generated and placed into the location pointed to by <i>*handlep</i>. If the call fails, the value of <i>*handlep</i> is undefined.</td></tr><tr><td><i>devprivate</i></td><td>Driver private mapping data. This value is passed into each mapping call back routine.</td></tr></table>	<i>dev</i>	The device whose memory is to be mapped.	<i>offset</i>	The offset within device memory at which the mapping begins.	<i>as</i>	An opaque pointer to the user address space into which the device memory should be mapped.	<i>addrp</i>	Pointer to the starting address within the user address space to which the device memory should be mapped.	<i>len</i>	Length (in bytes) of the memory to be mapped.	<i>prot</i>	A bit field that specifies the protections.	<i>maxprot</i>	Maximum protection flag possible for attempted mapping.	<i>flags</i>	Flags indicating type of mapping.	<i>cred</i>	Pointer to the user credentials structure.	<i>ctl</i>	A pointer to a ddi_mapdev_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.	<i>handlep</i>	An opaque pointer to a device mapping handle. A handle to the new device mapping is generated and placed into the location pointed to by <i>*handlep</i> . If the call fails, the value of <i>*handlep</i> is undefined.	<i>devprivate</i>	Driver private mapping data. This value is passed into each mapping call back routine.
<i>dev</i>	The device whose memory is to be mapped.																								
<i>offset</i>	The offset within device memory at which the mapping begins.																								
<i>as</i>	An opaque pointer to the user address space into which the device memory should be mapped.																								
<i>addrp</i>	Pointer to the starting address within the user address space to which the device memory should be mapped.																								
<i>len</i>	Length (in bytes) of the memory to be mapped.																								
<i>prot</i>	A bit field that specifies the protections.																								
<i>maxprot</i>	Maximum protection flag possible for attempted mapping.																								
<i>flags</i>	Flags indicating type of mapping.																								
<i>cred</i>	Pointer to the user credentials structure.																								
<i>ctl</i>	A pointer to a ddi_mapdev_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.																								
<i>handlep</i>	An opaque pointer to a device mapping handle. A handle to the new device mapping is generated and placed into the location pointed to by <i>*handlep</i> . If the call fails, the value of <i>*handlep</i> is undefined.																								
<i>devprivate</i>	Driver private mapping data. This value is passed into each mapping call back routine.																								
<b>DESCRIPTION</b>	<p>Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use devmap_setup(9F) instead. See devmap_setup(9F) for details.</p> <p>ddi_mapdev( ) sets up user mappings to device space. The driver is notified of user events on the mappings via the entry points defined by <i>ctl</i>.</p> <p>The user events that the driver is notified of are:</p> <table><tr><td>access</td><td>User has accessed an address in the mapping that has no translations.</td></tr></table>	access	User has accessed an address in the mapping that has no translations.																						
access	User has accessed an address in the mapping that has no translations.																								

**duplication** User has duplicated the mapping. Mappings are duplicated when the process calls `fork(2)`.

**unmapping** User has called `munmap(2)` on the mapping or is exiting.

See `mapdev_access(9E)`, `mapdev_dup(9E)`, and `mapdev_free(9E)` for details on these entry points.

The range to be mapped, defined by *offset* and *len* must be valid.

The arguments *dev*, *asp*, *addrp*, *len*, *prot*, *maxprot*, *flags*, and *cred* are provided by the `segmap(9E)` entry point and should not be modified. See `segmap(9E)` for a description of these arguments. Unlike `ddi_segmap(9F)`, the driver's `mmap(9E)` entry point is not called to verify the range to be mapped.

With the handle, device drivers can use `ddi_mapdev_intercept(9F)` and `ddi_mapdev_nointercept(9F)` to inform the system of whether or not they are interested in being notified when the user process accesses the mapping. By default, user accesses to newly created mappings will generate a call to the `mapdev_access()` entry point. The driver is always notified of duplications and unmaps.

The device may also use the handle to assign certain characteristics to the mapping. See `ddi_mapdev_set_device_acc_attr(9F)` for details.

The device driver can use these interfaces to implement a device context and control user accesses to the device space. `ddi_mapdev()` is typically called from the `segmap(9E)` entry point.

**RETURN VALUES** `ddi_mapdev()` returns zero on success and non-zero on failure. The return value from `ddi_mapdev()` should be used as the return value for the driver's `segmap()` entry point.

**CONTEXT** This routine can be called from user or kernel context only.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**SEE ALSO** `fork(2)`, `mmap(2)`, `munmap(2)`, `attributes(5)`, `mapdev_access(9E)`, `mapdev_dup(9E)`, `mapdev_free(9E)`, `mmap(9E)`, `segmap(9E)`, `devmap_setup(9F)`, `ddi_mapdev_intercept(9F)`, `ddi_mapdev_nointercept(9F)`, `ddi_mapdev_set_device_acc_attr(9F)`, `ddi_segmap(9F)`, `ddi_mapdev_ctl(9S)`

*Writing Device Drivers*

ddi\_mapdev(9F)

**NOTES** | Only mappings of type `MAP_PRIVATE` should be used with `ddi_mapdev()`.

## ddi\_mapdev\_intercept(9F)

<b>NAME</b>	ddi_mapdev_intercept, ddi_mapdev_nointercept – control driver notification of user accesses						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/sunddi.h&gt;  int ddi_mapdev_intercept(ddi_mapdev_handle_t handle, off_t offset,                         off_t len);  int ddi_mapdev_nointercept(ddi_mapdev_handle_t handle, off_t offset,                            off_t len);</pre>						
<b>INTERFACE LEVEL</b>	These interfaces are obsolete. Use devmap_load(9F) instead of ddi_mapdev_intercept(). Use devmap_unload(9F) instead of ddi_mapdev_nointercept().						
<b>PARAMETERS</b>	<table><tr><td><i>handle</i></td><td>An opaque pointer to a device mapping handle.</td></tr><tr><td><i>offset</i></td><td>An offset in bytes within device memory.</td></tr><tr><td><i>len</i></td><td>Length in bytes.</td></tr></table>	<i>handle</i>	An opaque pointer to a device mapping handle.	<i>offset</i>	An offset in bytes within device memory.	<i>len</i>	Length in bytes.
<i>handle</i>	An opaque pointer to a device mapping handle.						
<i>offset</i>	An offset in bytes within device memory.						
<i>len</i>	Length in bytes.						
<b>DESCRIPTION</b>	<p>Future releases of Solaris will provide these functions for binary and source compatibility. However, for increased functionality, use devmap_load(9F) or devmap_unload(9F) instead. See devmap_load(9F) and devmap_unload(9F) for details.</p> <p>The ddi_mapdev_intercept() and ddi_mapdev_nointercept() functions control whether or not user accesses to device mappings created by ddi_mapdev(9F) in the specified range will generate calls to the mapdev_access(9E) entry point. ddi_mapdev_intercept() tells the system to intercept the user access and notify the driver to invalidate the mapping translations. ddi_mapdev_nointercept() tells the system to not intercept the user access and allow it to proceed by validating the mapping translations.</p> <p>For both routines, the range to be affected is defined by the <i>offset</i> and <i>len</i> arguments. Requests affect the entire page containing the <i>offset</i> and all pages up to and including the page containing the last byte as indicated by <i>offset + len</i>.</p> <p>Supplying a value of 0 for the <i>len</i> argument affects all addresses from the <i>offset</i> to the end of the mapping. Supplying a value of 0 for the <i>offset</i> argument and a value of 0 for <i>len</i> argument affect all addresses in the mapping.</p> <p>To manage a device context, a device driver would call ddi_mapdev_intercept() on the context about to be switched out, switch contexts, and then call ddi_mapdev_nointercept() on the context switched in.</p>						
<b>RETURN VALUES</b>	ddi_mapdev_intercept() and ddi_mapdev_nointercept() return the following values:						
	<table><tr><td>0</td><td>Successful completion.</td></tr><tr><td>Non-zero</td><td>An error occurred.</td></tr></table>	0	Successful completion.	Non-zero	An error occurred.		
0	Successful completion.						
Non-zero	An error occurred.						

## ddi\_mapdev\_intercept(9F)

**EXAMPLES** **EXAMPLE 1** managing a device context that is one page in length

The following shows an example of managing a device context that is one page in length.

```
ddi_mapdev_handle_t cur_hdl;
static int
xxmapdev_access(ddi_mapdev_handle_t handle, void *devprivate,
                off_t offset)
{
    int err;
    /* enable access callbacks for the current mapping */
    if (cur_hdl != NULL) {
        if ((err = ddi_mapdev_intercept(cur_hdl, offset, 0)) != 0)
            return (err);
    }
    /* Switch device context - device dependent*/
    ...
    /* Make handle the new current mapping */
    cur_hdl = handle;
    /*
     * Disable callbacks and complete the access for the
     * mapping that generated this callback.
     */
    return (ddi_mapdev_nointercept(handle, offset, 0));
}
```

**CONTEXT** These routines can be called from user or kernel context only.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**SEE ALSO** `attributes(5)`, `mapdev_access(9E)`, `devmap_load(9F)`, `ddi_mapdev(9F)`

*Writing Device Drivers*

<b>NAME</b>	ddi_mapdev_set_device_acc_attr – set the device attributes for the mapping										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_mapdev_set_device_acc_attr(ddi_mapdev_handle_t     mapping_handle, off_t offset, off_t len, ddi_device_acc_attr_t     *accattrp, uint_t rnumber);</pre>										
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).										
<b>PARAMETERS</b>	<table border="0"> <tr> <td style="padding-right: 10px;"><i>mapping_handle</i></td> <td>A pointer to a device mapping handle.</td> </tr> <tr> <td style="padding-right: 10px;"><i>offset</i></td> <td>The offset within device memory to which the device access attributes structure applies.</td> </tr> <tr> <td style="padding-right: 10px;"><i>len</i></td> <td>Length (in bytes) of the memory to which the device access attributes structure applies.</td> </tr> <tr> <td style="padding-right: 10px;"><i>*accattrp</i></td> <td>Pointer to a ddi_device_acc_attr(9S) structure. Contains the device access attributes to be applied to this range of memory.</td> </tr> <tr> <td style="padding-right: 10px;"><i>rnumber</i></td> <td>Index number to the register address space set.</td> </tr> </table>	<i>mapping_handle</i>	A pointer to a device mapping handle.	<i>offset</i>	The offset within device memory to which the device access attributes structure applies.	<i>len</i>	Length (in bytes) of the memory to which the device access attributes structure applies.	<i>*accattrp</i>	Pointer to a ddi_device_acc_attr(9S) structure. Contains the device access attributes to be applied to this range of memory.	<i>rnumber</i>	Index number to the register address space set.
<i>mapping_handle</i>	A pointer to a device mapping handle.										
<i>offset</i>	The offset within device memory to which the device access attributes structure applies.										
<i>len</i>	Length (in bytes) of the memory to which the device access attributes structure applies.										
<i>*accattrp</i>	Pointer to a ddi_device_acc_attr(9S) structure. Contains the device access attributes to be applied to this range of memory.										
<i>rnumber</i>	Index number to the register address space set.										
<b>DESCRIPTION</b>	<p>Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use devmap(9E) instead. See devmap(9E) for details.</p> <p>The ddi_mapdev_set_device_acc_attr() function assigns device access attributes to a range of device memory in the register set given by <i>rnumber</i>.</p> <p><i>*accattrp</i> defines the device access attributes. See ddi_device_acc_attr(9S) for more details.</p> <p><i>mapping_handle</i> is a mapping handle returned from a call to ddi_mapdev(9F).</p> <p>The range to be affected is defined by the <i>offset</i> and <i>len</i> arguments. Requests affect the entire page containing the <i>offset</i> and all pages up to and including the page containing the last byte as indicated by <i>offset+len</i>. Supplying a value of 0 for the <i>len</i> argument affects all addresses from the <i>offset</i> to the end of the mapping. Supplying a value of 0 for the <i>offset</i> argument and a value of 0 for the <i>len</i> argument affect all addresses in the mapping.</p>										
<b>RETURN VALUES</b>	<p>The ddi_mapdev_set_device_acc_attr() function returns the following values:</p> <table border="0"> <tr> <td style="padding-right: 10px;">DDI_SUCCESS</td> <td>The attributes were successfully set.</td> </tr> <tr> <td style="padding-right: 10px;">DDI_FAILURE</td> <td>It is not possible to set these attributes for this mapping handle.</td> </tr> </table>	DDI_SUCCESS	The attributes were successfully set.	DDI_FAILURE	It is not possible to set these attributes for this mapping handle.						
DDI_SUCCESS	The attributes were successfully set.										
DDI_FAILURE	It is not possible to set these attributes for this mapping handle.										
<b>CONTEXT</b>	This routine can be called from user or kernel context only.										

ddi\_mapdev\_set\_device\_acc\_attr(9F)

**SEE ALSO** | segmap(9E), ddi\_mapdev(9F), ddi\_segmap\_setup(9F),  
ddi\_device\_acc\_attr(9S)

*Writing Device Drivers*

<b>NAME</b>	ddi_map_regs, ddi_unmap_regs – map or unmap registers										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>										
<b>INTERFACE LEVEL</b>	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().										
<b>PARAMETERS</b>											
<b>ddi_map_regs()</b>	<table border="0"> <tr> <td style="padding-right: 10px;"><i>dip</i></td> <td>Pointer to the device's dev_info structure.</td> </tr> <tr> <td style="padding-right: 10px;"><i>rnumber</i></td> <td>Register set number.</td> </tr> <tr> <td style="padding-right: 10px;"><i>kaddrp</i></td> <td>Pointer to the base kernel address of the mapped region (set on return).</td> </tr> <tr> <td style="padding-right: 10px;"><i>offset</i></td> <td>Offset into register space.</td> </tr> <tr> <td style="padding-right: 10px;"><i>len</i></td> <td>Length to be mapped.</td> </tr> </table>	<i>dip</i>	Pointer to the device's dev_info structure.	<i>rnumber</i>	Register set number.	<i>kaddrp</i>	Pointer to the base kernel address of the mapped region (set on return).	<i>offset</i>	Offset into register space.	<i>len</i>	Length to be mapped.
<i>dip</i>	Pointer to the device's dev_info structure.										
<i>rnumber</i>	Register set number.										
<i>kaddrp</i>	Pointer to the base kernel address of the mapped region (set on return).										
<i>offset</i>	Offset into register space.										
<i>len</i>	Length to be mapped.										
<b>ddi_unmap_regs()</b>	<table border="0"> <tr> <td style="padding-right: 10px;"><i>dip</i></td> <td>Pointer to the device's dev_info structure.</td> </tr> <tr> <td style="padding-right: 10px;"><i>rnumber</i></td> <td>Register set number.</td> </tr> <tr> <td style="padding-right: 10px;"><i>kaddrp</i></td> <td>Pointer to the base kernel address of the region to be unmapped.</td> </tr> <tr> <td style="padding-right: 10px;"><i>offset</i></td> <td>Offset into register space.</td> </tr> <tr> <td style="padding-right: 10px;"><i>len</i></td> <td>Length to be unmapped.</td> </tr> </table>	<i>dip</i>	Pointer to the device's dev_info structure.	<i>rnumber</i>	Register set number.	<i>kaddrp</i>	Pointer to the base kernel address of the region to be unmapped.	<i>offset</i>	Offset into register space.	<i>len</i>	Length to be unmapped.
<i>dip</i>	Pointer to the device's dev_info structure.										
<i>rnumber</i>	Register set number.										
<i>kaddrp</i>	Pointer to the base kernel address of the region to be unmapped.										
<i>offset</i>	Offset into register space.										
<i>len</i>	Length to be unmapped.										
<b>DESCRIPTION</b>	<p>ddi_map_regs() maps in the register set given by <i>rnumber</i>. 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 <i>kaddrp</i>. <i>offset</i> specifies an offset into the register space to start from and <i>len</i> indicates the size of the area to be mapped. If <i>len</i> is non-zero, it overrides the length given in the register set description. See the discussion of the <code>reg</code> property in <code>sbus(4)</code> and for more information on register set descriptions. If <i>len</i> and <i>offset</i> are 0, the entire space is mapped.</p> <p>ddi_unmap_regs() undoes mappings set up by ddi_map_regs(). This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to ddi_unmap_regs() must correspond to a previous call to ddi_map_regs()). Releasing portions of previous mappings is not allowed. <i>rnumber</i></p>										

## ddi\_map\_regs(9F)

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** `ddi_map_regs()` returns:

DDI\_SUCCESS on success.

**CONTEXT** These functions can be called from user or interrupt context.

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

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*

<b>NAME</b>	ddi_mem_alloc, ddi_mem_free – allocate and free sequentially accessed memory												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>												
<b>INTERFACE LEVEL</b>	These interfaces are obsolete. ddi_dma_mem_alloc(9F) and ddi_dma_mem_free(9F) should be used instead.												
<b>PARAMETERS</b>													
<b>ddi_mem_alloc()</b>	<table border="0"> <tr> <td style="vertical-align: top;"><i>dip</i></td> <td>A pointer to the device's dev_info structure.</td> </tr> <tr> <td style="vertical-align: top;"><i>limits</i></td> <td>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.</td> </tr> <tr> <td style="vertical-align: top;"><i>length</i></td> <td>The length in bytes of the desired allocation.</td> </tr> <tr> <td style="vertical-align: top;"><i>flags</i></td> <td>The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.</td> </tr> <tr> <td style="vertical-align: top;"><i>kaddrp</i></td> <td>On a successful return, *kaddrp points to the allocated memory.</td> </tr> <tr> <td style="vertical-align: top;"><i>real_length</i></td> <td>The length in bytes that was allocated. Alignment and padding requirements may cause ddi_mem_alloc() to allocate more memory than requested in <i>length</i>.</td> </tr> </table>	<i>dip</i>	A pointer to the device's dev_info structure.	<i>limits</i>	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.	<i>length</i>	The length in bytes of the desired allocation.	<i>flags</i>	The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.	<i>kaddrp</i>	On a successful return, *kaddrp points to the allocated memory.	<i>real_length</i>	The length in bytes that was allocated. Alignment and padding requirements may cause ddi_mem_alloc() to allocate more memory than requested in <i>length</i> .
<i>dip</i>	A pointer to the device's dev_info structure.												
<i>limits</i>	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.												
<i>length</i>	The length in bytes of the desired allocation.												
<i>flags</i>	The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.												
<i>kaddrp</i>	On a successful return, *kaddrp points to the allocated memory.												
<i>real_length</i>	The length in bytes that was allocated. Alignment and padding requirements may cause ddi_mem_alloc() to allocate more memory than requested in <i>length</i> .												
<b>ddi_mem_free()</b>	<table border="0"> <tr> <td style="vertical-align: top;"><i>kaddr</i></td> <td>The memory returned from a successful call to ddi_mem_alloc().</td> </tr> </table>	<i>kaddr</i>	The memory returned from a successful call to ddi_mem_alloc().										
<i>kaddr</i>	The memory returned from a successful call to ddi_mem_alloc().												
<b>DESCRIPTION</b>	<p>ddi_mem_alloc() allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as <i>streaming</i> access. The allocation will obey the alignment and padding constraints as specified by the <i>limits</i> argument and other limits imposed by the system.</p> <p>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.</p> <p>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.</p> <p>ddi_mem_free() frees up memory allocated by ddi_mem_alloc().</p>												
<b>RETURN VALUES</b>	ddi_mem_alloc() returns:												

## ddi\_mem\_alloc(9F)

DDI\_SUCCESS     Memory successfully allocated.

DDI\_FAILURE     Allocation failed.

**CONTEXT**     `ddi_mem_alloc()` can be called from user or interrupt context, except when *flags* is set to 1, in which case it can be called from user context only.

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

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*

ddi\_mem\_get8(9F)

**NAME** ddi\_mem\_get8, ddi\_mem\_get16, ddi\_mem\_get32, ddi\_mem\_get64, ddi\_mem\_getw, ddi\_mem\_getl, ddi\_mem\_getll, ddi\_mem\_getb – read data from mapped device in the memory space or allocated DMA memory

**SYNOPSIS**

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

uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t *
    dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

**INTERFACE LEVEL PARAMETERS**

*handle* The data access handle returned from setup calls, such as ddi\_regs\_map\_setup(9F).

*dev\_addr* Base device address.

**DESCRIPTION**

These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi\_mem\_get8(), ddi\_mem\_get16(), ddi\_mem\_get32(), and ddi\_mem\_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, *dev\_addr*, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

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

**SEE ALSO** ddi\_mem\_put8(9F), ddi\_mem\_rep\_get8(9F), ddi\_mem\_rep\_put8(9F), ddi\_regs\_map\_setup(9F), ddi\_device\_acc\_attr(9S)

**NOTES** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

Previous Name	New Name
ddi_mem_getb	ddi_mem_get8
ddi_mem_getw	ddi_mem_get16
ddi_mem_getl	ddi_mem_get32
ddi_mem_getll	ddi_mem_get64

## ddi\_mem\_put8(9F)

<b>NAME</b>	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						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>						
<b>PARAMETERS</b>	<table><tr><td><i>handle</i></td><td>The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</td></tr><tr><td><i>dev_addr</i></td><td>Base device address.</td></tr><tr><td><i>value</i></td><td>The data to be written to the device.</td></tr></table>	<i>handle</i>	The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).	<i>dev_addr</i>	Base device address.	<i>value</i>	The data to be written to the device.
<i>handle</i>	The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).						
<i>dev_addr</i>	Base device address.						
<i>value</i>	The data to be written to the device.						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>DESCRIPTION</b>	<p>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, <i>dev_addr</i>, in memory space.</p> <p>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.</p>						
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.						
<b>SEE ALSO</b>	ddi_mem_get8(9F), ddi_mem_rep_get8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)						
<b>NOTES</b>	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
ddi_mem_putw	ddi_mem_put16

ddi\_mem\_put8(9F)

Previous Name	New Name
ddi_mem_put1	ddi_mem_put32
ddi_mem_put11	ddi_mem_put64

## ddi\_mem\_rep\_get8(9F)

<b>NAME</b>	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										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,     uint8_t *dev_addr, size_t recount, uint_t flags);  void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,     uint16_t *dev_addr, size_t recount, uint_t flags);  void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,     uint32_t *dev_addr, size_t recount, uint_t flags);  void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,     uint64_t *dev_addr, size_t recount, uint_t flags);</pre>										
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <table><tr><td><i>handle</i></td><td>The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</td></tr><tr><td><i>host_addr</i></td><td>Base host address.</td></tr><tr><td><i>dev_addr</i></td><td>Base device address.</td></tr><tr><td><i>recount</i></td><td>Number of data accesses to perform.</td></tr><tr><td><i>flags</i></td><td>Device address flags:  DDI_DEV_AUTOINCR      Automatically increment the device address, <i>dev_addr</i>, during data accesses.  DDI_DEV_NO_AUTOINCR    Do not advance the device address, <i>dev_addr</i>, during data accesses.</td></tr></table>	<i>handle</i>	The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).	<i>host_addr</i>	Base host address.	<i>dev_addr</i>	Base device address.	<i>recount</i>	Number of data accesses to perform.	<i>flags</i>	Device address flags:  DDI_DEV_AUTOINCR      Automatically increment the device address, <i>dev_addr</i> , during data accesses.  DDI_DEV_NO_AUTOINCR    Do not advance the device address, <i>dev_addr</i> , during data accesses.
<i>handle</i>	The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).										
<i>host_addr</i>	Base host address.										
<i>dev_addr</i>	Base device address.										
<i>recount</i>	Number of data accesses to perform.										
<i>flags</i>	Device address flags:  DDI_DEV_AUTOINCR      Automatically increment the device address, <i>dev_addr</i> , during data accesses.  DDI_DEV_NO_AUTOINCR    Do not advance the device address, <i>dev_addr</i> , during data accesses.										
<b>DESCRIPTION</b>	<p>These routines generate multiple reads from memory space or allocated DMA memory. <i>recount</i> data is copied from the device address, <i>dev_addr</i>, in memory space to the host address, <i>host_addr</i>. 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, <i>dev_addr</i>. <i>dev_addr</i> and <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>										

## ddi\_mem\_rep\_get8(9F)

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, *dev\_addr*, as a memory buffer location on the device and increments its address on the next input datum. However, when the *flags* argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when 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
<code>ddi_mem_rep_getb</code>	<code>ddi_mem_rep_get8</code>
<code>ddi_mem_rep_getw</code>	<code>ddi_mem_rep_get16</code>
<code>ddi_mem_rep_getl</code>	<code>ddi_mem_rep_get32</code>
<code>ddi_mem_rep_getll</code>	<code>ddi_mem_rep_get64</code>

## ddi\_mem\_rep\_put8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <i>handle</i> The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).  <i>host_addr</i> Base host address.  <i>dev_addr</i> Base device address.  <i>repcount</i> Number of data accesses to perform.  <i>flags</i> Device address flags:  DDI_DEV_AUTOINCR Automatically increment the device address, <i>dev_addr</i> , during data accesses.  DDI_DEV_NO_AUTOINCR Do not advance the device address, <i>dev_addr</i> , during data accesses.
<b>DESCRIPTION</b>	<p>These routines generate multiple writes to memory space or allocated DMA memory. <i>repcount</i> data is copied from the host address, <i>host_addr</i>, to the device address, <i>dev_addr</i>, 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. <i>dev_addr</i> and <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>

## ddi\_mem\_rep\_put8(9F)

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, *dev\_addr*, as a memory buffer location on the device and increments its address on the next input datum. However, when the *flags* argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

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

**SEE ALSO** `ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_get8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

Previous Name	New Name
<code>ddi_mem_rep_putb</code>	<code>ddi_mem_rep_put8</code>
<code>ddi_mem_rep_putw</code>	<code>ddi_mem_rep_put16</code>
<code>ddi_mem_rep_putl</code>	<code>ddi_mem_rep_put32</code>
<code>ddi_mem_rep_putll</code>	<code>ddi_mem_rep_put64</code>

## ddi\_mmap\_get\_model(9F)

<b>NAME</b>	ddi_mmap_get_model – return data model type of current thread						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  uint_t ddi_mmap_get_model(void);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>DESCRIPTION</b>	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.						
<b>RETURN VALUES</b>	<table><tr><td>DDI_MODEL_ILP32</td><td>Current thread expects 32-bit (<i>ILP32</i>) semantics.</td></tr><tr><td>DDI_MODEL_LP64</td><td>Current thread expects 64-bit (<i>LP64</i>) semantics.</td></tr><tr><td>DDI_FAILURE</td><td>The ddi_mmap_get_model() function was not called from the mmap(9E) entry point.</td></tr></table>	DDI_MODEL_ILP32	Current thread expects 32-bit ( <i>ILP32</i> ) semantics.	DDI_MODEL_LP64	Current thread expects 64-bit ( <i>LP64</i> ) semantics.	DDI_FAILURE	The ddi_mmap_get_model() function was not called from the mmap(9E) entry point.
DDI_MODEL_ILP32	Current thread expects 32-bit ( <i>ILP32</i> ) semantics.						
DDI_MODEL_LP64	Current thread expects 64-bit ( <i>LP64</i> ) semantics.						
DDI_FAILURE	The ddi_mmap_get_model() function was not called from the mmap(9E) entry point.						
<b>CONTEXT</b>	The ddi_mmap_get_model() function can only be called from the mmap(9E) driver entry point.						
<b>EXAMPLES</b>	<p><b>EXAMPLE 1:</b> Using ddi_mmap_get_model()</p> <p>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.</p> <pre>struct data32 {     int len;     caddr32_t addr; };  struct data {     int len;     caddr_t addr; };  xxmmap(dev_t dev, off_t off, int prot) {     struct data dtc; /* a local copy for clash resolution */     struct data *dp = (struct data *)shared_area;      switch (ddi_model_convert_from(ddi_mmap_get_model())) {     case DDI_MODEL_ILP32:         {             struct data32 *da32p;              da32p = (struct data32 *)shared_area;             dp = &amp;dtc;             dp-&gt;len = da32p-&gt;len;             dp-&gt;address = da32p-&gt;address;             break;         }     }</pre>						

**EXAMPLE 1:** Using `ddi_mmap_get_model()` (Continued)

```
        case DDI_MODEL_NONE:
            break;
    }
    /* continues along using dp */
    ...
}
```

**SEE ALSO** `mmap(9E)`, `ddi_model_convert_from(9F)`

*Writing Device Drivers*

## ddi\_model\_convert\_from(9F)

<b>NAME</b>	ddi_model_convert_from – determine data model type mismatch
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  uint_t ddi_model_convert_from(uint_t model);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>model</i> The data model type of the current thread.
<b>DESCRIPTION</b>	ddi_model_convert_from() is used to determine if the current thread uses a different C Language Type Model than the device driver. The 64-bit version of Solaris will require a 64-bit kernel to support both 64-bit and 32-bit user mode programs. The difference between a 32-bit program and a 64-bit program is in its C Language Type Model: a 32-bit program is ILP32 (integer, longs, and pointers are 32-bit) and a 64-bit program is LP64 (longs and pointers are 64-bit). There are a number of driver entry points such as ioctl(9E) and mmap(9E) where it is necessary to identify the C Language Type Model of the user-mode originator of a kernel event. For example any data which flows between programs and the device driver or vice versa need to be identical in format. A 64-bit device driver may need to modify the format of the data before sending it to a 32-bit application. ddi_model_convert_from() is used to determine if data that is passed between the device driver and the application requires reformatting to any non-native data model.
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	ddi_model_convert_from() can be called from any context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1:</b> Using ddi_model_convert_from() in the ioctl() entry point to support both 32-bit and 64-bit applications.</p> <p>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.</p> <pre>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;      switch (ddi_model_convert_from(mode &amp; FMODELS)) {         case DDI_MODEL_ILP32:             {</pre>

**EXAMPLE 1:** Using `ddi_model_convert_from()` in the `ioctl()` entry point to support both 32-bit and 64-bit applications. (*Continued*)

```

        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*

## ddi\_node\_name(9F)

<b>NAME</b>	ddi_node_name – return the devinfo node name
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  char *<b>ddi_node_name</b>(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> A pointer the device's dev_info structure.
<b>DESCRIPTION</b>	ddi_node_name() returns the device node name contained in the dev_info node pointed to by <i>dip</i> .
<b>RETURN VALUES</b>	ddi_node_name() returns the device node name contained in the dev_info structure.
<b>CONTEXT</b>	ddi_node_name() can be called from user or interrupt context.
<b>SEE ALSO</b>	ddi_binding_name(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_no_info – stub for getinfo(9E)
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_no_info</b>(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,                 void **result);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI)</p> <p><i>dev_info_t *dip</i> Pointer to dev_info structure.</p> <p><i>ddi_info_cmd_t infocmd</i> Command argument. Valid command values are: DDI_INFO_DEVT2DEVINFO and DDI_INFO_DEVT2INSTANCE.</p> <p><i>void *arg</i> Command-specific argument.</p> <p><i>void **result</i> Pointer to where the requested information is stored.</p>
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	The ddi_no_info() function always returns DDI_FAILURE.
<b>SEE ALSO</b>	getinfo(9E), qassociate(9F), cb_ops(9S), dev_ops(9S)

## ddi\_peek(9F)

<b>NAME</b>	<code>ddi_peek</code> , <code>ddi_peek8</code> , <code>ddi_peek16</code> , <code>ddi_peek32</code> , <code>ddi_peek64</code> , <code>ddi_peekc</code> , <code>ddi_peeks</code> , <code>ddi_peekl</code> , <code>ddi_peekd</code> – read a value from a location
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI). The <code>ddi_peekc()</code> , <code>ddi_peeks()</code> , <code>ddi_peekl()</code> , and <code>ddi_peekd()</code> functions are obsolete. Use, respectively, <code>ddi_peek8()</code> , <code>ddi_peek16()</code> , <code>ddi_peek32()</code> , and <code>ddi_peek64()</code> , instead.
<b>PARAMETERS</b>	<p><i>dip</i>            A pointer to the device's <code>dev_info</code> structure.</p> <p><i>addr</i>            Virtual address of the location to be examined.</p> <p><i>valuep</i>        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.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.</p> <p>The routines are most useful when first trying to establish the presence of a device on the system in a driver's <code>probe(9E)</code> or <code>attach(9E)</code> routines.</p>
<b>RETURN VALUES</b>	<p><code>DDI_SUCCESS</code>    The value at the given virtual address was successfully read, and if <i>valuep</i> is non-null, <i>valuep</i> will have been updated.</p> <p><code>DDI_FAILURE</code>    An error occurred while trying to read the location. <i>valuep</i> is unchanged.</p>
<b>CONTEXT</b>	These functions can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Checking to see that the status register of a device is mapped into the kernel address space:</p> <pre>if (ddi_peek8(dip, csr, (int8_t *)0) != DDI_SUCCESS) {     cmn_err(CE_WARN, "Status register not mapped");     return (DDI_FAILURE); }</pre>

**EXAMPLE 1** Checking to see that the status register of a device is mapped into the kernel address space: *(Continued)*

**EXAMPLE 2** Reading and logging the device type of a particular device:

```
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

## ddi\_poke(9F)

<b>NAME</b>	ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>				
<b>INTERFACE LEVEL</b>	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.				
<b>PARAMETERS</b>	<p><i>dip</i>                      A pointer to the device's dev_info structure.</p> <p><i>addr</i>                     Virtual address of the location to be written to.</p> <p><i>value</i>                    Value to be written to the location.</p>				
<b>DESCRIPTION</b>	<p>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.</p> <p>If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.</p> <p>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.</p> <p>On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.</p>				
<b>RETURN VALUES</b>	<p>DDI_SUCCESS      The value was successfully written to the given virtual address.</p> <p>DDI_FAILURE      An error occurred while trying to write to the location.</p>				
<b>CONTEXT</b>	These functions can be called from user or interrupt context.				
<b>SEE ALSO</b>	attach(9E), probe(9E), ddi_peek(9F)				
	<i>Writing Device Drivers</i>				
<b>NOTES</b>	The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:				
	<table border="1"><thead><tr><th>Previous Name</th><th>New Name</th></tr></thead><tbody><tr><td>ddi_pokec</td><td>ddi_poke8</td></tr></tbody></table>	Previous Name	New Name	ddi_pokec	ddi_poke8
Previous Name	New Name				
ddi_pokec	ddi_poke8				

ddi\_poke(9F)

ddi_pokes	ddi_poke16
ddi_poke1	ddi_poke32
ddi_poked	ddi_poke64

## ddi\_prop\_create(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	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.
<b>PARAMETERS</b>	<p>ddi_prop_create()</p> <p><i>dev</i> dev_t of the device.</p> <p><i>dip</i> dev_info_t pointer of the device.</p> <p><i>flags</i> flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.</p> <p><i>name</i> name of property.</p> <p><i>valuep</i> pointer to property value.</p> <p><i>length</i> property length.</p> <p>ddi_prop_undefine()</p> <p><i>dev</i> dev_t of the device.</p> <p><i>dip</i> dev_info_t pointer of the device.</p> <p><i>flags</i> flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.</p> <p><i>name</i> name of property.</p> <p>ddi_prop_modify()</p> <p><i>dev</i> dev_t of the device.</p> <p><i>dip</i> dev_info_t pointer of the device.</p>

<i>flags</i>	flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.
<i>name</i>	name of property.
<i>valuep</i>	pointer to property value.
<i>length</i>	property length.
ddi_prop_remove()	
<i>dev</i>	dev_t of the device.
<i>dip</i>	dev_info_t pointer of the device.
<i>name</i>	name of property.
ddi_prop_remove_all()	
<i>dip</i>	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\_create(9F)

### ddi\_prop\_undefine()

ddi\_prop\_undefine() is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. However, ddi\_prop\_undefine() will not terminate a search when the ddi\_prop\_get\_int64(9F) or ddi\_prop\_lookup\_int64\_array(9F) routines are used for lookup of 64-bit property value. See ddi\_prop\_op(9F).

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi\_prop\_create().

### ddi\_prop\_modify()

ddi\_prop\_modify() modifies the length and the value of a property. If ddi\_prop\_modify() finds the property in the driver's property list, allocates memory for the property value and returns DDI\_PROP\_SUCCESS. If the property was not found, the function returns DDI\_PROP\_NOT\_FOUND.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as ddi\_prop\_create().

### ddi\_prop\_remove()

ddi\_prop\_remove() unlinks a property from the device's property list. If ddi\_prop\_remove() finds the property (an exact match of both *name* and *dev*), it unlinks the property, frees its memory, and returns DDI\_PROP\_SUCCESS, otherwise, it returns DDI\_PROP\_NOT\_FOUND.

### ddi\_prop\_remove\_all()

ddi\_prop\_remove\_all() removes the properties of all the dev\_t's associated with the *dip*. It is called before unloading a driver.

## RETURN VALUES

<b>ddi_prop_create()</b>	DDI_PROP_SUCCESS	On success.
	DDI_PROP_NO_MEMORY	On memory allocation failure.
	DDI_PROP_INVALID_ARG	If an attempt is made to create a property with <i>dev</i> equal to DDI_DEV_T_ANY or if <i>name</i> is NULL or <i>name</i> is the NULL string.
<b>ddi_prop_undefine()</b>	DDI_PROP_SUCCESS	On success.
	DDI_PROP_NO_MEMORY	On memory allocation failure.
	DDI_PROP_INVALID_ARG	If an attempt is made to create a property with <i>dev</i> DDI_DEV_T_ANY or if <i>name</i> is NULL or <i>name</i> is the NULL string.
<b>ddi_prop_modify()</b>	DDI_PROP_SUCCESS	On success.
	DDI_PROP_NO_MEMORY	On memory allocation failure.

ddi\_prop\_create(9F)

<b>ddi_prop_remove()</b>	DDI_PROP_INVALID_ARG	If an attempt is made to create a property with <i>dev</i> equal to DDI_DEV_T_ANY or if <i>name</i> is NULL or <i>name</i> is the NULL string.
	DDI_PROP_NOT_FOUND	On property search failure.
	DDI_PROP_SUCCESS	On success.
	DDI_PROP_INVALID_ARG	If an attempt is made to create a property with <i>dev</i> equal to DDI_DEV_T_ANY or if <i>name</i> is NULL or <i>name</i> is the NULL string.
	DDI_PROP_NOT_FOUND	On property search failure.

**CONTEXT** If DDI\_PROP\_CANSLEEP is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

**EXAMPLES** **EXAMPLE 1** Creating a Property

The following example creates a property called *nblocks* for each partition on a disk.

```
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\_getpropflen(9F), ddi\_prop\_op(9F), ddi\_prop\_update(9F), makedevice(9F)

*Writing Device Drivers*

## ddi\_prop\_exists(9F)

<b>NAME</b>	ddi_prop_exists – check for the existence of a property						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_prop_exists(dev_t match_dev, dev_info_t *dip, uint_t flags,     char *name);</pre>						
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <table><tr><td><i>match_dev</i></td><td>Device number associated with property or DDI_DEV_T_ANY.</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:  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).  <i>name</i> String containing the name of the property.</td></tr></table>	<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.	<i>dip</i>	Pointer to the device info node of device whose property list should be searched.	<i>flags</i>	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).  <i>name</i> String containing the name of the property.
<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.						
<i>dip</i>	Pointer to the device info node of device whose property list should be searched.						
<i>flags</i>	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).  <i>name</i> String containing the name of the property.						
<b>DESCRIPTION</b>	<p>ddi_prop_exists() checks for the existence of a property regardless of the property value data type.</p> <p>Properties are searched for based on the <i>dip</i>, <i>name</i>, and <i>match_dev</i>. The property search order is as follows:</p> <ol style="list-style-type: none"><li>1. Search software properties created by the driver.</li><li>2. Search the software properties created by the system (or nexus nodes in the device info tree).</li><li>3. Search the driver global properties list.</li><li>4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).</li><li>5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.</li><li>6. Return 0 if not found and 1 if found.</li></ol> <p>Usually, the <i>match_dev</i> argument should be set to the actual device number that this property is associated with. However, if the <i>match_dev</i> argument is DDI_DEV_T_ANY, then ddi_prop_exists() will match the request regardless of the <i>match_dev</i> the property was created with. That is the first property whose name matches <i>name</i> will be returned. If a property was created with <i>match_dev</i> set to DDI_DEV_T_NONE then the only way to look up this property is with a <i>match_dev</i> set to DDI_DEV_T_ANY. PROM properties are always created with <i>match_dev</i> set to DDI_DEV_T_NONE.</p> <p><i>name</i> must always be set to the name of the property being looked up.</p>						

**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*

## ddi\_prop\_get\_int(9F)

<b>NAME</b>	ddi_prop_get_int, ddi_prop_get_int64 – lookup integer property										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>										
<b>PARAMETERS</b>	<table><tr><td><i>match_dev</i></td><td>Device number associated with property or DDI_DEV_T_ANY.</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:  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).</td></tr><tr><td><i>name</i></td><td>String containing the name of the property.</td></tr><tr><td><i>defvalue</i></td><td>An integer value that is returned if the property cannot be found.</td></tr></table>	<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.	<i>dip</i>	Pointer to the device info node of device whose property list should be searched.	<i>flags</i>	Possible flag values are some combination of:  DDI_PROP_DONTPASS Do not pass request to parent device information node if property not found.  DDI_PROP_NOTPROM Do not look at PROM properties (ignored on platforms that do not support PROM properties).	<i>name</i>	String containing the name of the property.	<i>defvalue</i>	An integer value that is returned if the property cannot be found.
<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.										
<i>dip</i>	Pointer to the device info node of device whose property list should be searched.										
<i>flags</i>	Possible flag values are some combination of:  DDI_PROP_DONTPASS Do not pass request to parent device information node if property not found.  DDI_PROP_NOTPROM Do not look at PROM properties (ignored on platforms that do not support PROM properties).										
<i>name</i>	String containing the name of the property.										
<i>defvalue</i>	An integer value that is returned if the property cannot be found.										
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).										
<b>DESCRIPTION</b>	<p>The <code>ddi_prop_get_int()</code> and <code>ddi_prop_get_int64()</code> functions search for an integer property and, if found, returns the value of the property.</p> <p>Properties are searched for based on the <i>dip</i>, <i>name</i>, <i>match_dev</i>, and the type of the data (integer). The property search order is as follows:</p> <ol style="list-style-type: none"><li>1. Search software properties created by the driver.</li><li>2. Search the software properties created by the system (or nexus nodes in the device info tree).</li><li>3. Search the driver global properties list.</li><li>4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).</li><li>5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.</li><li>6. Return <i>defvalue</i>.</li></ol> <p>Usually, the <i>match_dev</i> argument should be set to the actual device number that this property is associated with. However, if the <i>match_dev</i> argument is DDI_DEV_T_ANY, then <code>ddi_prop_get_int()</code> and <code>ddi_prop_get_int64()</code> will match the request</p>										

## ddi\_prop\_get\_int(9F)

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*

## ddi\_prop\_lookup(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>PARAMETERS</b>	<p><i>match_dev</i> Device number associated with property or DDI_DEV_T_ANY.</p> <p><i>dip</i> Pointer to the device info node of device whose property list should be searched.</p> <p><i>flags</i> Possible flag values are some combination of:</p> <ul style="list-style-type: none"><li>DDI_PROP_DONTPASS Do not pass request to parent device information node if the property is not found.</li><li>DDI_PROP_NOTPROM Do not look at PROM properties (ignored on platforms that do not support PROM properties).</li></ul> <p><i>name</i> String containing the name of the property.</p> <p><i>nelementsp</i> 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.</p> <p><i>datap</i></p> <p>ddi_prop_lookup_int_array() The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.</p>

## ddi\_prop\_lookup(9F)

`ddi_prop_lookup_int64_array()`

The address of a pointer to an array of 64-bit integers which, upon successful return, will point to memory containing the integer array property value.

`ddi_prop_lookup_string_array()`

The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to `execve(2)`.

`ddi_prop_lookup_string()`

The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.

`ddi_prop_lookup_byte_array()`

The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

### INTERFACE LEVEL DESCRIPTION

Solaris DDI specific (Solaris DDI).

The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the *dip*, *name*, *match\_dev*, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If `DDI_PROP_NOTPROM` is not set, search the PROM properties (if they exist).
5. If `DDI_PROP_DONTPASS` is not set, pass this request to the parent device information node.
6. Return `DDI_PROP_NOT_FOUND`.

Usually, the *match\_dev* argument should be set to the actual device number that this property is associated with. However, if the *match\_dev* argument is `DDI_DEV_T_ANY`, the property look up routines will match the request regardless of the actual *match\_dev* the property was created with. If a property was created with *match\_dev* set to `DDI_DEV_T_NONE`, then the only way to look up this property is with a *match\_dev* set to `DDI_DEV_T_ANY`. PROM properties are always created with *match\_dev* set to `DDI_DEV_T_NONE`.

*name* must always be set to the name of the property being looked up.

## ddi\_prop\_lookup(9F)

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:

<code>DDI_PROP_SUCCESS</code>	Upon success.
<code>DDI_PROP_INVALID_ARG</code>	If an attempt is made to look up a property with <i>match_dev</i> equal to <code>DDI_DEV_T_NONE</code> , <i>name</i> is <code>NULL</code> or <i>name</i> is the null string.
<code>DDI_PROP_NOT_FOUND</code>	Property not found.
<code>DDI_PROP_UNDEFINED</code>	Property explicitly not defined (see <code>ddi_prop_undefine(9F)</code> ).
<code>DDI_PROP_CANNOT_DECODE</code>	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()`.

```
int    *options;
int    noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
 */
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
    "options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
     * Do "our thing" with the options data from the property
     */
    xx_process_options(options, noptions);

    /*
     * Free the memory allocated for the property data
     */
    ddi_prop_free(options);
}
```

**SEE ALSO**

`execve(2)`, `ddi_prop_exists(9F)`, `ddi_prop_get_int(9F)`, `ddi_prop_remove(9F)`, `ddi_prop_undefine(9F)`, `ddi_prop_update(9F)`

ddi\_prop\_lookup(9F)

*Writing Device Drivers*

<b>NAME</b>	ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen – get property information for leaf device drivers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	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().
<b>PARAMETERS</b>	<p><i>dev</i> Device number associated with property or DDI_DEV_T_ANY as the <i>wildcard</i> device number.</p> <p><i>dip</i> Pointer to a device info node.</p> <p><i>prop_op</i> Property operator.</p> <p><i>flags</i> Possible flag values are some combination of:</p> <ul style="list-style-type: none"> <li>DDI_PROP_DONTPASS do not pass request to parent device information node if property not found</li> <li>DDI_PROP_CANSLEEP the routine may sleep while allocating memory</li> <li>DDI_PROP_NOTPROM do not look at PROM properties (ignored on architectures that do not support PROM properties)</li> </ul> <p><i>name</i> String containing the name of the property.</p> <p><i>valuep</i> If <i>prop_op</i> is PROP_LEN_AND_VAL_BUF, this should be a pointer to the users buffer. If <i>prop_op</i> is PROP_LEN_AND_VAL_ALLOC, this should be the <i>address</i> of a pointer.</p>

## ddi\_prop\_op(9F)

*lengthp* On exit, *\*lengthp* will contain the property length. If *prop\_op* is `PROP_LEN_AND_VAL_BUF` then before calling `ddi_prop_op()`, *lengthp* should point to an `int` that contains the length of callers buffer.

*defvalue* The value that `ddi_getprop()` returns if the property is not found.

### DESCRIPTION

`ddi_prop_op()` gets arbitrary-size properties for leaf devices. The routine searches the device's property list. If it does not find the property at the device level, it examines the *flags* argument, and if `DDI_PROP_DONTPASS` is set, then `ddi_prop_op()` returns `DDI_PROP_NOT_FOUND`. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns `DDI_PROP_UNDEFINED`. Otherwise it returns either the property length, or both the length and value of the property to the caller via the *valuep* and *lengthp* pointers, depending on the value of *prop\_op*, as described below, and returns `DDI_PROP_SUCCESS`. If a property cannot be found at all, `DDI_PROP_NOT_FOUND` is returned.

Usually, the *dev* argument should be set to the actual device number that this property applies to. However, if the *dev* argument is `DDI_DEV_T_ANY`, the *wildcard dev*, then `ddi_prop_op()` will match the request based on *name* only (regardless of the actual *dev* the property was created with). This property/*dev* match is done according to the property search order which is to first search software properties created by the driver in *last-in, first-out* (LIFO) order, next search software properties created by the *system* in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the *prop\_op* argument. If *prop\_op* is `PROP_LEN`, then `ddi_prop_op()` just sets the callers length, *\*lengthp*, to the property length and returns the value `DDI_PROP_SUCCESS` to the caller. The *valuep* argument is not used in this case. Property lengths are 0 for boolean properties, `sizeof(int)` for integer properties, and size in bytes for long (variable size) properties.

If *prop\_op* is `PROP_LEN_AND_VAL_BUF`, then *valuep* should be a pointer to a user-supplied buffer whose length should be given in *\*lengthp* by the caller. If the requested property exists, `ddi_prop_op()` first sets *\*lengthp* to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns `DDI_PROP_SUCCESS`. If the named property exists but the buffer supplied is too small to hold it, it returns `DDI_PROP_BUF_TOO_SMALL`.

If *prop\_op* is `PROP_LEN_AND_VAL_ALLOC`, and the property is found, `ddi_prop_op()` sets *\*lengthp* to the property length. It then attempts to allocate a buffer to return to the caller using the `kmem_alloc(9F)` routine, so that memory can be later recycled using `kmem_free(9F)`. The driver is expected to call `kmem_free()` with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets *\*valuep* to point to the allocated buffer, copies the property value into the buffer and returns `DDI_PROP_SUCCESS`. Otherwise, it returns

## ddi\_prop\_op(9F)

DDI\_PROP\_NO\_MEMORY. Note that the *flags* argument may affect the behavior of memory allocation in `ddi_prop_op()`. In particular, if DDI\_PROP\_CANSLEEP is set, then the routine will wait until memory is available to copy the requested property.

`ddi_getprop()` returns boolean and integer-size properties. It is a convenience wrapper for `ddi_prop_op()` with *prop\_op* set to PROP\_LEN\_AND\_VAL\_BUF, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

`ddi_getlongprop()` returns arbitrary-size properties. It is a convenience wrapper for `ddi_prop_op()` with *prop\_op* set to PROP\_LEN\_AND\_VAL\_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *\*valuep*.

`ddi_getlongprop_buf()` returns arbitrary-size properties. It is a convenience wrapper for `ddi_prop_op()` with *prop\_op* set to PROP\_LEN\_AND\_VAL\_BUF so the user must supply a buffer.

`ddi_getproplen()` returns the length of a given property. It is a convenience wrapper for `ddi_prop_op()` with *prop\_op* set to PROP\_LEN.

### RETURN VALUES

`ddi_prop_op()` `ddi_getlongprop()` `ddi_getlongprop_buf()`  
`ddi_getproplen()` return:

DDI_PROP_SUCCESS	Property found and returned.
DDI_PROP_NOT_FOUND	Property not found.
DDI_PROP_UNDEFINED	Property already explicitly undefined.
DDI_PROP_NO_MEMORY	Property found, but unable to allocate memory. <i>lengthp</i> points to the correct property length.
DDI_PROP_BUF_TOO_SMALL	Property found, but the supplied buffer is too small. <i>lengthp</i> points to the correct property length.

`ddi_getprop()` returns:

The value of the property or the value passed into the routine as *defvalue* if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

### CONTEXT

These functions can be called from user or interrupt context, provided DDI\_PROP\_CANSLEEP is not set; if it is set, they can be called from user context only.

### ATTRIBUTES

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

## ddi\_prop\_op(9F)

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*

ddi\_prop\_update(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>PARAMETERS</b>	<p><i>dev</i> Device number associated with the device.</p> <p><i>dip</i> Pointer to the device info node of device whose property list should be updated.</p> <p><i>name</i> String containing the name of the property to be updated.</p> <p><i>nelements</i> The number of elements contained in the memory pointed at by <i>data</i>.</p> <p>ddi_prop_update_int_array()  <i>data</i> A pointer an integer array with which to update the property.</p> <p>ddi_prop_update_int()  <i>data</i> An integer value with which to update the property.</p> <p>ddi_prop_update_int64_array()  <i>data</i> An pointer to a 64-bit integer array with which to update the property.</p> <p>ddi_prop_update_int64()  <i>data</i> A 64-bit integer value with which to update the property.</p> <p>ddi_prop_update_string_array()</p>

## ddi\_prop\_update(9F)

*data* A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to `execve(2)`.

`ddi_prop_update_string()`

*data* A pointer to a string value with which to update the property.

`ddi_prop_update_byte_array()`

*data* A pointer to a byte array with which to update the property.

### INTERFACE LEVEL DESCRIPTION

Solaris DDI specific (Solaris DDI).

The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the *dip*, *name*, *dev*, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either `ddi_prop_update_int_array()` or `ddi_prop_update_int()`. For a 64-bit integer, you must use `ddi_prop_update_int64_array()` or `ddi_prop_update_int64()`. Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the *dev* argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular *dev*, then the argument *dev* should be set to `DDI_DEV_T_NONE`. This property will then match a look up request (see `ddi_prop_lookup(9F)`) with the *match\_dev* argument set to `DDI_DEV_T_ANY`. If no *dev* is available for the device (for example during `attach(9E)` time), one can be created using `makedevice(9F)` with a major number of `DDI_MAJOR_T_UNKNOWN`. The update routines will then generate the correct *dev* when creating or updating the property.

*name* must always be set to the name of the property being updated.

For the routines `ddi_prop_update_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_update_string_array()`, `ddi_prop_update_string()`, and `ddi_prop_update_byte_array()`, *data* is a pointer which points to memory containing the value of the property. In each case

## ddi\_prop\_update(9F)

*\*data* points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. *nelements* is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *\*data*.

For the routines `ddi_prop_update_int()` and `ddi_prop_update_int64()`, *data* is the new value of the property.

`ddi_prop_update_int_array()`

Updates or creates an array of integer property values. An array of integers is defined to be *nelements* of 4 byte long integer elements. *data* must be a pointer to an integer array with which to update the property.

`ddi_prop_update_int()`

Update or creates a single integer value of a property. *data* must be an integer value with which to update the property.

`ddi_prop_update_int64_array()`

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be *nelements* of `int64_t` integer elements. *data* must be a pointer to a 64-bit integer array with which to update the property.

`ddi_prop_update_int64()`

Updates or creates a single 64-bit integer value of a property. *data* must be an `int64_t` value with which to update the property.

`ddi_prop_update_string_array()`

Updates or creates a property that is an array of strings. *data* must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULLterminated strings, much like the *argv* argument to `execve(2)`.

`ddi_prop_update_string()`

Updates or creates a property that is a single string value. *data* must be a pointer to a string with which to update the property.

`ddi_prop_update_byte_array()`

Updates or creates a property that is an array of bytes. *data* should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.

**RETURN VALUES** All of the property update routines return:

## ddi\_prop\_update(9F)

DDI_PROP_SUCCESS	On success.
DDI_PROP_INVALID_ARG	If an attempt is made to update a property with <i>name</i> set to NULL or <i>name</i> 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*

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_put8</b>(ddi_acc_handle_t <i>handle</i>, uint8_t *<i>dev_addr</i>, uint8_t     <i>value</i>);  void <b>ddi_put16</b>(ddi_acc_handle_t <i>handle</i>, uint16_t *<i>dev_addr</i>, uint16_t     <i>value</i>);  void <b>ddi_put32</b>(ddi_acc_handle_t <i>handle</i>, uint32_t *<i>dev_addr</i>, uint32_t     <i>value</i>);  void <b>ddi_put64</b>(ddi_acc_handle_t <i>handle</i>, uint64_t *<i>dev_addr</i>, uint64_t     <i>value</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</p> <p><i>value</i>             The data to be written to the device.</p> <p><i>dev_addr</i>          Base device address.</p>
<b>DESCRIPTION</b>	<p>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, <i>dev_addr</i>.</p> <p>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.</p> <p>For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA, EISA, and SBus buses. See sysbus(4), isa(4), eisa(4), and sbus(4) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See pci(4).</p>
<b>CONTEXT</b>	These functions can be called from user, kernel, or interrupt context.
<b>SEE ALSO</b>	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)
<b>NOTES</b>	The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

## ddi\_put8(9F)

<b>Previous Name</b>	<b>New Name</b>
ddi_putb	ddi_put8
ddi_putw	ddi_put16
ddi_putl	ddi_put32
ddi_putll	ddi_put64

ddi\_regs\_map\_free(9F)

**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, EISA

**SEE ALSO** attributes(5), ddi\_regs\_map\_setup(9F)  
*Writing Device Drivers*

## ddi\_regs\_map\_setup(9F)

<b>NAME</b>	ddi_regs_map_setup – set up a mapping for a register address space														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <table><tr><td><i>dip</i></td><td>Pointer to the device's dev_info structure.</td></tr><tr><td><i>rnumber</i></td><td>Index number to the register address space set.</td></tr><tr><td><i>addrp</i></td><td>A platform-dependent value that, when added to an offset that is less than or equal to the <i>len</i> parameter (see below), is used for the dev_addr argument to the ddi_get, ddi_mem_get, and ddi_io_get/put routines.</td></tr><tr><td><i>offset</i></td><td>Offset into the register address space.</td></tr><tr><td><i>len</i></td><td>Length to be mapped.</td></tr><tr><td><i>accattrp</i></td><td>Pointer to a device access attribute structure of this mapping (see ddi_device_acc_attr(9S)).</td></tr><tr><td><i>handlep</i></td><td>Pointer to a data access handle.</td></tr></table>	<i>dip</i>	Pointer to the device's dev_info structure.	<i>rnumber</i>	Index number to the register address space set.	<i>addrp</i>	A platform-dependent value that, when added to an offset that is less than or equal to the <i>len</i> parameter (see below), is used for the dev_addr argument to the ddi_get, ddi_mem_get, and ddi_io_get/put routines.	<i>offset</i>	Offset into the register address space.	<i>len</i>	Length to be mapped.	<i>accattrp</i>	Pointer to a device access attribute structure of this mapping (see ddi_device_acc_attr(9S)).	<i>handlep</i>	Pointer to a data access handle.
<i>dip</i>	Pointer to the device's dev_info structure.														
<i>rnumber</i>	Index number to the register address space set.														
<i>addrp</i>	A platform-dependent value that, when added to an offset that is less than or equal to the <i>len</i> parameter (see below), is used for the dev_addr argument to the ddi_get, ddi_mem_get, and ddi_io_get/put routines.														
<i>offset</i>	Offset into the register address space.														
<i>len</i>	Length to be mapped.														
<i>accattrp</i>	Pointer to a device access attribute structure of this mapping (see ddi_device_acc_attr(9S)).														
<i>handlep</i>	Pointer to a data access handle.														
<b>DESCRIPTION</b>	<p>ddi_regs_map_setup() maps in the register set given by <i>rnumber</i>. The register number determines which register set is mapped if more than one exists.</p> <p><i>offset</i> specifies the starting location within the register space and <i>len</i> indicates the size of the area to be mapped. If <i>len</i> is non-zero, it overrides the length given in the register set description. If both <i>len</i> and <i>offset</i> are 0, the entire space is mapped. The base of the mapped register space is returned in <i>addrp</i>.</p> <p>The device access attributes are specified in the location pointed by the <i>accattrp</i> argument (see ddi_device_acc_attr(9S) for details).</p> <p>The data access handle is returned in <i>handlep</i>. <i>handlep</i> 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.</p>														
<b>RETURN VALUES</b>	ddi_regs_map_setup() returns:  <table><tr><td>DDI_SUCCESS</td><td>Successfully set up the mapping for data access.</td></tr><tr><td>DDI_FAILURE</td><td>Invalid register number <i>rnumber</i>, offset <i>offset</i>, or length <i>len</i>.</td></tr></table>	DDI_SUCCESS	Successfully set up the mapping for data access.	DDI_FAILURE	Invalid register number <i>rnumber</i> , offset <i>offset</i> , or length <i>len</i> .										
DDI_SUCCESS	Successfully set up the mapping for data access.														
DDI_FAILURE	Invalid register number <i>rnumber</i> , offset <i>offset</i> , or length <i>len</i> .														

ddi\_regs\_map\_setup(9F)

DDI_ME_RNUMBER_RANGE	Invalid register number <i>rnumber</i> or unable to find <i>reg</i> 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, EISA

**SEE ALSO** attributes(5), ddi\_regs\_map\_free(9F), ddi\_device\_acc\_attr(9S)

*Writing Device Drivers*

## ddi\_remove\_minor\_node(9F)

<b>NAME</b>	ddi_remove_minor_node – remove a minor node for this dev_info
<b>SYNOPSIS</b>	<pre>void ddi_remove_minor_node(dev_info_t *dip, char *name);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>dip</i>        A pointer to the device's dev_info structure.</p> <p><i>name</i>       The name of this minor device. If <i>name</i> is NULL, then remove all minor data structures from this dev_info.</p>
<b>DESCRIPTION</b>	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.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Removing a minor node</p> <p>This will remove a data structure describing a minor device called dev1 which is linked into the dev_info structure pointed to by dip:</p> <pre>    ddi_remove_minor_node(dip, "dev1");</pre>
<b>SEE ALSO</b>	attach(9E), detach(9E), ddi_create_minor_node(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	ddi_removing_power – check whether DDI_SUSPEND might result in power being removed from a device
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_removing_power(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p>
<b>PARAMETERS</b>	<p>The ddi_removing_power() function supports the following parameter:</p> <p><i>dip</i> pointer to the device's dev_info structure</p>
<b>RETURN VALUES</b>	<p>The ddi_removing_power() function returns:</p> <p>1 Power might be removed by the framework as a result of the pending DDI_SUSPEND call.</p> <p>0 Power will not be removed by the framework as a result of the pending DDI_SUSPEND call.</p>
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Protecting a Tape from Abrupt Power Removal</p> <p>A tape driver that has hardware that would damage the tape if power is removed might include this code in its detach(9E) code:</p> <pre>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) &amp;&amp; xxdamages_tape(dip))             return (DDI_FAILURE);     ... }</pre>

ddi\_removing\_power(9F)

**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*

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,                  uint8_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,                   uint16_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,                   uint32_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,                   uint64_t *dev_addr, size_t recount, uint_t flags);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>handle</i>            The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).</p> <p><i>host_addr</i>        Base host address.</p> <p><i>dev_addr</i>         Base device address.</p> <p><i>recount</i>         Number of data accesses to perform.</p> <p><i>flags</i>            Device address flags:</p> <p>                  DDI_DEV_AUTOINCR                   Automatically increment the device address, <i>dev_addr</i>, during data accesses.</p> <p>                  DDI_DEV_NO_AUTOINCR                   Do not advance the device address, <i>dev_addr</i>, during data accesses.</p>
<b>DESCRIPTION</b>	<p>These routines generate multiple reads from the mapped memory or device register. <i>recount</i> data is copied from the device address, <i>dev_addr</i>, to the host address, <i>host_addr</i>. 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, <i>dev_addr</i>. <i>dev_addr</i> and <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>

## ddi\_rep\_get8(9F)

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, *dev\_addr*, as a memory buffer location on the device and increment its address on the next input datum. However, when the *flags* argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

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

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

**SEE ALSO** `ddi_get8(9F)`, `ddi_put8(9F)`, `ddi_regs_map_free(9F)`,  
`ddi_regs_map_setup(9F)`, `ddi_rep_put8(9F)`

**NOTES** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

Previous Name	New Name
<code>ddi_rep_getb</code>	<code>ddi_rep_get8</code>
<code>ddi_rep_getw</code>	<code>ddi_rep_get16</code>
<code>ddi_rep_getl</code>	<code>ddi_rep_get32</code>
<code>ddi_rep_getll</code>	<code>ddi_rep_get64</code>

<b>NAME</b>	ddi_report_dev – announce a device
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>ddi_report_dev</b>(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> a pointer the device's dev_info structure.
<b>DESCRIPTION</b>	ddi_report_dev() prints a banner at boot time, announcing the device pointed to by <i>dip</i> . 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.
<b>CONTEXT</b>	ddi_report_dev() can be called from user context.
<b>SEE ALSO</b>	dmesg(1M), kernel(1M) <i>Writing Device Drivers</i>

## ddi\_rep\_put8(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,                  uint8_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,                   uint16_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,                   uint32_t *dev_addr, size_t recount, uint_t flags);  void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,                   uint64_t *dev_addr, size_t recount, uint_t flags);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>handle</i> The data access handle returned from setup calls, such as ddi_regs_map_setup(9F). <i>host_addr</i> Base host address. <i>dev_addr</i> Base device address. <i>recount</i> Number of data accesses to perform. <i>flags</i> Device address flags: DDI_DEV_AUTOINCR Automatically increment the device address, <i>dev_addr</i> , during data accesses. DDI_DEV_NO_AUTOINCR Do not advance the device address, <i>dev_addr</i> , during data accesses.
<b>DESCRIPTION</b>	<p>These routines generate multiple writes to the mapped memory or device register. <i>recount</i> data is copied from the host address, <i>host_addr</i>, to the device address, <i>dev_addr</i>. 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, <i>dev_addr</i>. <i>dev_addr</i> and <i>host_addr</i> must be aligned to the datum boundary described by the function.</p> <p>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.</p>

## ddi\_rep\_put8(9F)

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, *dev\_addr*, as a memory buffer location on the device and increment its address on the next input datum. However, when the *flags* argument is 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
<code>ddi_rep_putb</code>	<code>ddi_rep_put8</code>
<code>ddi_rep_putw</code>	<code>ddi_rep_put16</code>
<code>ddi_rep_putl</code>	<code>ddi_rep_put32</code>
<code>ddi_rep_putll</code>	<code>ddi_rep_put64</code>

## ddi\_root\_node(9F)

<b>NAME</b>	ddi_root_node – get the root of the dev_info tree
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  dev_info_t *<b>ddi_root_node</b>(void);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>DESCRIPTION</b>	ddi_root_node() returns a pointer to the root node of the device information tree.
<b>RETURN VALUES</b>	ddi_root_node() returns a pointer to a device information structure.
<b>CONTEXT</b>	ddi_root_node() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

<b>NAME</b>	ddi_segmap, ddi_segmap_setup – set up a user mapping using seg_dev
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	These interfaces are obsolete. See devmap(9E) for an alternative to ddi_segmap(). Use devmap_setup(9F) instead of ddi_segmap_setup().
<b>PARAMETERS</b>	<p><i>dev</i>           The device whose memory is to be mapped.</p> <p><i>offset</i>        The offset within device memory at which the mapping begins.</p> <p><i>asp</i>           An opaque pointer to the user address space into which the device memory should be mapped.</p> <p><i>addrp</i>         Pointer to the starting address within the user address space to which the device memory should be mapped.</p> <p><i>len</i>           Length (in bytes) of the memory to be mapped.</p> <p><i>prot</i>          A bit field that specifies the protections. Some combinations of possible settings are:</p> <p style="margin-left: 40px;">PROT_READ     Read access is desired.</p> <p style="margin-left: 40px;">PROT_WRITE     Write access is desired.</p> <p style="margin-left: 40px;">PROT_EXEC     Execute access is desired.</p> <p style="margin-left: 40px;">PROT_USER     User-level access is desired (the mapping is being done as a result of a mmap(2) system call).</p> <p style="margin-left: 40px;">PROT_ALL     All access is desired.</p> <p><i>maxprot</i>       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 &amp; prot) != prot then there is an access violation.</p> <p><i>flags</i>         Flags indicating type of mapping. Possible values are (other bits may be set):</p>

## ddi\_segmap(9F)

	<b>MAP_PRIVATE</b>	Changes are private.				
	<b>MAP_SHARED</b>	Changes should be shared.				
	<b>MAP_FIXED</b>	The user specified an address in <i>*addrp</i> rather than letting the system pick an address.				
	<i>credp</i>	Pointer to user credential structure.				
<b>ddi_segmap_setup()</b>	<i>dev_acc_attr</i>	Pointer to a <code>ddi_device_acc_attr(9S)</code> structure which contains the device access attributes to apply to this mapping.				
	<i>rnumber</i>	Index number to the register address space set.				
<b>DESCRIPTION</b>	<p>Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use <code>ddi_devmap_segmap(9F)</code> instead. See <code>ddi_devmap_segmap(9F)</code> for details.</p> <p><code>ddi_segmap()</code> and <code>ddi_segmap_setup()</code> set up user mappings to device space. When setting up the mapping, the <code>ddi_segmap()</code> and <code>ddi_segmap_setup()</code> routines call the <code>mmap(9E)</code> entry point to validate the range to be mapped. When a user process accesses the mapping, the driver's <code>mmap(9E)</code> 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.</p> <p><code>ddi_segmap()</code> is typically used as the <code>segmap(9E)</code> entry in the <code>cb_ops(9S)</code> structure for those devices that do not choose to provide their own <code>segmap(9E)</code> entry point. However, some drivers may have their own <code>segmap(9E)</code> entry point to do some initial processing on the parameters and then call <code>ddi_segmap()</code> to establish the default memory mapping.</p> <p><code>ddi_segmap_setup()</code> is used in the driver's <code>segmap(9E)</code> entry point to set up the mapping and assign device access attributes to that mapping. <i>rnumber</i> specifies the register set representing the range of device memory being mapped. See <code>ddi_device_acc_attr(9S)</code> for details regarding what device access attributes are available.</p> <p><code>ddi_segmap_setup()</code> cannot be used directly in the <code>cb_ops(9S)</code> structure and requires a driver to have a <code>segmap(9E)</code> entry point.</p>					
<b>RETURN VALUES</b>	<p><code>ddi_segmap()</code> and <code>ddi_segmap_setup()</code> return the following values:</p> <table><tr><td>0</td><td>Successful completion.</td></tr><tr><td>Non-zero</td><td>An error occurred. In particular, they return <code>ENXIO</code> if the range to be mapped is invalid.</td></tr></table>		0	Successful completion.	Non-zero	An error occurred. In particular, they return <code>ENXIO</code> if the range to be mapped is invalid.
0	Successful completion.					
Non-zero	An error occurred. In particular, they return <code>ENXIO</code> if the range to be mapped is invalid.					
<b>CONTEXT</b>	<code>ddi_segmap()</code> and <code>ddi_segmap_setup()</code> 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)`, `ddi_mapdev(9F)`, `cb_ops(9S)`, `ddi_device_acc_attr(9S)`

*Writing Device Drivers*

**NOTES** If driver notification of user accesses to the mappings is required, the driver should use `ddi_mapdev(9F)` instead.

## ddi\_slaveonly(9F)

<b>NAME</b>	ddi_slaveonly – tell if a device is installed in a slave access only location
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int ddi_slaveonly(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>dip</i> A pointer to the device's dev_info structure.
<b>DESCRIPTION</b>	ddi_slaveonly() tells the caller if the bus, or part of the bus that the device is installed on, does not permit the device to become a DMA master, that is, whether the device has been installed in a slave access only slot.
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	ddi_slaveonly() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>state_p</i> Address of the opaque state pointer which will be initialized by ddi_soft_state_init() to point to implementation dependent data.</p> <p><i>size</i> Size of the item which will be allocated by subsequent calls to ddi_soft_state_zalloc().</p> <p><i>n_items</i> A hint of the number of items which will be preallocated; zero is allowed.</p> <p><i>state</i> An opaque pointer to implementation-dependent data that describes the soft state.</p> <p><i>item</i> The item number for the state structure; usually the instance number of the associated devinfo node.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p> <p>To use the routines, the driver writer needs to declare a state pointer, <i>state_p</i>, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.</p> <p>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.</p> <p>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</p>

## ddi\_soft\_state(9F)

the devinfo node, obtained with `ddi_get_instance(9F)`. The routine attempts to allocate space for the new structure, and if the space allocation was successful, `DDI_SUCCESS` is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling `ddi_get_soft_state()` with the appropriate item number.

The space used by a given soft state structure can be returned to the system using `ddi_soft_state_free()`. This routine is usually called from the driver's `detach(9E)` entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using `ddi_soft_state_fini()`. This routine can be called from the driver's `_fini(9E)` routine.

The `ddi_soft_state_zalloc()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

### RETURN VALUES

`ddi_get_soft_state()`

`NULL` The requested state structure was not allocated at the time of the call.

*pointer* The pointer to the state structure.

`ddi_soft_state_init()`

`0` The allocation was successful.

`EINVAL` Either the size parameter was zero, or the *state\_p* parameter was invalid.

`ddi_soft_state_zalloc()`

`DDI_SUCCESS` The allocation was successful.

`DDI_FAILURE` The routine failed to allocate the storage required; either the *state* parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

### CONTEXT

`ddi_soft_state_init()`, and `ddi_soft_state_alloc()` can be called from user context only, since they may internally call `kmem_zalloc(9F)` with the `KM_SLEEP` flag.

The `ddi_soft_state_fini()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines can be called from any driver context.

**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)
            return (DDI_FAILURE);
        softc = ddi_get_soft_state(statep, instance);
        softc->dip = dip;
        ...
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
}
```

## ddi\_soft\_state(9F)

### EXAMPLE 1 Creating and Removing Data Structures (Continued)

```
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {

    case DDI_DETACH:
        instance = ddi_get_instance(dip);
        ...
        ddi_soft_state_free(statep, instance);
        return (DDI_SUCCESS);

    default:
        return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
    ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}
```

**SEE ALSO** \_fini(9E), \_init(9E), attach(9E), detach(9E), ddi\_get\_instance(9F), getminor(9F), kmem\_zalloc(9F)

#### *Writing Device Drivers*

**WARNINGS** There is no attempt to validate the *item* parameter given to `ddi_soft_state_zalloc()` other than it must be a positive signed integer. Therefore very large *item* numbers may cause the driver to hang forever waiting for virtual memory resources that can never be satisfied.

**NOTES** If necessary, a hierarchy of state structures can be constructed by embedding state pointers in higher order state structures.

**DIAGNOSTICS** All of the messages described below usually indicate bugs in the driver and should not appear in normal operation of the system.

```
WARNING: ddi_soft_state_zalloc: bad handle  
WARNING: ddi_soft_state_free: bad handle  
WARNING: ddi_soft_state_fini: bad handle
```

The implementation-dependent information kept in the state variable is corrupt.

```
WARNING: ddi_soft_state_free: null handle  
WARNING: ddi_soft_state_fini: null handle
```

The routine has been passed a null or corrupt state pointer. Check that `ddi_soft_state_init()` has been called.

```
WARNING: ddi_soft_state_free: item %d not in range [0..%d]
```

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.

## ddi\_umem\_alloc(9F)

<b>NAME</b>	ddi_umem_alloc, ddi_umem_free – allocate and free page-aligned kernel memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/sunddi.h&gt;  void *<b>ddi_umem_alloc</b>(size_t <i>size</i>, int <i>flag</i>, ddi_umem_cookie_t     *<i>cookiep</i>);  void <b>ddi_umem_free</b>(ddi_umem_cookie_t <i>cookie</i>);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	
<b>ddi_umem_alloc()</b>	<p><i>size</i>        Number of bytes to allocate.</p> <p><i>flag</i>        Used to determine the sleep and pageable conditions.</p> <p>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.</p> <p>The default condition is to allocate locked memory; this can be changed to allocate pageable memory using the DDI_UMEM_PAGEABLE flag.</p> <p><i>cookiep</i>     Pointer to a kernel memory cookie.</p>
<b>ddi_umem_free()</b>	<p><i>cookie</i>     A kernel memory cookie allocated in ddi_umem_alloc().</p>
<b>DESCRIPTION</b>	<p>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 <i>size</i>). 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.</p> <p><i>flag</i> 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.</p> <p>*<i>cookiep</i> is a pointer to the kernel memory cookie that describes the kernel memory being allocated. A typical use of <i>cookiep</i> is in devmap_umem_setup(9F) when the drivers want to export the kernel memory to a user application.</p> <p>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.</p>
<b>RETURN VALUES</b>	Non-null        Successful completion. ddi_umem_alloc() returns a pointer to the allocated memory.

## ddi\_umem\_alloc(9F)

	NULL	Memory cannot be allocated by <code>ddi_umem_alloc()</code> because <code>DDI_UMEM_NOSLEEP</code> is set and the system is out of resources.
<b>CONTEXT</b>		<code>ddi_umem_alloc()</code> can be called from any context if <i>flag</i> is set to <code>DDI_UMEM_NOSLEEP</code> . If <code>DDI_UMEM_SLEEP</code> is set, <code>ddi_umem_alloc()</code> can be called from user and kernel context only. <code>ddi_umem_free()</code> can be called from any context.
<b>SEE ALSO</b>		<code>devmap(9E)</code> , <code>condvar(9F)</code> , <code>devmap_umem_setup(9F)</code> , <code>kmem_alloc(9F)</code> , <code>mutex(9F)</code> , <code>rwlock(9F)</code> , <code>semaphore(9F)</code>
		<i>Writing Device Drivers</i>
<b>WARNINGS</b>		<p>Setting the <code>DDI_UMEM_PAGEABLE</code> flag in <code>ddi_umem_alloc()</code> 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:</p> <ul style="list-style-type: none"><li>■ For synchronization objects such as locks and condition variables. See <code>mutex(9F)</code>, <code>semaphore(9F)</code>, <code>rwlock(9F)</code>, and <code>condvar(9F)</code>.</li><li>■ For driver interrupt routines.</li></ul> <p>Memory allocated using <code>ddi_umem_alloc()</code> without setting <code>DDI_UMEM_PAGEABLE</code> 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.</p> <p>Excessive use of kernel memory is likely to effect overall system performance. Over-commitment of kernel memory may cause unpredictable consequences.</p> <p>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.</p> <p>Do not call <code>ddi_umem_alloc()</code> within <code>DDI_SUSPEND</code> and <code>DDI_RESUME</code> operations. Memory acquired at these times is not reliable. In some cases, such a call can cause a system to hang.</p>
<b>NOTES</b>		<code>ddi_umem_alloc(0, flag, cookiep)</code> always returns <code>NULL</code> . <code>ddi_umem_free(NULL)</code> has no effects on system.

## ddi\_umem\_iosetup(9F)

<b>NAME</b>	ddi_umem_iosetup – Setup I/O requests to application memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  struct buf *<b>ddi_umem_iosetup</b>(ddi_umem_cookie_t <i>cookie</i>, off_t <i>off</i>,     size_t <i>len</i>, int <i>direction</i>, dev_t <i>dev</i>, daddr_t <i>blkno</i>, int (*<i>iodone</i>)     (struct buf *), int <i>sleepflag</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)
<i>cookie</i>	The kernel memory cookie allocated by ddi_umem_lock(9F).
<i>off</i>	Offset from the start of the cookie.
<i>len</i>	Length of the I/O request in bytes.
<i>direction</i>	Must be set to B_READ for reads from the device or B_WRITE for writes to the device.
<i>dev</i>	Device number
<i>blkno</i>	Block number on device.
<i>iodone</i>	Specific biodone(9F) routine.
<i>sleepflag</i>	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.
<b>DESCRIPTION</b>	<p>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).</p> <p>The ddi_umem_iosetup(9F) function returns a pointer to a buf(9S) structure corresponding to the memory cookie <i>cookie</i>. 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.</p> <p>The <i>off</i> parameter specifies the offset from the start of the cookie. The <i>len</i> parameter represents the length of region to be mapped by the buffer. The <i>direction</i> 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 <i>flags</i> parameter set to DDI_UMEMLOCK_READ, the <i>direction</i> parameter in ddi_umem_iosetup() should be set to B_WRITE.</p>

## ddi\_umem\_iosetup(9F)

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)*, *ddi\_umem\_unlock(9F)*, *freerbuf(9F)*, *physio(9F)*, *buf(9S)*

## ddi\_umem\_lock(9F)

<b>NAME</b>	ddi_umem_lock, ddi_umem_unlock – lock and unlock memory pages
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>ddi_umem_lock</b>(caddr_t <i>addr</i>, size_t <i>len</i>, int <i>flags</i>,     ddi_umem_cookie_t *<i>cookiep</i>);  void <b>ddi_umem_unlock</b>(ddi_umem_cookie_t <i>cookie</i>);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	
<b>ddi_umem_lock</b>	<p><i>addr</i> Virtual address of memory object</p> <p><i>len</i> Length of memory object in bytes</p> <p><i>flags</i> Valid flags include:</p> <ul style="list-style-type: none"><li>DDI_UMEMLOCK_READ Memory pages are locked to be read from. (Disk write or a network send.)</li><li>DDI_UMEMLOCK_WRITE Memory pages are locked to be written to. (Disk read or a network receive.)</li></ul> <p><i>cookiep</i> Pointer to a kernel memory cookie.</p>
<b>ddi_umem_unlock</b>	<p><i>cookie</i> Kernel memory cookie allocated by <code>ddi_umem_lock()</code>.</p>
<b>DESCRIPTION</b>	<p>The <code>ddi_umem_lock(9F)</code> function locks down the physical pages (including I/O pages) that correspond to the current process' virtual address range [<code>addr</code>, <code>addr + size</code>) and fills in a cookie representing the locked pages. This cookie can be used to create a <code>buf(9S)</code> structure that can be used to perform I/O (see <code>ddi_umem_iosetup(9F)</code> and <code>ddi_dma_buf_bind_handle(9F)</code>), or it can be used with <code>devmap_umem_setup(9F)</code> to export the memory to an application.</p> <p>The virtual address and length specified must be at a page boundary and the mapping performed in terms of the system page size. See <code>pagesize(1)</code>.</p> <p>The flags argument indicates the intended use of the locked memory. Set flags to <code>DDI_UMEMLOCK_READ</code> if the memory pages will be read (for example, in a disk write or a network send.) Set flags to <code>DDI_UMEMLOCK_WRITE</code> 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.</p> <p>To unlock the locked pages, the drivers call <code>ddi_umem_unlock(9F)</code> with the cookie obtained from <code>ddi_umem_lock(9F)</code>.</p>

	The process is not allowed to <code>exec(2)</code> or <code>fork(2)</code> 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 <code>close(2)</code> .
<b>RETURN VALUES</b>	On success, a 0 is returned. Otherwise, one of the following <code>errno</code> values is returned.
	<b>EFAULT</b> User process has no mapping at that address range or does not support locking
	<b>EACCES</b> User process does not have the required permission.
	<b>ENOMEM</b> The system does not have sufficient resources to lock memory.
	<b>EAGAIN</b> Could not allocate system resources required to lock the pages. The <code>ddi_umem_lock()</code> could succeed at a later time.
	<b>EINVAL</b> Requested memory is not aligned on a system page boundary.
<b>CONTEXT</b>	The <code>ddi_umem_lock()</code> function can only be called from user context; <code>ddi_umem_unlock()</code> from user, kernel, and interrupt contexts.
<b>SEE ALSO</b>	<code>ddi_umem_iosetup(9F)</code> , <code>ddi_dma_buf_bind_handle(9F)</code> , <code>devmap_umem_setup(9F)</code> , <code>ddi_umem_alloc(9F)</code>
<b>NOTES</b>	The <code>ddi_umem_lock(9F)</code> function consumes physical memory. The driver is responsible for a speedy unlock to free up the resources.
	<code>ddi_umem_unlock()</code> can defer unlocking of the pages to a later time depending on the implementation.

## delay(9F)

<b>NAME</b>	delay – delay execution for a specified number of clock ticks
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  void <b>delay</b>(clock_t ticks);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>ticks</i> The number of clock cycles to delay.
<b>DESCRIPTION</b>	<p><code>delay()</code> 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 <code>drv_usec2ohz(9F)</code> to convert microseconds into clock ticks.</p> <p><code>delay()</code> uses <code>timeout(9F)</code> to schedule an internal function to be called after the specified amount of time has elapsed. <code>delay()</code> then waits until the function is called. Because <code>timeout()</code> is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use <code>cv_timedwait(9F)</code> rather than <code>delay()</code>.</p> <p><code>delay()</code> does not busy-wait. If busy-waiting is required, use <code>drv_usecwait(9F)</code>.</p>
<b>CONTEXT</b>	<code>delay()</code> can be called from user and kernel contexts.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> <code>delay()</code> Example</p> <p>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.</p> <pre>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 8     . . . 9 /* get device registers */ 10 register struct device *rp = ... 11 12 while (rp-&gt;status &amp; NOPAPER) { /* while printer is out of paper */ 13     /* display message and ring bell */ 14     /* on system console */ 15     cmn_err(CE_WARN, "^007", 16             (getminor(dev) &amp; 0xf)); 17     /* wait one minute and try again */ 18     delay(60 * drv_usec2ohz(1000000)); 19 }</pre>

delay(9F)

**EXAMPLE 1** delay() Example (Continued)

**SEE ALSO** biodone(9F), biowait(9F), cv\_timedwait(9F), ddi\_in\_panic(9F),  
drv\_hztousec(9F), drv\_usectohz(9F), drv\_usecwait(9F), timeout(9F),  
untimeout(9F)

*Writing Device Drivers*

## devmap\_default\_access(9F)

<b>NAME</b>	devmap_default_access – default driver memory access function
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int devmap_default_access(devmap_cookie_t dhp, void *pvtp,     offset_t off, size_t len, uint_t type, uint_t rw);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>dhp</i> An opaque mapping handle that the system uses to describe the mapping. <i>pvtp</i> Driver private mapping data. <i>off</i> User offset within the logical device memory at which the access begins. <i>len</i> Length (in bytes) of the memory being accessed. <i>type</i> Type of access operation. <i>rw</i> Type of access.
<b>DESCRIPTION</b>	<p>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.</p> <p>devmap_default_access() can either be called from devmap_access(9E) or be used as the devmap_access(9E) entry point. The arguments <i>dhp</i>, <i>pvtp</i>, <i>off</i>, <i>len</i>, <i>type</i>, and <i>rw</i> are provided by the devmap_access(9E) entry point and must not be modified.</p>
<b>RETURN VALUES</b>	0 Successful completion. Non-zero An error occurred.
<b>CONTEXT</b>	devmap_default_access() must be called from the driver's devmap_access(9E) entry point.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using devmap_default_access in devmap_access.</p> <p>The following shows an example of using devmap_default_access() in the devmap_access(9E) entry point.</p> <pre>. . . #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,</pre>

**EXAMPLE 1** Using devmap\_default\_access in devmap\_access. (Continued)

```

    size_t length, uint_t type, uint_t rw)
{
    .....
    /*
     * see devmap_contextmgt(9E) for an example
     */
}

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;

    /*
     * check if off is within the range that supports
     * context management.
     */
    if ((diff = off - OFF_DO_CTXMG) >= 0 && diff < CTXMGT_SIZE) {
        /*
         * calculates the length for context switching
         */
        if ((len + off) > (OFF_DO_CTXMG + CTXMGT_SIZE))
            return (-1);
        /*
         * perform context switching
         */
        err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
            rw, xx_context_mgt);
    }
    /*
     * check if off is within the range that does normal
     * memory mapping.
     */
    } else if ((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) {
        if ((len + off) > (OFF_NORMAL + NORMAL_SIZE))
            return (-1);
        err = devmap_default_access(dhp, pvtp, off, len, type, rw);
    } else
        return (-1);

    return (err);
}

```

**SEE ALSO** devmap\_access(9E), devmap\_do\_ctxmgt(9F), devmap\_callback\_ctl(9S)

*Writing Device Drivers*

## devmap\_devmem\_setup(9F)

<b>NAME</b>	devmap_devmem_setup, devmap_umem_setup – set driver memory mapping parameters										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip,     struct devmap_callback_ctl *callbackops, uint_t rnumber, offset_t     roff, size_t len, uint_t maxprot, uint_t flags,     ddi_device_acc_attr_t *accattrp);  int devmap_umem_setup(devmap_cookie_t dhp, dev_info_t *dip, struct     devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie,     offset_t koff, size_t len, uint_t maxprot, uint_t flags,     ddi_device_acc_attr_t *accattrp);</pre>										
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p>devmap_devmem_setup() parameters:</p> <p><i>dhp</i>                    An opaque mapping handle that the system uses to describe the mapping.</p> <p><i>dip</i>                     Pointer to the device's dev_info structure.</p> <p><i>callbackops</i>            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.</p> <p><i>rnumber</i>                Index number to the register address space set.</p> <p><i>roff</i>                    Offset into the register address space.</p> <p><i>len</i>                     Length (in bytes) of the mapping to be mapped.</p> <p><i>maxprot</i>                Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:</p> <table border="0" style="margin-left: 20px;"> <tr> <td>PROT_READ</td> <td>Read access is allowed.</td> </tr> <tr> <td>PROT_WRITE</td> <td>Write access is allowed.</td> </tr> <tr> <td>PROT_EXEC</td> <td>Execute access is allowed.</td> </tr> <tr> <td>PROT_USER</td> <td>User-level access is allowed (the mapping is being done as a result of a mmap(2) system call).</td> </tr> <tr> <td>PROT_ALL</td> <td>All access is allowed.</td> </tr> </table> <p><i>flags</i>                  Must be set to 0.</p> <p><i>accattrp</i>                Pointer to a ddi_device_acc_attr(9S) structure. The structure contains the device access attributes to be applied to this range of memory.</p>	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.
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.										

devmap\_umem\_setup() parameters:

<i>dhp</i>	An opaque data structure that the system uses to describe the mapping.										
<i>dip</i>	Pointer to the device's <code>dev_info</code> structure.										
<i>callbackops</i>	Pointer to a <code>devmap_callback_ctl(9S)</code> structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.										
<i>cookie</i>	A kernel memory <i>cookie</i> (see <code>ddi_umem_alloc(9F)</code> ).										
<i>koff</i>	Offset into the kernel memory defined by <i>cookie</i> .										
<i>len</i>	Length (in bytes) of the mapping to be mapped.										
<i>maxprot</i>	Maximum protection flag possible for attempted mapping. Some combinations of possible settings are: <table> <tr> <td><code>PROT_READ</code></td> <td>Read access is allowed.</td> </tr> <tr> <td><code>PROT_WRITE</code></td> <td>Write access is allowed.</td> </tr> <tr> <td><code>PROT_EXEC</code></td> <td>Execute access is allowed.</td> </tr> <tr> <td><code>PROT_USER</code></td> <td>User-level access is allowed (the mapping is being done as a result of a <code>mmap(2)</code> system call).</td> </tr> <tr> <td><code>PROT_ALL</code></td> <td>All access is allowed.</td> </tr> </table>	<code>PROT_READ</code>	Read access is allowed.	<code>PROT_WRITE</code>	Write access is allowed.	<code>PROT_EXEC</code>	Execute access is allowed.	<code>PROT_USER</code>	User-level access is allowed (the mapping is being done as a result of a <code>mmap(2)</code> system call).	<code>PROT_ALL</code>	All access is allowed.
<code>PROT_READ</code>	Read access is allowed.										
<code>PROT_WRITE</code>	Write access is allowed.										
<code>PROT_EXEC</code>	Execute access is allowed.										
<code>PROT_USER</code>	User-level access is allowed (the mapping is being done as a result of a <code>mmap(2)</code> system call).										
<code>PROT_ALL</code>	All access is allowed.										
<i>flags</i>	Must be set to 0.										
<i>accattrp</i>	Pointer to a <code>ddi_device_acc_attr(9S)</code> 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 <code>mmap(2)</code> 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 <code>fork(2)</code> .
Unmapping	User has called <code>munmap(2)</code> on the mapping or is exiting, <code>exit(2)</code> .

## devmap\_devmem\_setup(9F)

See `devmap_map(9E)`, `devmap_access(9E)`, `devmap_dup(9E)`, and `devmap_unmap(9E)` for details on these entry points.

By specifying a valid *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*

<b>NAME</b>	devmap_do_ctxmgt – perform device context switching on a mapping
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dhp</i> An opaque mapping handle that the system uses to describe the mapping.</p> <p><i>pvtp</i> Driver private mapping data.</p> <p><i>off</i> User offset within the logical device memory at which the access begins.</p> <p><i>len</i> Length (in bytes) of the memory being accessed.</p> <p><i>devmap_contextmgt</i> The address of driver function that the system will call to perform context switching on a mapping. See devmap_contextmgt(9E) for details.</p> <p><i>type</i> Type of access operation. Provided by devmap_access(9E). Should not be modified.</p> <p><i>rw</i> Direction of access. Provided by devmap_access(9E). Should not be modified.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>devmap_do_ctxmgt () performs context switching on the mapping object identified by <i>dhp</i> and <i>pvtp</i> in the range specified by <i>off</i> and <i>len</i>. The arguments <i>dhp</i>, <i>pvtp</i>, <i>type</i>, and <i>rw</i> are provided by the devmap_access(9E) entry point and must not be modified. The range from <i>off</i> to <i>off+len</i> must support context switching.</p> <p>The system will pass through <i>dhp</i>, <i>pvtp</i>, <i>off</i>, <i>len</i>, <i>type</i>, and <i>rw</i> 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 ().</p>
<b>RETURN VALUES</b>	<p>0 Successful completion.</p> <p>Non-zero An error occurred.</p>

## devmap\_do\_ctxmgt(9F)

**CONTEXT** devmap\_do\_ctxmgt () must be called from the driver's devmap\_access(9E) entry point.

**EXAMPLES** **EXAMPLE 1** Using devmap\_do\_ctxmgt in the devmap\_access entry point.

The following shows an example of using devmap\_do\_ctxmgt () in the devmap\_access(9E) entry point.

```
. . .
#define OFF_DO_CTXMGT  0x40000000
#define OFF_NORMAL    0x40100000
#define CTXMGT_SIZE   0x100000
#define NORMAL_SIZE   0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
 */
static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset,
              size_t length, uint_t type, uint_t rw)
{
    . . . . .
    /*
     * see devmap_contextmgt(9E) for an example
     */
}

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
               size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;

    /*
     * check if off is within the range that supports
     * context management.
     */
    if ((diff = off - OFF_DO_CTXMGT) >= 0 && diff < CTXMGT_SIZE) {
        /*
         * calculates the length for context switching
         */
        if ((len + off) > (OFF_DO_CTXMGT + CTXMGT_SIZE))
            return (-1);
        /*
         * perform context switching
         */
        err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
                              rw, xx_context_mgt);
    }
    /*
     * check if off is within the range that does normal
     * memory mapping.
     */
}
```

**EXAMPLE 1** Using `devmap_do_ctxmgt` in the `devmap_access` entry point. *(Continued)*

```
    } 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*

## devmap\_set\_ctx\_timeout(9F)

<b>NAME</b>	devmap_set_ctx_timeout – set the timeout value for the context management callback
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  void <b>devmap_set_ctx_timeout</b>(devmap_cookie_t <i>dhp</i>, clock_t <i>ticks</i>);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>dhp</i>        An opaque mapping handle that the system uses to describe the mapping.</p> <p><i>ticks</i>      Number of clock ticks to wait between successive calls to the context management callback function.</p>
<b>DESCRIPTION</b>	<p>devmap_set_ctx_timeout() specifies the time interval for the system to wait between successive calls to the driver's context management callback function, devmap_contextmgt(9E).</p> <p>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.</p>
<b>CONTEXT</b>	devmap_set_ctx_timeout() can be called from user or interrupt context.
<b>SEE ALSO</b>	devmap_contextmgt(9E), devmap_map(9E), timeout(9F)

<b>NAME</b>	devmap_setup, ddi_devmap_segmap – set up a user mapping to device memory using the devmap framework																
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int devmap_setup(dev_t dev, offset_t off, ddi_as_handle_t as,     caddr_t *addrp, size_t len, uint_t prot, uint_t maxprot, uint_t     flags, cred_t *cred);  int ddi_devmap_segmap(dev_t dev, off_t off, ddi_as_handle_t as,     caddr_t *addrp, off_t len, uint_t prot, uint_t maxprot, uint_t flags,     cred_t *cred);</pre>																
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dev</i> Device whose memory is to be mapped.</p> <p><i>off</i> User offset within the logical device memory at which the mapping begins.</p> <p><i>as</i> An opaque data structure that describes the address space into which the device memory should be mapped.</p> <p><i>addrp</i> Pointer to the starting address in the address space into which the device memory should be mapped.</p> <p><i>len</i> Length (in bytes) of the memory to be mapped.</p> <p><i>prot</i> A bit field that specifies the protections. Some possible settings combinations are:</p> <table border="0" style="margin-left: 2em;"> <tr> <td>PROT_READ</td> <td>Read access is desired.</td> </tr> <tr> <td>PROT_WRITE</td> <td>Write access is desired.</td> </tr> <tr> <td>PROT_EXEC</td> <td>Execute access is desired.</td> </tr> <tr> <td>PROT_USER</td> <td>User-level access is desired (the mapping is being done as a result of a mmap(2) system call).</td> </tr> <tr> <td>PROT_ALL</td> <td>All access is desired.</td> </tr> </table> <p><i>maxprot</i> Maximum protection flag possible for attempted mapping; the PROT_WRITE bit may be masked out if the user opened the special file read-only.</p> <p><i>flags</i> Flags indicating type of mapping. The following flags can be specified:</p> <table border="0" style="margin-left: 2em;"> <tr> <td>MAP_PRIVATE</td> <td>Changes are private.</td> </tr> <tr> <td>MAP_SHARED</td> <td>Changes should be shared.</td> </tr> <tr> <td>MAP_FIXED</td> <td>The user specified an address in <i>*addrp</i> rather than letting the system choose an address.</td> </tr> </table> <p><i>cred</i> Pointer to the user credential structure.</p>	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.	MAP_PRIVATE	Changes are private.	MAP_SHARED	Changes should be shared.	MAP_FIXED	The user specified an address in <i>*addrp</i> rather than letting the system choose an address.
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.																
MAP_PRIVATE	Changes are private.																
MAP_SHARED	Changes should be shared.																
MAP_FIXED	The user specified an address in <i>*addrp</i> rather than letting the system choose an address.																

## devmap\_setup(9F)

### DESCRIPTION

`devmap_setup()` and `ddi_devmap_segmap()` allow device drivers to use the devmap framework to set up user mappings to device memory. The devmap framework provides several advantages over the default device mapping framework that is used by `ddi_segmap(9F)` or `ddi_segmap_setup(9F)`. Device drivers should use the devmap framework, if the driver wants to:

- use an optimal MMU pagesize to minimize address translations,
- conserve kernel resources,
- receive callbacks to manage events on the mapping,
- export kernel memory to applications,
- set up device contexts for the user mapping if the device requires context switching,
- assign device access attributes to the user mapping, or
- change the maximum protection for the mapping.

`devmap_setup()` must be called in the `segmap(9E)` entry point to establish the mapping for the application. `ddi_devmap_segmap()` can be called in, or be used as, the `segmap(9E)` entry point. The differences between `devmap_setup()` and `ddi_devmap_segmap()` are in the data type used for *off* and *len*.

When setting up the mapping, `devmap_setup()` and `ddi_devmap_segmap()` call the `devmap(9E)` entry point to validate the range to be mapped. The `devmap(9E)` entry point also translates the logical offset (as seen by the application) to the corresponding physical offset within the device address space. If the driver does not provide its own `devmap(9E)` entry point, `EINVAL` will be returned to the `mmap(2)` system call.

### RETURN VALUES

0	Successful completion.
Non-zero	An error occurred. The return value of <code>devmap_setup()</code> and <code>ddi_devmap_segmap()</code> should be used directly in the <code>segmap(9E)</code> 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*

<b>NAME</b>	devmap_unload, devmap_load – control validation of memory address translations
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dhp</i>        An opaque mapping handle that the system uses to describe the mapping.</p> <p><i>off</i>         User offset within the logical device memory at which the loading or unloading of the address translations begins.</p> <p><i>len</i>         Length (in bytes) of the range being affected.</p> <p><b>devmap_load() only</b></p> <p><i>type</i>        Type of access operation.</p> <p><i>rw</i>         Direction of access.</p>
<b>DESCRIPTION</b>	<p>devmap_unload() and devmap_load() are used to control the validation of the memory mapping described by <i>dhp</i> 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.</p> <p>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.</p> <p>For both routines, the range to be affected is defined by the <i>off</i> and <i>len</i> arguments. Requests affect the entire page containing the <i>off</i> and all pages up to and including the page containing the last byte as indicated by <i>off + len</i>. The arguments <i>type</i> and <i>rw</i> are provided by the system to the calling function (for example, devmap_contextmgt(9E)) and should not be modified.</p> <p>Supplying a value of 0 for the <i>len</i> argument affects all addresses from the <i>off</i> to the end of the mapping. Supplying a value of 0 for the <i>off</i> argument and a value of 0 for <i>len</i> argument affect all addresses in the mapping.</p> <p>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.</p>
<b>RETURN VALUES</b>	0                    Successful completion.

## devmap\_unload(9F)

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*

<b>NAME</b>	disksort – single direction elevator seek sort for buffers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt; void <b>disksort</b>(struct diskhd *dp, struct buf *bp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dp</i>        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.</p> <p><i>bp</i>        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().</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>
<b>CONTEXT</b>	This function can be called from user or interrupt context.
<b>SEE ALSO</b>	strategy(9E), buf(9S) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	disksort() does no locking. Therefore, any locking is completely the responsibility of the caller.

## drv\_getparm(9F)

<b>NAME</b>	drv_getparm – retrieve kernel state information
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  int <b>drv_getparm</b>(unsigned int <i>parm</i>, void *<i>value_p</i>);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI).
	<i>parm</i> The kernel parameter to be obtained. Possible values are:
	LBOLT            Read the value of <code>lbolt</code> . <code>lbolt</code> is a <code>clock_t</code> 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 <code>clock_t</code> . 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 <code>lbolt</code> 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.
	<i>value_p</i> A pointer to the data space in which the value of the parameter is to be copied.
<b>DESCRIPTION</b>	<p>Since the release of the Solaris 2.6 operating environment, the <code>drv_getparm()</code> function has been replaced by <code>ddi_get_lbolt(9F)</code>, <code>ddi_get_time(9F)</code>, and <code>ddi_get_pid(9F)</code>.</p> <p><code>drv_getparm()</code> function verifies that <i>parm</i> corresponds to a kernel parameter that may be read. If the value of <i>parm</i> 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 <i>value_p</i>.</p> <p><code>drv_getparm()</code> 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 <i>value_p</i>. 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.</p>

drv\_getparm(9F)

**RETURN VALUES** | `drv_getparm()` returns 0 to indicate success, -1 to indicate failure. The value stored in the space pointed to by *value\_p* is the value of the parameter if 0 is returned, or undefined if -1 is returned. -1 is returned if you specify a value other than LBOLT, PPGRP, PPID, PSID, TIME, UCRED, or UPROCP. Always check the return code when using this function.

**CONTEXT** | `drv_getparm()` can be called from user context only when using PPGRP, PPID, PSID, UCRED, or UPROCP. It can be called from user or interrupt context when using the LBOLT or TIME argument.

**SEE ALSO** | `ddi_get_lbolt(9F)`, `ddi_get_pid(9F)`, `ddi_get_time(9F)`, `buf(9S)`

*Writing Device Drivers*

## drv\_hztousec(9F)

<b>NAME</b>	drv_hztousec – convert clock ticks to microseconds
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  clock_t <b>drv_hztousec</b>(clock_t <i>hertz</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>hertz</i> The number of clock ticks to convert.
<b>DESCRIPTION</b>	<p>drv_hztousec() converts into microseconds the time expressed by <i>hertz</i>, which is in system clock ticks.</p> <p>The kernel variable <code>lbolt</code>, whose value should be retrieved by calling <code>ddi_get_lbolt(9F)</code>, is the length of time the system has been up since boot and is expressed in clock ticks. Drivers often use the value of <code>lbolt</code> before and after an I/O request to measure the amount of time it took the device to process the request. <code>drv_hztousec()</code> can be used by the driver to convert the reading from clock ticks to a known unit of time.</p>
<b>RETURN VALUES</b>	The number of microseconds equivalent to the <i>hertz</i> parameter. No error value is returned. If the microsecond equivalent to <i>hertz</i> is too large to be represented as a <code>clock_t</code> , then the maximum <code>clock_t</code> value will be returned.
<b>CONTEXT</b>	drv_hztousec() can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>ddi_get_lbolt(9F)</code> , <code>drv_usectohz(9F)</code> , <code>drv_usecwait(9F)</code> <i>Writing Device Drivers</i>

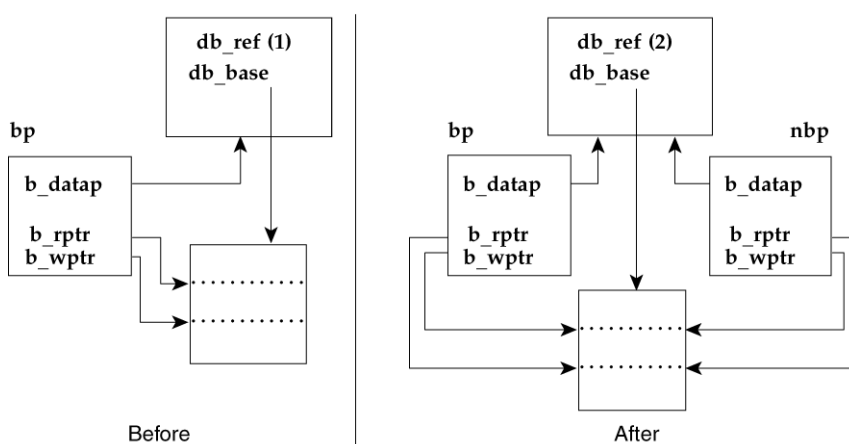
<b>NAME</b>	drv_priv – determine driver privilege
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/cred.h&gt; #include &lt;sys/ddi.h&gt;  int <b>drv_priv</b>(cred_t *cr);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>cr</i> Pointer to the user credential structure.
<b>DESCRIPTION</b>	<i>drv_priv()</i> provides a general interface to the system privilege policy. It determines whether the credentials supplied by the user credential structure pointed to by <i>cr</i> identify a privileged process. This function should only be used when file access modes and special minor device numbers are insufficient to provide protection for the requested driver function. It is intended to replace all calls to <i>suser()</i> and any explicit checks for effective user ID = 0 in driver code.
<b>RETURN VALUES</b>	This routine returns 0 if it succeeds, EPERM if it fails.
<b>CONTEXT</b>	<i>drv_priv()</i> can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

drv\_usectohz(9F)

<b>NAME</b>	drv_usectohz – convert microseconds to clock ticks
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  clock_t <b>drv_usectohz</b>(clock_t <i>microsecs</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>microsecs</i> The number of microseconds to convert.
<b>DESCRIPTION</b>	<p>drv_usectohz() converts a length of time expressed in microseconds to a number of system clock ticks. The time arguments to timeout(9F) and delay(9F) are expressed in clock ticks.</p> <p>drv_usectohz() is a portable interface for drivers to make calls to timeout(9F) and delay(9F) and remain binary compatible should the driver object file be used on a system with a different clock speed (a different number of ticks in a second).</p>
<b>RETURN VALUES</b>	The value returned is the number of system clock ticks equivalent to the <i>microsecs</i> argument. No error value is returned. If the clock tick equivalent to <i>microsecs</i> is too large to be represented as a <code>clock_t</code> , then the maximum <code>clock_t</code> value will be returned.
<b>CONTEXT</b>	drv_usectohz() can be called from user or interrupt context.
<b>SEE ALSO</b>	delay(9F), drv_hztousec(9F), timeout(9F)
	<i>Writing Device Drivers</i>

<b>NAME</b>	drv_usecwait – busy-wait for specified interval
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt;  void <b>drv_usecwait</b>(clock_t <i>microsecs</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>microsecs</i> The number of microseconds to busy-wait.
<b>DESCRIPTION</b>	<p>drv_usecwait () gives drivers a means of busy-waiting for a specified microsecond count. The amount of time spent busy-waiting may be greater than the microsecond count but will minimally be the number of microseconds specified.</p> <p>delay(9F) can be used by a driver to delay for a specified number of system ticks, but it has two limitations. First, the granularity of the wait time is limited to one clock tick, which may be more time than is needed for the delay. Second, delay(9F) may only be invoked from user context and hence cannot be used at interrupt time or system initialization.</p> <p>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.</p>
<b>CONTEXT</b>	drv_usecwait () can be called from user or interrupt context.
<b>SEE ALSO</b>	delay(9F), timeout(9F), untimeout(9F)
	<i>Writing Device Drivers</i>
<b>NOTES</b>	<p>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.</p> <p>drv_usecwait () does not mask out interrupts.</p>

## dupb(9F)

<b>NAME</b>	dupb – duplicate a message block descriptor
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *dupb(mblk_t *bp);</pre>
<b>INTERFACE LEVEL DESCRIPTION</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p>dupb () creates a new mblk_t structure (see msgb(9S)) to reference the message block pointed to by bp.</p> <p>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.</p>
	 <p style="text-align: center;">nbp=dupb(bp);</p>
<b>PARAMETERS</b>	<p><i>bp</i> Pointer to the message block to be duplicated. mblk_t is an instance of the msgb(9S) structure.</p>
<b>RETURN VALUES</b>	<p>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).</p>
<b>CONTEXT</b>	<p>dupb () can be called from user, kernel, or interrupt context.</p>

**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)) {
21                 if ((bp = dupb(hdr)) == NULL)
22                     bp = copyb(hdr);
23                 if (bp == NULL) {
24                     size_t size = msgdsize(mp);
25                     putbq(q, mp);
26                     if (xx->xx_qbufcall_id) {

```

dupb(9F)

**EXAMPLE 1** Using dupb() (Continued)

```
27             /* qbufcall pending */
28             return;
29         }
30         xx->xx_qbufcall_id = qbufcall(q, size,
31             BPRI_MED, xxxcallback, (intptr_t)q);
32         return;
33     }
34     linkb(bp, mp);
35     putnext(q, bp);
36 } else {
37     putbq(q, mp);
38     return;
39 }
40 }
41 }
42 }
43 void
44 xxxcallback(q)
45     queue_t *q;
46 {
47     struct xx *xx = (struct xx *)q->q_ptr;
48
49     xx->xx_qbufcall_id = 0;
50     qenable(q);
51 }
52
53 xxxclose(q, cflag, crp)
54     queue_t *q;
55     int cflag;
56     cred_t *crp;
57 {
58     struct xx *xx = (struct xx *)q->q_ptr;
59     ...
60     if (xx->xx_qbufcall_id)
61         qunbufcall(q, xx->xx_qbufcall_id);
62     ...
63 }
```

**SEE ALSO** [srv\(9E\)](#), [copyb\(9F\)](#), [qbufcall\(9F\)](#), [datab\(9S\)](#), [msgb\(9S\)](#)

*Writing Device Drivers STREAMS Programming Guide*

<b>NAME</b>	dupmsg – duplicate a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *dupmsg(mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>mp</i> Pointer to the message.
<b>DESCRIPTION</b>	dupmsg() forms a new message by copying the message block descriptors pointed to by <i>mp</i> and linking them. dupb(9F) is called for each message block. The data blocks themselves are not duplicated.
<b>RETURN VALUES</b>	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).
<b>CONTEXT</b>	dupmsg() can be called from user, kernel, or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using dupmsg()</p> <p>See copyb(9F) for an example using dupmsg().</p>
<b>SEE ALSO</b>	copyb(9F), copymsg(9F), dupb(9F), datab(9S) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

## enableok(9F)

<b>NAME</b>	enableok – reschedule a queue for service
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void enableok(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> A pointer to the queue to be rescheduled.
<b>DESCRIPTION</b>	enableok () enables queue <i>q</i> to be rescheduled for service. It reverses the effect of a previous call to noenable(9F) on <i>q</i> by turning off the QNOENB flag in the queue.
<b>CONTEXT</b>	enableok () can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using enableok ()</p> <p>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.</p> <pre>1 void 2 qrestart(rdwr_q) 3     register queue_t *rdwr_q; 4 { 5     enableok(rdwr_q); 6     /* re-enable a queue that has been disabled */ 7     (void) qenable(rdwr_q); 8 }</pre>
<b>SEE ALSO</b>	noenable(9F), qenable(9F) <i>Writing Device Drivers STREAMS Programming Guide</i>

<b>NAME</b>	esballoc – allocate a message block using a caller-supplied buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  mblk_t *esballoc (uchar *base, size_t size, uint_t pri, frtn_t                  *fr_rtnp) ;</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>base</i>                    Address of user supplied data buffer.</p> <p><i>size</i>                    Number of bytes in data buffer.</p> <p><i>pri</i>                     Priority of allocation request (to be used by allocb(9F) function, called by esballoc( )).</p> <p><i>fr_rtnp</i>                 Free routine data structure.</p>
<b>DESCRIPTION</b>	<p>esballoc( ) creates a STREAMS message and attaches a user-supplied data buffer in place of a STREAMS data buffer. It calls allocb(9F) to get a message and data block header only. The newly allocated message will have both the <i>b_wptr</i> and <i>b_rptr</i> set to the base of the buffer. As when using allocb(9F), the newly allocated message will have both <i>b_wptr</i> and <i>b_rptr</i> set to the base of the data buffer. The user-supplied data buffer, pointed to by <i>base</i>, is used as the data buffer for the message.</p> <p>When freeb(9F) is called to free the message, the driver's message freeing routine (referenced through the <i>free_rtn</i> structure) is called, with appropriate arguments, to free the data buffer.</p> <p>The <i>free_rtn</i> structure includes the following members:</p> <pre>void (*free_func) ();        /* user's freeing routine */ char *free_arg;             /* arguments to free_func() */</pre> <p>Instead of requiring a specific number of arguments, the <i>free_arg</i> field is defined of type <i>char *</i>. This way, the driver can pass a pointer to a structure if more than one argument is needed.</p> <p>The method by which <i>free_func</i> is called is implementation-specific. The module writer must not assume that <i>free_func</i> will or will not be called directly from STREAMS utility routines like freeb(9F) which free a message block.</p> <p><i>free_func</i> must not call another modules put procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.</p> <p><i>free_func</i> must not access any dynamically allocated data structure that might no longer exist when it runs.</p>
<b>RETURN VALUES</b>	On success, a pointer to the newly allocated message block is returned. On failure, NULL is returned.

esballoc(9F)

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

**SEE ALSO** | allocb(9F), freeb(9F), datab(9S), free\_rtn(9S)

*Writing Device Drivers STREAMS Programming Guide*

**WARNINGS** | The free\_func must be defined in kernel space, should be declared void and accept one argument. It has no user context and must not sleep.

<b>NAME</b>	esbbscall – call function when buffer is available
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; bufcall_id_t <b>esbbscall</b>(uint_t <i>pri</i>, void *<i>func</i>void *<i>arg</i>, void <i>arg</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>pri</i>        Priority of allocation request (to be used by <code>allocb(9F)</code> function, called by <code>esbbscall()</code>)</p> <p><i>func</i>        Function to be called when buffer becomes available.</p> <p><i>arg</i>        Argument to <i>func</i>.</p>
<b>DESCRIPTION</b>	<p><code>esbbscall()</code>, like <code>bufcall(9F)</code>, serves as a <code>timeout(9F)</code> call of indeterminate length. If <code>esbbsalloc(9F)</code> is unable to allocate a message and data block header to go with its externally supplied data buffer, <code>esbbscall()</code> can be used to schedule the routine <i>func</i>, to be called with the argument <i>arg</i> when a buffer becomes available. <i>func</i> may be a routine that calls <code>esbbsalloc(9F)</code> or it may be another kernel function.</p>
<b>RETURN VALUES</b>	<p>On success, a <code>bufcall</code> ID is returned. On failure, 0 is returned. The value returned from a successful call should be saved for possible future use with <code>unbufcall()</code> should it become necessary to cancel the <code>esbbscall()</code> request (as at driver close time).</p>
<b>CONTEXT</b>	<code>esbbscall()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<p><code>allocb(9F)</code>, <code>bufcall(9F)</code>, <code>esbbsalloc(9F)</code>, <code>timeout(9F)</code>, <code>datadb(9S)</code>, <code>unbufcall(9F)</code></p> <p><i>Writing Device Drivers STREAMS Programming Guide</i></p>

## flushband(9F)

<b>NAME</b>	flushband – flush messages for a specified priority band
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void flushband(queue_t *q, unsigned char pri, int flag);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI).
	<i>q</i> Pointer to the queue.
	<i>pri</i> Priority of messages to be flushed.
	<i>flag</i> Valid <i>flag</i> values are:
	FLUSHDATA Flush only data messages (types M_DATA, M_DELAY, M_PROTO, and M_PCPROTO).
	FLUSHALL Flush all messages.
<b>DESCRIPTION</b>	flushband() flushes messages associated with the priority band specified by <i>pri</i> . If <i>pri</i> is 0, only normal and high priority messages are flushed. Otherwise, messages are flushed from the band <i>pri</i> according to the value of <i>flag</i> .
<b>CONTEXT</b>	flushband() can be called from user or interrupt context.
<b>SEE ALSO</b>	flushq(9F)  <i>Writing Device Drivers STREAMS Programming Guide</i>

<b>NAME</b>	flushq – remove messages from a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void flushq(queue_t *q, int flag);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i> Pointer to the queue to be flushed.</p> <p><i>flag</i> Valid <i>flag</i> values are:</p> <p>FLUSHDATA Flush only data messages (types M_DATA M_DELAY M_PROTO and M_PCPROTO).</p> <p>FLUSHALL Flush all messages.</p>
<b>DESCRIPTION</b>	flushq() frees messages and their associated data structures by calling freemsg(9F). If the queue's count falls below the low water mark and the queue was blocking an upstream service procedure, the nearest upstream service procedure is enabled.
<b>CONTEXT</b>	flushq() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using flushq()</p> <p>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.</p> <pre> 1  /* 2  * Module write-side put procedure. 3  */ 4  xxxwput(q, mp) 5      queue_t *q; 6      mblk_t *mp; 7  { 8      switch(mp-&gt;b_datap-&gt;db_type) { 9          case M_FLUSH: 10             if (*mp-&gt;b_rptr &amp; FLUSHR) 11                 flushq(RD(q), FLUSHALL); 12             if (*mp-&gt;b_rptr &amp; FLUSHW) 13                 flushq(q, FLUSHALL); 14             putnext(q, mp); 15             break; 16             . . . 16     } 17 }</pre>
<b>SEE ALSO</b>	flushband(9F), freemsg(9F), putq(9F), qreply(9F)

flushq(9F)

*Writing Device Drivers STREAMS Programming Guide*

<b>NAME</b>	freeb – free a message block
<b>SYNOPSIS</b>	#include <sys/stream.h> void <b>freeb</b> (mblk_t *bp);
<b>PARAMETERS</b>	<i>bp</i> Pointer to the message block to be deallocated. mblk_t is an instance of the msgb(9S) structure.
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>DESCRIPTION</b>	<p>freeb() deallocates a message block. If the reference count of the db_ref member of the datab(9S) structure is greater than 1, freeb() decrements the count. If db_ref equals 1, it deallocates the message block and the corresponding data block and buffer.</p> <p>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.</p>
<b>CONTEXT</b>	freeb() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using freeb()</p> <p>See copyb(9F) for an example of using freeb().</p>
<b>SEE ALSO</b>	<p>allocb(9F), copyb(9F), dupb(9F), esballoc(9F), free_rtn(9S)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## freemsg(9F)

<b>NAME</b>	freemsg – free all message blocks in a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void <b>freemsg</b>(mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>mp</i> Pointer to the message blocks to be deallocated. mblk_t is an instance of the msgb(9S) structure. If <i>mp</i> is NULL, freemsg() immediately returns.
<b>DESCRIPTION</b>	freemsg() calls freeb(9F) to free all message and data blocks associated with the message pointed to by <i>mp</i> .
<b>CONTEXT</b>	freemsg() can be called from user or interrupt context.
<b>EXAMPLES</b>	<b>EXAMPLE 1</b> Using freemsg() See copymsg(9F).
<b>SEE ALSO</b>	copymsg(9F), freeb(9F), msgb(9S) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>
<b>NOTES</b>	The behavior of freemsg() when passed a NULL pointer is Solaris-specific.

<b>NAME</b>	freerbuf – free a raw buffer header
<b>SYNOPSIS</b>	<pre>#include &lt;sys/buf.h&gt; #include &lt;sys/ddi.h&gt;  void <b>freerbuf</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to a previously allocated buffer header structure.
<b>DESCRIPTION</b>	freerbuf () frees a raw buffer header previously allocated by getrbuf(9F). This function does not sleep and so may be called from an interrupt routine.
<b>CONTEXT</b>	freerbuf () can be called from user or interrupt context.
<b>SEE ALSO</b>	getrbuf(9F), kmem_alloc(9F), kmem_free(9F), kmem_zalloc(9F)

## freezestr(9F)

<b>NAME</b>	freezestr, unfreezestr – freeze, thaw the state of a stream
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>freezestr</b>(queue_t *q); void <b>unfreezestr</b>(queue_t *q);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>q</i> Pointer to the message queue to freeze/unfreeze.
<b>DESCRIPTION</b>	<p><code>freezestr()</code> freezes the state of the entire stream containing the queue pair <i>q</i>. 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 <code>unfreezestr()</code>.</p> <p>Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as <code>getq(9F)</code>, <code>putq(9F)</code>, <code>putbq(9F)</code>, etc.)</p>
<b>CONTEXT</b>	These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>
<b>NOTES</b>	<p>The <code>freezestr()</code> and <code>unfreezestr()</code> functions can have a serious impact on system performance. Their use should be very limited. In most cases, there is no need to use <code>freezestr()</code> and there are usually better ways to accomplish what you need to do than by freezing the stream.</p> <p>Calling <code>freezestr()</code> to freeze a stream that is already frozen by the caller will result in a single-party deadlock.</p> <p>The caller of <code>unfreezestr()</code> must be the thread who called <code>freezestr()</code>.</p> <p>STREAMS utility functions such as <code>getq(9F)</code>, <code>putq(9F)</code>, <code>putbq(9F)</code>, and so forth, should not be called by the caller of <code>freezestr()</code> while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.</p>

<b>NAME</b>	geterror – return I/O error
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt; #include &lt;sys/ddi.h&gt;  int <b>geterror</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to a buf(9S) structure.
<b>DESCRIPTION</b>	geterror() returns the error number from the error field of the buffer header structure.
<b>RETURN VALUES</b>	An error number indicating the error condition of the I/O request is returned. If the I/O request completes successfully, 0 is returned.
<b>CONTEXT</b>	geterror() can be called from user or interrupt context.
<b>SEE ALSO</b>	buf(9S) <i>Writing Device Drivers</i>

## gethrtime(9F)

<b>NAME</b>	gethrtime – get high resolution time
<b>SYNOPSIS</b>	<pre>#include &lt;sys/time.h&gt;  hrtime_t <b>gethrtime</b>(void);</pre>
<b>DESCRIPTION</b>	The <code>gethrtime()</code> 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 <code>adjtime(2)</code> or <code>settimeofday(3C)</code> . The hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.
<b>RETURN VALUES</b>	<code>gethrtime()</code> always returns the current high-resolution real time. There are no error conditions.
<b>CONTEXT</b>	There are no restrictions on the context from which <code>gethrtime()</code> can be called.
<b>SEE ALSO</b>	<code>proc(1)</code> , <code>gettimeofday(3C)</code> , <code>settimeofday(3C)</code> , <code>attributes(5)</code>
<b>NOTES</b>	<p>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.</p> <p>The time base used for this function is the same as that for <code>gethrtime(3C)</code>. Values returned by both of these functions can be interleaved for comparison purposes.</p>

<b>NAME</b>	getmajor – get major device number
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/mkdev.h&gt; #include &lt;sys/ddi.h&gt;  major_t <b>getmajor</b>(dev_t dev);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>dev</i> Device number.
<b>DESCRIPTION</b>	getmajor() extracts the major number from a device number.
<b>RETURN VALUES</b>	The major number.
<b>CONTEXT</b>	getmajor() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using getmajor()</p> <p>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.</p> <pre>dev_t dev;  #ifdef DEBUG cmn_err(CE_NOTE, "Driver Started. Major# = %d,            Minor# = %d", getmajor(dev), getminor(dev)); #endif</pre>
<b>SEE ALSO</b>	cmn_err(9F), getminor(9F), makedevice(9F) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	No validity checking is performed. If <i>dev</i> is invalid, an invalid number is returned.

## getminor(9F)

<b>NAME</b>	getminor – get minor device number
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/mkdev.h&gt; #include &lt;sys/ddi.h&gt;  minor_t <b>getminor</b>(dev_t dev);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>dev</i> Device number.
<b>DESCRIPTION</b>	getminor() extracts the minor number from a device number.
<b>RETURN VALUES</b>	The minor number.
<b>CONTEXT</b>	getminor() can be called from user or interrupt context.
<b>EXAMPLES</b>	See the getmajor(9F) manual page for an example of how to use getminor().
<b>SEE ALSO</b>	getmajor(9F), makedevice(9F) <i>Writing Device Drivers</i>
<b>WARNINGS</b>	No validity checking is performed. If <i>dev</i> is invalid, an invalid number is returned.

<b>NAME</b>	get_pktiopb, free_pktiopb – allocate/free a SCSI packet in the iopb map																
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  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);</pre>																
<b>INTERFACE LEVEL</b>	These interfaces are obsolete. Use scsi_alloc_consistent_buf(9F) instead of get_pktiopb(). Use scsi_free_consistent_buf(9F) instead of free_pktiopb().																
<b>PARAMETERS</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>ap</i></td> <td>Pointer to the target's <i>scsi_address</i> structure.</td> </tr> <tr> <td><i>datap</i></td> <td>Pointer to the address of the packet, set by this function.</td> </tr> <tr> <td><i>cdblen</i></td> <td>Number of bytes required for the SCSI command descriptor block (CDB).</td> </tr> <tr> <td><i>statuslen</i></td> <td>Number of bytes required for the SCSI status area.</td> </tr> <tr> <td><i>datalen</i></td> <td>Number of bytes required for the data area of the SCSI command.</td> </tr> <tr> <td><i>readflag</i></td> <td>If non-zero, data will be transferred from the SCSI target.</td> </tr> <tr> <td><i>callback</i></td> <td>Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC</td> </tr> <tr> <td><i>pkt</i></td> <td>Pointer to a <i>scsi_pkt</i>(9S) structure.</td> </tr> </table>	<i>ap</i>	Pointer to the target's <i>scsi_address</i> structure.	<i>datap</i>	Pointer to the address of the packet, set by this function.	<i>cdblen</i>	Number of bytes required for the SCSI command descriptor block (CDB).	<i>statuslen</i>	Number of bytes required for the SCSI status area.	<i>datalen</i>	Number of bytes required for the data area of the SCSI command.	<i>readflag</i>	If non-zero, data will be transferred from the SCSI target.	<i>callback</i>	Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC	<i>pkt</i>	Pointer to a <i>scsi_pkt</i> (9S) structure.
<i>ap</i>	Pointer to the target's <i>scsi_address</i> structure.																
<i>datap</i>	Pointer to the address of the packet, set by this function.																
<i>cdblen</i>	Number of bytes required for the SCSI command descriptor block (CDB).																
<i>statuslen</i>	Number of bytes required for the SCSI status area.																
<i>datalen</i>	Number of bytes required for the data area of the SCSI command.																
<i>readflag</i>	If non-zero, data will be transferred from the SCSI target.																
<i>callback</i>	Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC																
<i>pkt</i>	Pointer to a <i>scsi_pkt</i> (9S) structure.																
<b>DESCRIPTION</b>	<p>get_pktiopb() allocates a <i>scsi_pkt</i> 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_realloc(9F) to allocate the packet and DMA resources.</p> <p><i>callback</i> indicates what get_pktiopb() should do when resources are not available:</p> <table border="0"> <tr> <td style="padding-right: 20px;">NULL_FUNC</td> <td>Do not wait for resources. Return a NULL pointer.</td> </tr> <tr> <td>SLEEP_FUNC</td> <td>Wait indefinitely for resources.</td> </tr> <tr> <td>Other Values</td> <td><i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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.</td> </tr> </table> <p>free_pktiopb() is used for freeing the packet and its associated resources.</p>	NULL_FUNC	Do not wait for resources. Return a NULL pointer.	SLEEP_FUNC	Wait indefinitely for resources.	Other Values	<i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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.										
NULL_FUNC	Do not wait for resources. Return a NULL pointer.																
SLEEP_FUNC	Wait indefinitely for resources.																
Other Values	<i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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.																
<b>RETURN VALUES</b>	get_pktiopb() returns a pointer to the newly allocated <i>scsi_pkt</i> or a NULL pointer.																

## get\_pktiopb(9F)

**CONTEXT** If *callback* is `SLEEP_FUNC`, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

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

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

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_realloc(9F)`, `scsi_pkt(9S)`

*Writing Device Drivers*

**NOTES** The `get_pktiopb()` and `free_pktiopb()` functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, `scsi_alloc_consistent_buf(9F)` and `scsi_free_consistent_buf(9F)`.

`get_pktiopb()` uses scarce resources. For this reason and its obsolescence (see above), its use is discouraged.

<b>NAME</b>	getq – get the next message from a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *getq(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the queue from which the message is to be retrieved.
<b>DESCRIPTION</b>	<p>getq() is used by a service (srv(9E)) routine to retrieve its enqueued messages.</p> <p>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 <i>q</i>. It should be called in a while loop that is exited only when there are no more messages or flow control prevents further processing.</p> <p>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.</p>
<b>RETURN VALUES</b>	If there is a message to retrieve, getq() returns a pointer to it. If no message is queued, getq() returns a NULL pointer.
<b>CONTEXT</b>	getq() can be called from user or interrupt context.
<b>EXAMPLES</b>	See dupb(9F).
<b>SEE ALSO</b>	srv(9E), bcanput(9F), canput(9F), dupb(9F), putbq(9F), putq(9F), qenable(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

## getrbuf(9F)

<b>NAME</b>	getrbuf – get a raw buffer header
<b>SYNOPSIS</b>	<pre>#include &lt;sys/buf.h&gt; #include &lt;sys/kmem.h&gt; #include &lt;sys/ddi.h&gt;  struct buf *getrbuf(int sleepflag);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>sleepflag</i> Indicates whether driver should sleep for free space.
<b>DESCRIPTION</b>	<p>getrbuf () allocates the space for a buffer header to the caller. It is used in cases where a block driver is performing raw (character interface) I/O and needs to set up a buffer header that is not associated with the buffer cache.</p> <p>getrbuf () calls kmem_alloc(9F) to perform the memory allocation. kmem_alloc () requires the information included in the <i>sleepflag</i> argument. If <i>sleepflag</i> is set to KM_SLEEP, the driver may sleep until the space is freed up. If <i>sleepflag</i> 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.</p>
<b>RETURN VALUES</b>	getrbuf () returns a pointer to the allocated buffer header, or NULL if no space is available.
<b>CONTEXT</b>	getrbuf () can be called from user or interrupt context. (Drivers must not allow getrbuf () to sleep if called from an interrupt routine.)
<b>SEE ALSO</b>	bioinit(9F), freerbuf(9F), kmem_alloc(9F), kmem_free(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	<code>gld, gld_mac_alloc, gld_mac_free, gld_register, gld_unregister, gld_recv, gld_sched, gld_intr</code> – Generic LAN Driver service routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/gld.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris architecture specific (Solaris DDI).</p> <p><i>macinfo</i>            Pointer to a <code>gld_mac_info(9S)</code> structure.</p> <p><i>dip</i>                 Pointer to <code>dev_info</code> structure.</p> <p><i>name</i>                Device interface name.</p> <p><i>mp</i>                  Pointer to a message block containing a received packet.</p>
<b>DESCRIPTION</b>	<p><code>gld_mac_alloc()</code> allocates a new <code>gld_mac_info(9S)</code> structure and returns a pointer to it. Some of the GLD-private elements of the structure may be initialized before <code>gld_mac_alloc()</code> returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in <code>gld_mac_info(9S)</code>, before passing the <code>macinfo</code> pointer to <code>gld_register()</code>.</p> <p><code>gld_mac_free()</code> frees a <code>gld_mac_info(9S)</code> structure previously allocated by <code>gld_mac_alloc()</code>.</p> <p><code>gld_register()</code> is called from the device driver's <code>attach(9E)</code> routine, and is used to link the GLD-based device driver with the GLD framework. Before calling <code>gld_register()</code> the device driver's <code>attach(9E)</code> routine must first use <code>gld_mac_alloc()</code> to allocate a <code>gld_mac_info(9S)</code> structure, and initialize several of its structure elements. See <code>gld_mac_info(9S)</code> for more information. A successful call to <code>gld_register()</code> performs the following actions:</p> <ul style="list-style-type: none"> <li>■ links the device-specific driver with the GLD system;</li> <li>■ sets the device-specific driver's private data pointer (using <code>ddi_set_driver_private(9F)</code>) to point to the <code>macinfo</code> structure;</li> <li>■ creates the minor device node.</li> </ul> <p>The device interface name passed to <code>gld_register()</code> must exactly match the name of the driver module as it exists in the filesystem.</p>

## gld(9F)

The driver's `attach(9E)` routine should return `DDI_SUCCESS` if `gld_register()` succeeds. If `gld_register()` returns `DDI_FAILURE`, the `attach(9E)` routine should deallocate any resources it allocated before calling `gld_register()` and then also return `DDI_FAILURE`.

`gld_unregister()` is called by the device driver's `detach(9E)` function, and if successful, performs the following tasks:

- ensures the device's interrupts are stopped, calling the driver's `gldm_stop()` routine if necessary;
- removes the minor device node;
- unlinks the device-specific driver from the GLD system.

If `gld_unregister()` returns `DDI_SUCCESS`, the `detach(9E)` routine should deallocate any data structures allocated in the `attach(9E)` routine, using `gld_mac_free()` to deallocate the `macinfo` structure, and return `DDI_SUCCESS`. If `gld_unregister()` returns `DDI_FAILURE`, the driver's `detach(9E)` routine must leave the device operational and return `DDI_FAILURE`.

`gld_recv()` is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a `STREAMS_M_DATA` message containing the raw packet. `gld_recv()` determines which `STREAMS` queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a `DL_UNITDATA_IND` message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to `gld_recv()`. In particular, locks that could be taken by a transmit thread may not be held during a call to `gld_recv()`: the interrupt thread that calls `gld_recv()` may in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's `gldm_send()` routine. If the `gldm_send()` routine were to try to acquire a mutex being held by the `gldm_intr()` routine at the time it calls `gld_recv()`, this could result in a panic due to recursive mutex entry.

`gld_sched()` is called by the device driver to reschedule stalled outbound packets. Whenever the driver's `gldm_send()` routine has returned `GLD_NORESOURCES`, the driver must later call `gld_sched()` to inform the GLD framework that it should retry the packets that previously could not be sent. `gld_sched()` should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's `gldm_send()` routine in a timely way. (If the driver's `gldm_stop()` routine is called, the driver is absolved from this obligation until it later again returns `GLD_NORESOURCES` from its `gldm_send()` routine; however, extra calls to `gld_sched()` will not cause incorrect operation.)

`gld_intr()` is GLD's main interrupt handler. Normally it is specified as the interrupt routine in the device driver's call to `ddi_add_intr(9F)`. The argument to the interrupt handler (specified as `int_handler_arg` in the call to `ddi_add_intr(9F)`) must be a pointer to the `gld_mac_info(9S)` structure. `gld_intr()` will, when appropriate,

call the device driver's `gldm_intr()` function, passing that pointer to the `gld_mac_info(9S)` structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler, and trigger a soft interrupt from within that. In this case, `gld_intr()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

**RETURN VALUES** `gld_mac_alloc()` returns a pointer to a new `gld_mac_info(9S)` structure.

`gld_register()` and `gld_unregister()` return:

`DDI_SUCCESS` on success.

`DDI_FAILURE` on failure.

`gld_intr()` returns a value appropriate for an interrupt handler.

**SEE ALSO** `gld(7D)`, `gld(9E)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `dlpi(7P)`, `attach(9E)`, `ddi_add_intr(9F)`.

*Writing Device Drivers*

## hat\_getkpfnum(9F)

<b>NAME</b>	hat_getkpfnum – get page frame number for kernel address
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  pfnt_t hat_getkpfnum(caddr_t addr);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 2 (DKI only).
<b>PARAMETERS</b>	<i>addr</i> The kernel virtual address for which the page frame number is to be returned.
<b>DESCRIPTION</b>	<p>hat_getkpfnum() returns the page frame number corresponding to the kernel virtual address, <i>addr</i>.</p> <p><i>addr</i> 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.</p>
<b>RETURN VALUES</b>	The page frame number corresponding to the valid virtual address <i>addr</i> . Otherwise the return value is undefined.
<b>CONTEXT</b>	hat_getkpfnum() can be called only from user or kernel context.
<b>SEE ALSO</b>	attach(9E), detach(9E), mmap(9E), ddi_map_regs(9F), ddi_soft_state(9F), ddi_unmap_regs(9F)
	<i>Writing Device Drivers</i>
<b>NOTES</b>	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.

<b>NAME</b>	id32_alloc, id32_free, id32_lookup – 32-bit driver ID management routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/id32.h&gt;  uint32_t id32_alloc(void *ptr, int flag); void id32_free(uint32_t token); void *id32_lookup(uint32_t token);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris architecture specific (Solaris DDI).</p> <p><i>ptr</i> any valid 32- or 64-bit pointer</p> <p><i>flag</i> determines whether caller can sleep for memory (see <code>kmem_alloc(9F)</code> for a description)</p>
<b>DESCRIPTION</b>	<p>These routines were originally developed so that device drivers could manage 64-bit pointers on devices that save space only for 32-bit pointers.</p> <p>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.</p> <p>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 <code>id32_*()</code> routines were created to help drivers translate between 64-bit pointers and a 32-bit token.</p> <p>Given a 32- or 64-bit pointer, the routine <code>id32_alloc()</code> allocates a 32-bit token, returning 0 if <code>KM_NOSLEEP</code> was specified and memory could not be allocated. The allocated token is passed back to <code>id32_lookup()</code> to obtain the original 32- or 64-bit pointer.</p> <p>The routine <code>id32_free()</code> is used to free an allocated token. Once <code>id32_free()</code> is called, the supplied token is no longer valid.</p> <p>Note that these routines have some degree of error checking. This is done so that an invalid token passed to <code>id32_lookup()</code> will not be accepted as valid. When <code>id32_lookup()</code> 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.</p>
<b>CONTEXT</b>	These functions can be called from user or interrupt context. The routine <code>id32_alloc()</code> should not be called from interrupt context when the <code>KM_SLEEP</code> flag is passed in. All other routines can be called from interrupt or kernel context.
<b>SEE ALSO</b>	<p><code>kmem_alloc(9F)</code></p> <p><i>Writing Device Drivers</i></p>

inb(9F)

<b>NAME</b>	inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  unsigned char <b>inb</b>(int <i>port</i>); unsigned short <b>inw</b>(int <i>port</i>); unsigned long <b>inl</b>(int <i>port</i>);  void <b>repinsb</b>(int <i>port</i>, unsigned char *<i>addr</i>, int <i>count</i>); void <b>repinsw</b>(int <i>port</i>, unsigned short *<i>addr</i>, int <i>count</i>); void <b>repinsd</b>(int <i>port</i>, unsigned long *<i>addr</i>, int <i>count</i>);</pre>						
<b>INTERFACE LEVEL</b>	The functions described here are obsolete. For the <code>inb()</code> , <code>inw()</code> , and <code>inl()</code> functions, use, respectively, <code>ddi_get8(9F)</code> , <code>ddi_get16(9F)</code> , and <code>ddi_get32(9F)</code> instead. For <code>repinsb()</code> , <code>repinsw()</code> , and <code>repinsd()</code> , use, respectively, <code>ddi_rep_get8(9F)</code> , <code>ddi_rep_get16(9F)</code> , and <code>ddi_rep_get32(9F)</code> instead.						
<b>PARAMETERS</b>	<table><tr><td><i>port</i></td><td>A valid I/O port address.</td></tr><tr><td><i>addr</i></td><td>The address of a buffer where the values will be stored.</td></tr><tr><td><i>count</i></td><td>The number of values to be read from the I/O port.</td></tr></table>	<i>port</i>	A valid I/O port address.	<i>addr</i>	The address of a buffer where the values will be stored.	<i>count</i>	The number of values to be read from the I/O port.
<i>port</i>	A valid I/O port address.						
<i>addr</i>	The address of a buffer where the values will be stored.						
<i>count</i>	The number of values to be read from the I/O port.						
<b>DESCRIPTION</b>	<p>These routines read data of various sizes from the I/O port with the address specified by <i>port</i>.</p> <p>The <code>inb()</code>, <code>inw()</code>, and <code>inl()</code> functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.</p> <p>The <code>repinsb()</code>, <code>repinsw()</code>, and <code>repinsd()</code> functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. <i>count</i> 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 <i>count</i> values of the requested size.</p>						
<b>RETURN VALUES</b>	<code>inb()</code> , <code>inw()</code> , and <code>inl()</code> return the value that was read from the I/O port.						
<b>CONTEXT</b>	These functions may be called from user or interrupt context.						
<b>ATTRIBUTES</b>	See <code>attributes(5)</code> for descriptions of the following attributes:						
	<table border="1"><thead><tr><th>ATTRIBUTE TYPE</th><th>ATTRIBUTE VALUE</th></tr></thead><tbody><tr><td>Architecture</td><td>x86</td></tr><tr><td>Stability Level</td><td>Obsolete</td></tr></tbody></table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Architecture	x86	Stability Level	Obsolete
ATTRIBUTE TYPE	ATTRIBUTE VALUE						
Architecture	x86						
Stability Level	Obsolete						
<b>SEE ALSO</b>	<code>eisa(4)</code> , <code>isa(4)</code> , <code>attributes(5)</code> , <code>ddi_get8(9F)</code> , <code>ddi_get16(9F)</code> , <code>ddi_get32(9F)</code> , <code>ddi_rep_get8(9F)</code> , <code>ddi_rep_get16(9F)</code> , <code>ddi_rep_get32(9F)</code> , <code>outb(9F)</code>						

inb(9F)

*Writing Device Drivers*

## insq(9F)

<b>NAME</b>	insq – insert a message into a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>insq</b>(queue_t *q, mblk_t *emp, mblk_t *nmp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>q</i> Pointer to the queue containing message <i>emp</i> . <i>emp</i> Enqueued message before which the new message is to be inserted. mblk_t is an instance of the msgb(9S) structure. <i>nmp</i> Message to be inserted.
<b>DESCRIPTION</b>	insq() inserts a message into a queue. The message to be inserted, <i>nmp</i> , is placed in <i>q</i> immediately before the message <i>emp</i> . If <i>emp</i> 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.
<b>RETURN VALUES</b>	insq() returns 1 on success, and 0 on failure.
<b>CONTEXT</b>	insq() can be called from user or interrupt context.
<b>EXAMPLES</b>	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. <pre>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-&gt;b_datap-&gt;db_type) { 12     case M_PROTO: 13         tp = (union T_primitives *)mp-&gt;b_rptr; 14         switch (tp-&gt;type) { 15         case T_DATA_REQ: 16             putq(q, mp); 17             break; 18 19         case T_EXDATA_REQ: 20             /* Insert code here to protect queue and message block */ 21             for (bp = q-&gt;q_first; bp = bp-&gt;b_next) { 22                 if (bp-&gt;b_datap-&gt;db_type == M_PROTO) { 23                     ntp = (union T_primitives *)bp-&gt;b_rptr;</pre>

```

24             if (ntp->type != T_EXDATA_REQ)
25                 break;
26             }
27         }
28         (void)insq(q, bp, mp);
29         /* End of region that must be protected */
30         break;
31     . . .
32 }
33 }

```

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.

## IOC\_CONVERT\_FROM(9F)

<b>NAME</b>	IOC_CONVERT_FROM – determine if there is a need to translate M_IOCTL contents.
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  uint_t IOC_CONVERT_FROM(struct iocblk *iopc);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI Specific (Solaris DDI)
<b>PARAMETERS</b>	<i>iopc</i> A pointer to the M_IOCTL control structure.
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	IOC_CONVERT_FROM() 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.
<b>CONTEXT</b>	IOC_CONVERT_FROM() can be called from user or interrupt context.
<b>SEE ALSO</b>	ddi_model_convert_from(9F)  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>

<b>NAME</b>	kmem_alloc, kmem_zalloc, kmem_free – allocate kernel memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kmem.h&gt;  void *kmem_alloc(size_t size, int flag); void *kmem_zalloc(size_t size, int flag); void kmem_free(void*buf, size_t size);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>size</i>                    Number of bytes to allocate.</p> <p><i>flag</i>                    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.</p> <p><i>buf</i>                     Pointer to allocated memory.</p>
<b>DESCRIPTION</b>	<p>kmem_alloc() allocates <i>size</i> 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. <i>flag</i> 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.</p> <p>kmem_zalloc() is like kmem_alloc() but returns zero-filled memory.</p> <p>kmem_free() frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.</p>
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	kmem_alloc() and kmem_zalloc() can be called from interrupt context only if the KM_NOSLEEP flag is set. They can be called from user context with any valid <i>flag</i> . kmem_free() can be called from user or interrupt context.
<b>SEE ALSO</b>	copyout(9F), freerbuf(9F), getrbuf(9F)
<b>WARNINGS</b>	<i>Writing Device Drivers</i>
<b>WARNINGS</b>	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.

## kmem\_alloc(9F)

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.

<b>NAME</b>	kmem_cache_create, kmem_cache_alloc, kmem_cache_free, kmem_cache_destroy – kernel memory cache allocator operations
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kmem.h&gt;  kmem_cache_t *kmem_cache_create(char *name, size_t bufsize, size_t     align, int (*constructor)(void *, void *, int), void     (*destructor)(void *, void *), void (*reclaim)(void *), void     *private, void *vmp, int cflags);  void kmem_cache_destroy(kmem_cache_t *cp);  void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);  void kmem_cache_free(kmem_cache_t *cp, void *obj);  [Synopsis for callback functions:]  int (*constructor)(void *buf, void *un, int kmflags);  void (*destructor)(void *buf, void *un);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<p>The parameters for the kmem_cache_* functions are as follows:</p> <p><i>name</i> Descriptive name of a kstat(9S) structure of class kmem_cache. Only alphanumeric characters can be used in <i>name</i>.</p> <p><i>bufsize</i> Size of the objects it manages.</p> <p><i>align</i> Required object alignment.</p> <p><i>constructor</i> Pointer to an object constructor function. Parameters are defined below.</p> <p><i>destructor</i> Pointer to an object destructor function. Parameters are defined below.</p> <p><i>reclaim</i> Drivers should pass NULL.</p> <p><i>private</i> Pass-through argument for constructor/destructor.</p> <p><i>vmp</i> Drivers should pass NULL.</p> <p><i>cflags</i> Drivers must pass 0.</p>

## kmem\_cache\_create(9F)

*kmflag*

Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.

*\*obj*

Pointer to the object allocated by `kmem_cache_alloc()`.

The parameters for the callback constructor function are as follows:

`void *buf`

Pointer to the object to be constructed.

`void *un`

The *private* parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

`int kmflags`

Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.

The parameters for the callback destructor function are as follows:

`void *buf`

Pointer to the object to be deconstructed.

`void *un`

The *private* parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

### DESCRIPTION

In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object's initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of `mutex_destroy()` and `mutex_init()` each time. An object's embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state. The essential requirement is that the client must free the object (using `kmem_cache_free()`) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.

A driver should call `kmem_cache_create()` at the time of `_init(9E)` or `attach(9E)`, and call the corresponding `kmem_cache_destroy()` at the time of `_fini(9E)` or `detach(9E)`.

`kmem_cache_create()` creates a cache of objects, each of size *size* bytes, aligned on an *align* boundary. Drivers not requiring a specific alignment can pass 0. *name* identifies the cache for statistics and debugging. *constructor* and *destructor* convert plain memory into objects and back again; *constructor* can fail if it needs to allocate memory but cannot. *private* is a parameter passed to the constructor and destructor callbacks to support parameterized caches (for example, a pointer to an instance of the driver's soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat(9S)` structure of class `kmem_cache` and name *name*. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. *kmflag* is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it is acceptable to wait for memory if none is currently available.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

**CONTEXT** Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

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

**EXAMPLES** **EXAMPLE 1** Object Caching

Consider the following data structure:

```
struct foo {
    kmutex_t foo_lock;
    kcondvar_t foo_cv;
    struct bar *foo_barlist;
    int foo_refcnt;
};
```

## kmem\_cache\_create(9F)

### EXAMPLE 1 Object Caching (Continued)

Assume that a `foo` structure cannot be freed until there are no outstanding references to it (`foo_refcnt == 0`) and all of its pending bar events (whatever they are) have completed (`foo_barlist == NULL`). The life cycle of a dynamically allocated `foo` would be something like this:

```
foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;
    use foo;
ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a `foo` object we perform a sequence of operations that constitutes nothing more expensive overhead. All of this overhead (that is, everything other than `use foo` above) can be eliminated by object caching.

```
int
foo_constructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}

void
foo_destructor(void *buf, void *arg, int size)
{
    struct foo *foo = buf;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}

un = ddi_get_soft_state(foo_softc, instance);
(void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache",
    ddi_get_instance(dip));
foo_cache = kmem_cache_create(buf,
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor,
    NULL, un, 0);
```

To allocate, use, and free a `foo` object:

```
foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
    use foo;
kmem_cache_free(foo_cache, foo);
```

**EXAMPLE 1** Object Caching (Continued)

This makes `foo` allocation fast, because the allocator will usually do nothing more than fetch an already-constructed `foo` from the cache. `foo_constructor` and `foo_destructor` will be invoked only to populate and drain the cache, respectively.

**RETURN VALUES**

If successful, the constructor function must return a pointer to the allocated object. If `KM_NOSLEEP` is set and memory cannot be allocated without sleeping, the constructor must return `NULL`.

`kmem_cache_create()` returns a pointer to the allocated cache. If the `name` parameter contains non-alphanumeric characters, `kmem_cache_create()` returns `NULL`.

If successful, `kmem_cache_alloc()` returns a pointer to the allocated object. If `KM_NOSLEEP` is set and memory cannot be allocated without sleeping, `kmem_cache_alloc()` returns `NULL`.

**ATTRIBUTES**

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

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).

## kstat\_create(9F)

<b>NAME</b>	kstat_create – create and initialize a new kstat
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kstat.h&gt;  kstat_t *kstat_create(char *module, int instance, char *name, char     *class, uchar_t type, ulong_t ndata, uchar_t ks_flag);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)
<i>module</i>	The name of the provider's module (such as "sd", "esp", ...). The "core" kernel uses the name "unix".
<i>instance</i>	The provider's instance number, as from ddi_get_instance(9F). Modules which do not have a meaningful instance number should use 0.
<i>name</i>	A pointer to a string that uniquely identifies this structure. Only KSTAT_STRLLEN – 1 characters are significant.
<i>class</i>	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.
<i>type</i>	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
<i>ndata</i>	The number of type-specific data records to allocate.
<i>flag</i>	A bit-field of various flags for this kstat. <i>flag</i> 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.

<b>DESCRIPTION</b>	<p><code>kstat_create()</code> is used in conjunction with <code>kstat_install(9F)</code> to allocate and initialize a <code>kstat(9S)</code> structure. The method is generally as follows:</p> <p><code>kstat_create()</code> allocates and performs necessary system initialization of a <code>kstat(9S)</code> structure. <code>kstat_create()</code> allocates memory for the entire <code>kstat</code> (header plus data), initializes all header fields, initializes the data section to all zeroes, assigns a unique <code>kstat</code> ID (KID), and puts the <code>kstat</code> onto the system's <code>kstat</code> chain. The returned <code>kstat</code> is marked invalid because the provider (caller) has not yet had a chance to initialize the data section.</p> <p>After a successful call to <code>kstat_create()</code> the driver must perform any necessary initialization of the data section (such as setting the name fields in a <code>kstat</code> of type <code>KSTAT_TYPE_NAMED</code>). Virtual <code>kstats</code> must have the <code>ks_data</code> field set at this time. The provider may also set the <code>ks_update</code>, <code>ks_private</code>, and <code>ks_lock</code> fields if necessary.</p> <p>Once the <code>kstat</code> is completely initialized, <code>kstat_install(9F)</code> is used to make the <code>kstat</code> accessible to the outside world.</p>
<b>RETURN VALUES</b>	If successful, <code>kstat_create()</code> returns a pointer to the allocated <code>kstat</code> . <code>NULL</code> is returned upon failure.
<b>CONTEXT</b>	<code>kstat_create()</code> can be called from user or kernel context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Allocating and Initializing a <code>kstat</code> Structure</p> <pre>pkstat_t    *ksp; ksp = kstat_create(module, instance, name, class, type, ndata, flags); if (ksp) {     /* ... provider initialization, if necessary */     kstat_install(ksp); }</pre>
<b>SEE ALSO</b>	<p><code>kstat(3KSTAT)</code>, <code>ddi_get_instance(9F)</code>, <code>kstat_delete(9F)</code>, <code>kstat_install(9F)</code>, <code>kstat_named_init(9F)</code>, <code>kstat(9S)</code>, <code>kstat_named(9S)</code></p> <p><i>Writing Device Drivers</i></p>

## kstat\_delete(9F)

<b>NAME</b>	kstat_delete – remove a kstat from the system
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kstat.h&gt;  void <b>kstat_delete</b>(kstat_t *ksp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<i>ksp</i> Pointer to a currently installed kstat(9S) structure.
<b>DESCRIPTION</b>	kstat_delete() removes <i>ksp</i> from the kstat chain and frees all associated system resources.
<b>RETURN VALUES</b>	None.
<b>CONTEXT</b>	kstat_delete() can be called from any context.
<b>SEE ALSO</b>	kstat_create(9F), kstat_install(9F), kstat_named_init(9F), kstat(9S) <i>Writing Device Drivers</i>
<b>NOTES</b>	When calling kstat_delete(), the driver must not be holding that kstat's ks_lock. Otherwise, it may deadlock with a kstat reader.

<b>NAME</b>	kstat_install – add a fully initialized kstat to the system
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kstat.h&gt;  void <b>kstat_install</b>(kstat_t *ksp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<i>ksp</i> Pointer to a fully initialized kstat(9S) structure.
<b>DESCRIPTION</b>	<p>kstat_install() is used in conjunction with kstat_create(9F) to allocate and initialize a kstat(9S) structure.</p> <p>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.</p> <p>Once the kstat is completely initialized, kstat_install is used to make the kstat accessible to the outside world.</p>
<b>RETURN VALUES</b>	None.
<b>CONTEXT</b>	kstat_install() can be called from user or kernel context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Allocating and Initializing a kstat Structure</p> <p>The method for allocating and initializing a kstat structure is generally as follows:</p> <pre>kstat_t *ksp; ksp = kstat_create(module, instance, name, class, type, ndata, flags); if (ksp) {     /* ... provider initialization, if necessary */     kstat_install(ksp); }</pre>
<b>SEE ALSO</b>	kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S)
	<i>Writing Device Drivers</i>

## kstat\_named\_init(9F)

<b>NAME</b>	kstat_named_init, kstat_named_setstr – initialize a named kstat
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kstat.h&gt;  void <b>kstat_named_init</b>(kstat_named_t *knp, char *name, uchar_t     data_type);  void <b>kstat_named_setstr</b>(kstat_named_t *knp, const char *str);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)
	<i>knp</i> Pointer to a kstat_named(9S) structure.
	<i>name</i> The name of the statistic.
	<i>data_type</i> The type of value. This indicates which field of the kstat_named(9S) structure should be used. Valid values are:
	KSTAT_DATA_CHAR The "char" field.
	KSTAT_DATA_LONG The "long" field.
	KSTAT_DATA_ULONG The "unsigned long" field.
	KSTAT_DATA_LONGLONG The "long long" field.
	KSTAT_DATA_ULONGLONG The "unsigned long long" field.
	KSTAT_DATA_STRING Arbitrary length "long string" field.
	<i>str</i> Pointer to a NULL-terminated string.
<b>DESCRIPTION</b>	<p>kstat_named_init() associates a name and a type with a kstat_named(9S) structure.</p> <p>kstat_named_setstr() associates <i>str</i> with the named kstat knp. It is an error for knp to be of type other than KSTAT_DATA_STRING. This is the only supported method of changing the value of long strings.</p>
<b>RETURN VALUES</b>	None.
<b>CONTEXT</b>	kstat_named_init() and kstat_named_setstr() can be called from user or kernel context.
<b>SEE ALSO</b>	kstat_create(9F), kstat_install(9F), kstat(9S), kstat_named(9S) <i>Writing Device Drivers</i>

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/kstat.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI)</p> <p><i>kiop</i>                      Pointer to a kstat_io(9S) structure.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>The <code>kstat_queue()</code> family of functions manage these times based on the transitions between the driver wait queue and run queue.</p> <p><code>kstat_waitq_enter()</code>  <code>kstat_waitq_enter()</code> should be called when a request arrives and is placed into a pre-service state (such as just prior to calling <code>disksort(9F)</code>).</p> <p><code>kstat_waitq_exit()</code>  <code>kstat_waitq_exit()</code> should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's start routine).</p> <p><code>kstat_runq_enter()</code>  <code>kstat_runq_enter()</code> is also called when a request is placed in its service state (just prior to calling the driver's start routine, but after <code>kstat_waitq_exit()</code>).</p> <p><code>kstat_runq_exit()</code>  <code>kstat_runq_exit()</code> is used when a request is removed from its service state (just prior to calling <code>biodone(9F)</code>).</p> <p><code>kstat_waitq_to_runq()</code>  <code>kstat_waitq_to_runq()</code> transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a <code>kstat_waitq_exit()</code> followed by a call to <code>kstat_runq_enter()</code>.</p> <p><code>kstat_runq_back_to_waitq()</code>  <code>kstat_runq_back_to_waitq()</code> transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an</p>

kstat\_queue(9F)

example).

**RETURN VALUES** None.

**CONTEXT** `kstat_create()` can be called from user or kernel context.

**WARNINGS** These transitions must be protected by holding the `kstat`'s `ks_lock`, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for 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*

<b>NAME</b>	linkb – concatenate two message blocks
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; void linkb(mblk_t *mp1, mblk_t *mp2);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>DESCRIPTION</b>	<p>linkb() creates a new message by adding <i>mp2</i> to the tail of <i>mp1</i>. The continuation pointer, <i>b_cont</i>, of <i>mp1</i> is set to point to <i>mp2</i>.</p> <p>The following figure describes how the linkb(<i>m1</i>, <i>m2</i>); function concatenates two message blocks, <i>mp1</i> and <i>mp2</i>:</p> <pre> graph LR     subgraph mp1         direction TB         mp1_datap[b_datap]         mp1_cont[b_cont]     end     subgraph mp2         direction TB         mp2_datap[b_datap]         mp2_cont[b_cont (0)]     end     subgraph db_base1 [db_base]         db1[db_base]     end     subgraph db_base2 [db_base]         db2[db_base]     end     subgraph data_buffer1 [data buffer]         db1 --&gt; data1[data buffer]     end     subgraph data_buffer2 [data buffer]         db2 --&gt; data2[data buffer]     end     mp1_datap --&gt; db1     mp1_cont --&gt; mp2_datap     mp2_datap --&gt; db2   </pre> <p style="text-align: center;"><b>linkb(<i>mp1</i>, <i>mp2</i>);</b></p>
<b>PARAMETERS</b>	<p><i>mp1</i>      The message to which <i>mp2</i> is to be added. mblk_t is an instance of the msgb(9S) structure.</p> <p><i>mp2</i>      The message to be added.</p>
<b>CONTEXT</b>	linkb() can be called from user or interrupt context.
<b>EXAMPLES</b>	See dupb(9F) for an example that uses linkb().
<b>SEE ALSO</b>	<p>dupb(9F), unlinkb(9F), msgb(9S)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## makecom(9F)

<b>NAME</b>	makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 – make a packet for SCSI commands														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>makecom_g0</b>(struct scsi_pkt *pkt, struct scsi_device *devp, int     flag, int cmd, int addr, int cnt);  void <b>makecom_g0_s</b>(struct scsi_pkt *pkt, struct scsi_device *devp,     int flag, int cmd, int cnt, int fixbit);  void <b>makecom_g1</b>(struct scsi_pkt *pkt, struct scsi_device *devp, int     flag, int cmd, int addr, int cnt);  void <b>makecom_g5</b>(struct scsi_pkt *pkt, struct scsi_device *devp, int     flag, int cmd, int addr, int cnt);</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	<p>These interfaces are obsolete. <code>scsi_setup_cdb(9F)</code> should be used instead.</p> <table><tr><td><i>pkt</i></td><td>Pointer to an allocated <code>scsi_pkt(9S)</code> structure.</td></tr><tr><td><i>devp</i></td><td>Pointer to the target's <code>scsi_device(9S)</code> structure.</td></tr><tr><td><i>flag</i></td><td>Flags for the <code>pkt_flags</code> member.</td></tr><tr><td><i>cmd</i></td><td>First byte of a group 0 or 1 or 5 SCSI CDB.</td></tr><tr><td><i>addr</i></td><td>Pointer to the location of the data.</td></tr><tr><td><i>cnt</i></td><td>Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.</td></tr><tr><td><i>fixbit</i></td><td>Fixed bit in sequential access device commands.</td></tr></table>	<i>pkt</i>	Pointer to an allocated <code>scsi_pkt(9S)</code> structure.	<i>devp</i>	Pointer to the target's <code>scsi_device(9S)</code> structure.	<i>flag</i>	Flags for the <code>pkt_flags</code> member.	<i>cmd</i>	First byte of a group 0 or 1 or 5 SCSI CDB.	<i>addr</i>	Pointer to the location of the data.	<i>cnt</i>	Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.	<i>fixbit</i>	Fixed bit in sequential access device commands.
<i>pkt</i>	Pointer to an allocated <code>scsi_pkt(9S)</code> structure.														
<i>devp</i>	Pointer to the target's <code>scsi_device(9S)</code> structure.														
<i>flag</i>	Flags for the <code>pkt_flags</code> member.														
<i>cmd</i>	First byte of a group 0 or 1 or 5 SCSI CDB.														
<i>addr</i>	Pointer to the location of the data.														
<i>cnt</i>	Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.														
<i>fixbit</i>	Fixed bit in sequential access device commands.														
<b>DESCRIPTION</b>	<p><code>makecom</code> functions initialize a packet with the specified command descriptor block, <i>devp</i> and transport flags. The <code>pkt_address</code>, <code>pkt_flags</code>, and the command descriptor block pointed to by <code>pkt_cdbp</code> are initialized using the remaining arguments. Target drivers may use <code>makecom_g0()</code> for Group 0 commands (except for sequential access devices), or <code>makecom_g0_s()</code> for Group 0 commands for sequential access devices, or <code>makecom_g1()</code> for Group 1 commands, or <code>makecom_g5()</code> for Group 5 commands. <i>fixbit</i> is used by sequential access devices for accessing fixed block sizes and sets the the tag portion of the SCSI CDB.</p>														
<b>CONTEXT</b>	These functions can be called from user or interrupt context.														
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <code>makecom</code> Functions</p> <pre>if (blkno &gt;= (1&lt;&lt;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); }</pre>														

**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).

## madevice(9F)

<b>NAME</b>	madevice – make device number from major and minor numbers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/mkdev.h&gt; #include &lt;sys/ddi.h&gt;  dev_t <b>madevice</b>(major_t <i>majnum</i>, minor_t <i>minnum</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>majnum</i> Major device number. <i>minnum</i> Minor device number.
<b>DESCRIPTION</b>	<code>madevice()</code> creates a device number from a major and minor device number. <code>madevice()</code> should be used to create device numbers so the driver will port easily to releases that treat device numbers differently.
<b>RETURN VALUES</b>	The device number, containing both the major number and the minor number, is returned. No validation of the major or minor numbers is performed.
<b>CONTEXT</b>	<code>madevice()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>getmajor(9F)</code> , <code>getminor(9F)</code>

<b>NAME</b>	max – return the larger of two integers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; int <b>max</b>(int <i>int1</i>, int <i>int2</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>int1</i> The first integer. <i>int2</i> The second integer.
<b>DESCRIPTION</b>	max() compares two signed integers and returns the larger of the two.
<b>RETURN VALUES</b>	The larger of the two numbers.
<b>CONTEXT</b>	max() can be called from user or interrupt context.
<b>SEE ALSO</b>	min(9F) <i>Writing Device Drivers</i>

min(9F)

<b>NAME</b>	min – return the lesser of two integers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; int <b>min</b>(int <i>int1</i>, int <i>int2</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>int1</i> The first integer. <i>int2</i> The second integer.
<b>DESCRIPTION</b>	min() compares two signed integers and returns the lesser of the two.
<b>RETURN VALUES</b>	The lesser of the two integers.
<b>CONTEXT</b>	min() can be called from user or interrupt context.
<b>SEE ALSO</b>	max(9F) <i>Writing Device Drivers</i>

<b>NAME</b>	mkiocb – allocates a STREAMS ioctl block for M_IOCTL messages in the kernel.
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *mkiocb (uint_t command) ;</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>command</i> ioctl command for the <i>ioc_cmd</i> field.
<b>DESCRIPTION</b>	<p>STREAMS modules or drivers might need to issue an ioctl to a lower module or driver. The <code>mkiocb()</code> function tries to allocate (using <code>allocb(9F)</code>) a STREAMS M_IOCTL message block (<code>iocblk(9S)</code>). Buffer allocation fails only when the system is out of memory. If no buffer is available, the <code>qbufcall(9F)</code> function can help a module recover from an allocation failure.</p> <p>The <code>mkiocb</code> function returns a <code>mblk_t</code> structure which is large enough to hold any of the ioctl messages (<code>iocblk(9S)</code>, <code>copyreq(9S)</code> or <code>copyresp(9S)</code>), and has the following special properties:</p> <pre>b_wptr                    Set to b_rptr + sizeof(struct iocblk). b_cont                    Set to NULL. b_datap-&gt;db_type         Set to M_IOCTL.</pre> <p>The fields in the <code>iocblk</code> structure are initialized as follows:</p> <pre>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.</pre>
<b>RETURN VALUES</b>	<p>Upon success, the <code>mkiocb()</code> function returns a pointer to the allocated <code>mblk_t</code> of type M_IOCTL.</p> <p>On failure, it returns a null pointer.</p>
<b>CONTEXT</b>	The <code>mkiocb()</code> function can be called from user or interrupt context.

## mkiocb(9F)

### 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.

```
1 test_lrsrv(queue_t *q)
2 {
3     ...
4
5     switch (DB_TYPE(mp)) {
6     case M_IOCACK:
7     case M_IOCNAK:
8         /* Does this match the ioctl that this module sent */
9         ioc = (struct iocblk*)mp->b_rptr;
10        if (ioc->ioc_id == testinfo->xx_iocid) {
11            /* matches, so process the message */
12            ...
13            freemsg(mp);
14        }
15        break;
16    }
17    ...
18 }
```

#### EXAMPLE 3 An iocblk Allocation Which Fails

The next example shows an iocblk allocation which fails. Since the open routine is in user context, the caller may block using qbufcall(9F) until memory is available.

```
1 test_open(queue_t *q, dev_t devp, int oflag, int sflag, cred_t *credp)
2 {
3     while ((mp = mkiocb(TEST_IOCTL)) == NULL) {
4         int id;
5     }
```

**EXAMPLE 3** An iocblk Allocation Which Fails *(Continued)*

```

6         id = qbufcall(q, sizeof (union ioctypes), BPRI_HI,
7             dummy_callback, 0);
8         /* Handle interrupts */
9         if (!qwait_sig(q)) {
10            qunbufcall(q, id);
11            return (EINTR);
12        }
13    }
14    putnext(q, mp);
15 }

```

**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.

## mod\_install(9F)

<b>NAME</b>	mod_install, mod_remove, mod_info – add, remove or query a loadable module
<b>SYNOPSIS</b>	<pre>#include &lt;sys/modctl.h&gt;  int mod_install(struct modlinkage *modlinkage); int mod_remove(struct modlinkage *modlinkage); int mod_info(struct modlinkage *modlinkage, struct modinfo *modinfo);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>modlinkage</i>            Pointer to the loadable module's modlinkage structure which describes what type(s) of module elements are included in this loadable module.</p> <p><i>modinfo</i>                Pointer to the modinfo structure passed to _info(9E).</p>
<b>DESCRIPTION</b>	<p>mod_install() must be called from a module's _init(9E) routine.</p> <p>mod_remove() must be called from a module's _fini(9E) routine.</p> <p>mod_info() must be called from a module's _info(9E) routine.</p>
<b>RETURN VALUES</b>	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.
<b>EXAMPLES</b>	See _init(9E) for an example that uses these functions.
<b>SEE ALSO</b>	<i>_fini(9E), _info(9E), _init(9E), modldrv(9S), modlinkage(9S), modlstrmod(9S)</i> <i>Writing Device Drivers</i>

<b>NAME</b>	msgdsize – return the number of bytes in a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  size_t msgdsize(mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>mp</i> Message to be evaluated.
<b>DESCRIPTION</b>	msgdsize() counts the number of bytes in a data message. Only bytes included in the data blocks of type M_DATA are included in the count.
<b>RETURN VALUES</b>	The number of data bytes in a message, expressed as an integer.
<b>CONTEXT</b>	msgdsize() can be called from user or interrupt context.
<b>EXAMPLES</b>	See bufcall(9F) for an example that uses msgdsize().
<b>SEE ALSO</b>	bufcall(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

## msgpullup(9F)

<b>NAME</b>	msgpullup – concatenate bytes in a message				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  mblk_t *msgpullup(mblk_t *mp, ssize_t len);</pre>				
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).				
<b>PARAMETERS</b>	<p><i>mp</i>            Pointer to the message whose blocks are to be concatenated.</p> <p><i>len</i>            Number of bytes to concatenate.</p>				
<b>DESCRIPTION</b>	msgpullup() concatenates and aligns the first <i>len</i> data bytes of the message pointed to by <i>mp</i> , 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 <i>len</i> equals -1, all data are concatenated. If <i>len</i> bytes of the same message type cannot be found, msgpullup() fails and returns NULL.				
<b>RETURN VALUES</b>	msgpullup returns the following values: <table><tr><td>Non-null</td><td>Successful completion. A pointer to the new message is returned.</td></tr><tr><td>NULL</td><td>An error occurred.</td></tr></table>	Non-null	Successful completion. A pointer to the new message is returned.	NULL	An error occurred.
Non-null	Successful completion. A pointer to the new message is returned.				
NULL	An error occurred.				
<b>CONTEXT</b>	msgpullup() can be called from user or interrupt context.				
<b>SEE ALSO</b>	srv(9E), allocb(9F), pullupmsg(9F), msgb(9S)  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>				
<b>NOTES</b>	msgpullup() is a DKI-compliant replacement for the older pullupmsg(9F) routine. Users are strongly encouraged to use msgpullup() instead of pullupmsg(9F).				

<b>NAME</b>	mt-streams – STREAMS multithreading								
<b>SYNOPSIS</b>	<code>#include &lt;sys/conf.h&gt;</code>								
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).								
<b>DESCRIPTION</b>	<p>STREAMS drivers configures the degree of concurrency using the <code>cb_flag</code> field in the <code>cb_ops</code> structure (see <code>cb_ops(9S)</code>). The corresponding field for STREAMS modules is the <code>f_flag</code> in the <code>fmodsw</code> structure.</p> <p>For the purpose of restricting and controlling the concurrency in drivers/modules, we define the concepts of <i>inner</i> and <i>outer perimeters</i>. 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 (<code>open(9E)</code>, <code>close(9E)</code>, <code>put(9E)</code>, or <code>srv(9E)</code>).</p> <p>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).</p> <p>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.</p> <p>All MT safe modules/drivers specify the <code>D_MP</code> flag.</p>								
<b>Inner Perimeter Flags</b>	<p>The inner perimeter presence and scope are controlled by the mutually exclusive flags:</p> <table border="0"> <tr> <td style="padding-right: 20px;"><code>D_MTPERQ</code></td> <td>The module/driver has an inner perimeter around each queue.</td> </tr> <tr> <td><code>D_MTQPAIR</code></td> <td>The module/driver has an inner perimeter around each read/write pair of queues.</td> </tr> <tr> <td><code>D_MTPERMOD</code></td> <td>The module/driver has an inner perimeter that encloses all the module's/driver's queues.</td> </tr> <tr> <td>None of the above</td> <td>The module/driver has no inner perimeter.</td> </tr> </table>	<code>D_MTPERQ</code>	The module/driver has an inner perimeter around each queue.	<code>D_MTQPAIR</code>	The module/driver has an inner perimeter around each read/write pair of queues.	<code>D_MTPERMOD</code>	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.
<code>D_MTPERQ</code>	The module/driver has an inner perimeter around each queue.								
<code>D_MTQPAIR</code>	The module/driver has an inner perimeter around each read/write pair of queues.								
<code>D_MTPERMOD</code>	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.								
<b>Outer Perimeter Flags</b>	<p>The outer perimeter presence is configured using:</p> <table border="0"> <tr> <td style="padding-right: 20px;"><code>D_MTOUTPERIM</code></td> <td>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 <code>D_MTPERMOD</code>.</td> </tr> </table> <p>Note that acquiring exclusive access at the outer perimeter (that is, using <code>qwriter(9F)</code> with the <code>PERIM_OUTER</code> flag) can incur significant performance penalties, which grow linearly with the number of open instances of the module or driver in the system.</p>	<code>D_MTOUTPERIM</code>	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 <code>D_MTPERMOD</code> .						
<code>D_MTOUTPERIM</code>	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 <code>D_MTPERMOD</code> .								

## mt-streams(9F)

The default concurrency can be modified using:

<code>D_MTPUTSHARED</code>	This flag modifies the default behavior when <code>put(9E)</code> procedure are invoked so that the inner perimeter is entered shared instead of exclusively.
<code>D_MTOEXCL</code>	This flag modifies the default behavior when <code>open(9E)</code> and <code>close(9E)</code> procedures are invoked so the 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 <code>open(9E)</code> and <code>close(9E)</code> both execute with user context, developers are encouraged to use traditional synchronization routines such as <code>cv_wait_sig(9F)</code> 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*

<b>NAME</b>	mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ksynch.h&gt;  void <b>mutex_init</b>(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);  void <b>mutex_destroy</b>(kmutex_t *mp);  void <b>mutex_enter</b>(kmutex_t *mp);  void <b>mutex_exit</b>(kmutex_t *mp);  int <b>mutex_owned</b>(kmutex_t *mp);  int <b>mutex_tryenter</b>(kmutex_t *mp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>mp</i> Pointer to a kernel mutex lock (kmutex_t).</p> <p><i>name</i> Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)</p> <p><i>type</i> Type of mutex lock.</p> <p><i>arg</i> Type-specific argument for initialization routine.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>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.</p> <p>mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The <i>type</i> argument should be set to MUTEX_DRIVER.</p> <p><i>arg</i> 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 <i>arg</i> should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). Note that <i>arg</i> should be the value of the iblock cookie casted to (void *), not the address of the cookie. The arguments passed to ddi_get_iblock_cookie(9F) and ddi_get_soft_iblock_cookie(9F), on the other hand, are the addresses of the cookie. If the mutex is never used inside an interrupt handler, the argument should be NULL.</p>

## mutex(9F)

`mutex_enter()` is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic 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 of failure.

`mutex_owned()` returns non-zero if the calling thread currently holds the mutex pointed to by `mp`, or when that cannot be determined, if any thread holds the mutex. `mutex_owned()` returns zero.

### CONTEXT

These functions can be called from user, kernel, or high-level interrupt context, except for `mutex_init()` and `mutex_destroy()`, which can be called from user or kernel context only.

### EXAMPLES

#### EXAMPLE 1 Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

```
ddi_get_iblock_cookie(dip, 0, &iblock);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
          (void *)iblock);
ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
            (caddr_t)un);
```

#### EXAMPLE 2 Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following `ASSERT`:

```
xxstart(struct xxunit *un)
{
    ASSERT(mutex_owned(&un->un_lock));
    ...
}
```

**EXAMPLE 2** Calling a Routine with a Lock (Continued)

**SEE ALSO** lockstat(1M), Intro(9F), condvar(9F), ddi\_add\_intr(9F), ddi\_get\_iblock\_cookie(9F), ddi\_get\_soft\_iblock\_cookie(9F), rwlock(9F), semaphore(9F)

*Writing Device Drivers*

**NOTES** Compiling with `_LOCKTEST` or `_MPSTATS` defined has no effect. To gather lock statistics, see `lockstat(1M)`.

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.

## nochpoll(9F)

<b>NAME</b>	nochpoll – error return function for non-pollable devices										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>nochpoll</b>(dev_t dev, short events, int anyyet, short *reventsp,               struct pollhead **pollhdrp);</pre>										
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).										
<b>PARAMETERS</b>	<table><tr><td><i>dev</i></td><td>Device number.</td></tr><tr><td><i>events</i></td><td>Event flags.</td></tr><tr><td><i>anyyet</i></td><td>Check current events only.</td></tr><tr><td><i>reventsp</i></td><td>Event flag pointer.</td></tr><tr><td><i>pollhdrp</i></td><td>Poll head pointer.</td></tr></table>	<i>dev</i>	Device number.	<i>events</i>	Event flags.	<i>anyyet</i>	Check current events only.	<i>reventsp</i>	Event flag pointer.	<i>pollhdrp</i>	Poll head pointer.
<i>dev</i>	Device number.										
<i>events</i>	Event flags.										
<i>anyyet</i>	Check current events only.										
<i>reventsp</i>	Event flag pointer.										
<i>pollhdrp</i>	Poll head pointer.										
<b>DESCRIPTION</b>	nochpoll () is a routine that simply returns the value ENXIO. It is intended to be used in the cb_ops(9S) structure of a device driver for devices that do not support the poll(2) system call.										
<b>RETURN VALUES</b>	nochpoll () returns ENXIO.										
<b>CONTEXT</b>	nochpoll () can be called from user or interrupt context.										
<b>SEE ALSO</b>	poll(2), chpoll(9E), cb_ops(9S) <i>Writing Device Drivers</i>										

<b>NAME</b>	nodev – error return function
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt;  int nodev( );</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	nodev() returns ENXIO.
<b>CONTEXT</b>	nodev() can be only called from user context.
<b>SEE ALSO</b>	nulldev(9F), cb_ops(9S) <i>Writing Device Drivers</i>

## noenable(9F)

<b>NAME</b>	noenable – prevent a queue from being scheduled
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>noenable</b>(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the queue.
<b>DESCRIPTION</b>	noenable () prevents the queue <i>q</i> from being scheduled for service by insq(9F), putq(9F) or putbq(9F) when enqueueing an ordinary priority message. The queue can be re-enabled with the enableok(9F) function.
<b>CONTEXT</b>	noenable () can be called from user or interrupt context.
<b>SEE ALSO</b>	enableok(9F), insq(9F), putbq(9F), putq(9F), qenable(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	nulldev – zero return function
<b>SYNOPSIS</b>	<pre>#include &lt;sys/conf.h&gt; #include &lt;sys/ddi.h&gt;  int nulldev();</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	nulldev() returns a 0.
<b>CONTEXT</b>	nulldev() can be called from any context.
<b>SEE ALSO</b>	nodev(9F), cb_ops(9S) <i>Writing Device Drivers</i>

## nvlist\_add\_boolean(9F)

<b>NAME</b>	nvlist_add_boolean, nvlist_add_byte, nvlist_add_int16, nvlist_add_uint16, nvlist_add_int32, nvlist_add_uint32, nvlist_add_int64, nvlist_add_uint64, nvlist_add_string, nvlist_add_nvlist, 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 – value pair functions
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  int nvlist_add_boolean(nvlist_t *nvl, char *name); int nvlist_add_byte(nvlist_t *nvl, char *name, uchar_t val); int nvlist_add_int16(nvlist_t *nvl, char *name, int16_t val); int nvlist_add_uint16(nvlist_t *nvl, char *name, uint16_t val); int nvlist_add_int32(nvlist_t *nvl, char *name, int32_t val); int nvlist_add_uint32(nvlist_t *nvl, char *name, uint32_t val); int nvlist_add_int64(nvlist_t *nvl, char *name, int64_t val); int nvlist_add_uint64(nvlist_t *nvl, char *name, uint64_t val); int nvlist_add_string(nvlist_t *nvl, char *name, char *val); int nvlist_add_nvlist(nvlist_t *nvl, char *name, nvlist_t *val); int nvlist_add_byte_array(nvlist_t *nvl, char *name, uchar_t *val,     uint_t nelem); int nvlist_add_int16_array(nvlist_t *nvl, char *name, int16_t *val,     uint_t nelem); int nvlist_add_uint16_array(nvlist_t *nvl, char *name, uint16_t     *val, uint_t nelem); int nvlist_add_int32_array(nvlist_t *nvl, char *name, int32_t *val,     uint_t nelem); int nvlist_add_uint32_array(nvlist_t *nvl, char *name, uint32_t     *val, uint_t nelem); int nvlist_add_int64_array(nvlist_t *nvl, char *name, int64_t *val,     uint_t nelem); int nvlist_add_uint64_array(nvlist_t *nvl, char *name, uint64_t     *val, uint_t nelem); int nvlist_add_string_array(nvlist_t *nvl, char *name, char **val,     uint_t nelem); int nvlist_add_nvlist_array(nvlist_t *nvl, char *name, nvlist_t     **val, uint_t nelem);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)

<b>PARAMETERS</b>	<p><i>nvlist</i>            The <code>nvlist_t</code> to be processed.</p> <p><i>nvpair</i>            The <code>nvpair_t</code> (name-value pair) to be processed.</p> <p><i>name</i>              Name of the name-value pair (<code>nvpair</code>).</p> <p><i>nelem</i>             Number of elements in value (that is, array size).</p> <p><i>val</i>                Value or starting address of the array value.</p>
<b>DESCRIPTION</b>	<p>These functions add a new name-value pair to an <code>nvlist_t</code>. The memory allocation policy follows that specified in <code>nvlist_alloc()</code>, <code>nvlist_unpack()</code>, or <code>nvlist_dup()</code>. See <code>nvlist_alloc(9F)</code>. The uniqueness of <code>nvpair</code> name and data types follows the <i>nvflag</i> argument specified in <code>nvlist_alloc()</code>.</p> <p>If <code>NV_UNIQUE_NAME</code> was specified for <i>nvflag</i>, existing <code>nvpairs</code> with matching names are removed before the new <code>nvpair</code> is added.</p> <p>If <code>NV_UNIQUE_NAME_TYPE</code> was specified for <i>nvflag</i>, existing <code>nvpairs</code> with matching names and data types are removed before the new <code>nvpair</code> is added.</p> <p>If neither was specified for <i>nvflag</i>, the new <code>nvpair</code> is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.</p>
<b>RETURN VALUES</b>	<p>0                    success</p> <p><code>EINVAL</code>            invalid argument</p> <p><code>ENOMEM</code>           insufficient memory</p>
<b>CONTEXT</b>	<p>These functions can be called from interrupt context only if the <code>nvlist_t</code> was allocated with the <code>KM_NOSLEEP</code> flag set. See <code>nvlist_alloc(9F)</code> for a description of <code>KM_NOSLEEP</code>. These functions can be called from user context in all cases.</p>

## nvlist\_alloc(9F)

<b>NAME</b>	nvlist_alloc, nvlist_free, nvlist_size, nvlist_pack, nvlist_unpack, nvlist_dup – manage a name-value pair list
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  int  nvlist_alloc(nvlist_t **nvp, uint_t nvflag, int kmflag); void nvlist_free(nvlist_t *nvl);  int  nvlist_size(nvlist_t *nvl, size_t *size, int encoding); int  nvlist_pack(nvlist_t *nvl, char **bufp, size_t *buflen, int                 encoding, int kmflag); int  nvlist_unpack(char *buf, size_t buflen, nvlist_t **nvp, int                 kmflag); int  nvlist_dup(nvlist_t *nvl, nvlist_t **nvp, int kmflag);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI)</p> <p><i>nvp</i>                    Address of a pointer to list of name-value pairs (nvlist_t).</p> <p><i>nvflag</i>                Specify bit fields defining nvlist_t properties:</p> <p style="padding-left: 40px;">NV_UNIQUE_NAME                         The nvpair names are unique.</p> <p style="padding-left: 40px;">NV_UNIQUE_NAME_TYPE                         Name-data type combination is unique</p> <p><i>kmflag</i>                Kernel memory allocation policy, either KM_SLEEP or KM_NOSLEEP.</p> <p><i>nvl</i>                    The nvlist_t to be processed.</p> <p><i>size</i>                    Pointer to buffer to contain the encoded size.</p> <p><i>bufp</i>                    Address of buffer to pack nvlist into. Must be 8-byte aligned. If NULL, library will allocate memory.</p> <p><i>buf</i>                    Buffer containing packed nvlist_t.</p> <p><i>buflen</i>                Size of buffer <i>bufp</i> or <i>buf</i> points to.</p> <p><i>encoding</i>              Encoding method for packing.</p>
<b>DESCRIPTION</b>	<p>The nvlist_alloc() function allocates a new name-value pair list and updates <i>nvp</i> to point to the handle. The argument <i>nvflag</i> specifies nvlist_t properties to remain persistent across packing, unpacking, and duplication.</p> <p>The nvlist_free() function frees a name-value pair list.</p> <p>The nvlist_size() function returns the minimum size of a contiguous buffer large enough to pack <i>nvl</i>. The <i>encoding</i> parameter specifies the method of encoding when packing <i>nvl</i>. Supported encoding methods are:</p>

NV_ENCODE_NATIVE	Straight bcopy () as described in bcopy(9F).
NV_ENCODE_XDR	Use XDR encoding, suitable for sending to another host.

The `nvlst_pack()` function packs *nvl* into contiguous memory starting at *\*bufp*. The *encoding* parameter specifies the method of encoding (see above).

- If *\*bufp* is not NULL, *\*bufp* is expected to be a caller-allocated buffer of size *\*buflen*. The *kmflag* argument is ignored.
- If *\*bufp* is NULL, the library will allocate memory and update *\*bufp* to point to the memory and update *\*buflen* to contain the size of the allocated memory. The value of *kmflag* indicates the memory allocation policy

The `nvlst_unpack()` function takes a buffer with a packed `nvlst_t` and unpacks it into a searchable `nvlst_t`. The library allocates memory for `nvlst_t`. The caller is responsible for freeing the memory by calling `nvlst_free()`.

The `nvlst_dup()` function makes a copy of *nvl* and updates *nvp* to point to the copy.

**RETURN VALUES**

For `nvlst_alloc()`, `nvlst_dup()`:

0	success
EINVAL	invalid argument
ENOMEM	insufficient memory

For `nvlst_pack()`, `nvlst_unpack()`:

0	success
EINVAL	invalid argument
ENOMEM	insufficient memory
EFAULT	encode/decode error
ENOTSUP	encode/decode method not supported

For `nvlst_size()`:

0	success
EINVAL	invalid argument

**CONTEXT**

The `nvlst_alloc()`, `nvlst_pack()`, `nvlst_unpack()`, and `nvlst_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.

## nvlist\_lookup\_boolean(9F)

<b>NAME</b>	nvlist_lookup_boolean, nvlist_lookup_byte, nvlist_lookup_int16, nvlist_lookup_int32, nvlist_lookup_int64, nvlist_lookup_uint16, nvlist_lookup_uint32, nvlist_lookup_uint64, nvlist_lookup_uint16_array, nvlist_lookup_uint32_array, nvlist_lookup_uint64_array, nvlist_lookup_nvlist, nvlist_lookup_byte_array, nvlist_lookup_string_array, nvlist_lookup_string, nvlist_lookup_int16_array, nvlist_lookup_int32_array, nvlist_lookup_int64_array, nvlist_lookup_nvlist_array – match name and type indicated by the interface name and retrieve data value
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  int nvlist_lookup_boolean(nvlist_t *nvl, char *name); int nvlist_lookup_byte(nvlist_t *nvl, char *name, uchar_t *val); int nvlist_lookup_int16(nvlist_t *nvl, char *name, int16_t *val); int nvlist_lookup_uint16(nvlist_t *nvl, char *name, uint16_t *val); int nvlist_lookup_int32(nvlist_t *nvl, char *name, int32_t *val); int nvlist_lookup_uint32(nvlist_t *nvl, char *name, uint32_t *val); int nvlist_lookup_int64(nvlist_t *nvl, char *name, int64_t *val); int nvlist_lookup_uint64(nvlist_t *nvl, char *name, uint64_t *val); int nvlist_lookup_string(nvlist_t *nvl, char *name, char **val); int nvlist_lookup_nvlist(nvlist_t *nvl, char *name, nvlist_t     **val); int nvlist_lookup_byte_array(nvlist_t *nvl, char *name, uchar_t     **val, uint_t *nelem); int nvlist_lookup_int16_array(nvlist_t *nvl, char *name, int16_t     **val, uint_t *nelem); int nvlist_lookup_uint16_array(nvlist_t *nvl, char *name, uint16_t     **val, uint_t *nelem); int nvlist_lookup_int32_array(nvlist_t *nvl, char *name, int32_t     **val, uint_t *nelem); int nvlist_lookup_uint32_array(nvlist_t *nvl, char *name, uint32_t     **val, uint_t *nelem); int nvlist_lookup_int64_array(nvlist_t *nvl, char *name, int64_t     **val, uint_t *nelem); int nvlist_lookup_uint64_array(nvlist_t *nvl, char *name, uint64_t     **val, uint_t *nelem); int nvlist_lookup_string_array(nvlist_t *nvl, char *name, char     ***val, uint_t *nelem);</pre>

nvlist\_lookup\_boolean(9F)

```
int nvlist_lookup_nvlist_array(nvlist_t *nvl, char *name, nvlist_t
***val, uint_t *nelem);
```

**INTERFACE  
LEVEL  
PARAMETERS**

Solaris DDI specific (Solaris DDI)

*nvl*                    The list of name-value pairs (nvlist\_t) to be processed.  
*name*                    Name of the name-value pair (nvpair) to search.  
*nelem*                    Address to store the number of elements in value.  
*val*                    Address to store the value or starting address of the array value.

**DESCRIPTION**

These functions find the nvpair that matches the name and type as indicated by the interface name. If one is found, *nelem* and *val* are modified to contain the number of elements in value and the starting address of data, respectively.

These interfaces work for nvlist\_ts allocated with NV\_UNIQUE\_NAME or NV\_UNIQUE\_NAME\_TYPE specified in nvlist\_alloc(). (See nvlist\_alloc(9F).) If this is not the case, the interface will return ENOTSUP because the list potentially contains multiple nvpairs with the same name and type.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until nvlist\_free() is called on *nvl*.

**RETURN VALUES**

0                    success  
EINVAL                invalid argument  
ENOENT                no matching name-value pair found  
ENOTSUP                encode/decode method not supported

**CONTEXT**

These functions can be called from user or interrupt contexts.

## nvlist\_next\_nvpair(9F)

<b>NAME</b>	nvlist_next_nvpair, nvpair_name, nvpair_type – return data regarding name-value pairs
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  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);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>nvl</i>           The list of name-value pairs (nvlist_t) to be processed.</p> <p><i>nvpair</i>       Handle to a name-value pair.</p>
<b>DESCRIPTION</b>	<p>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.</p> <p>The nvpair_name() function returns a string containing the name of nvpair.</p> <p>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.</p>
<b>RETURN VALUES</b>	<p>For nvpair_name(), a string containing the name.</p> <p>For nvpair_type(), an enumerated data type data_type_t. Possible values for data_type_t are as follows:</p> <pre>DATA_TYPE_BOOLEAN DATA_TYPE_BYTE DATA_TYPE_INT16 DATA_TYPE_UINT16 DATA_TYPE_INT32 DATA_TYPE_UINT32 DATA_TYPE_INT64 DATA_TYPE_UINT64 DATA_TYPE_STRING DATA_TYPE_BYTE_ARRAY DATA_TYPE_INT16_ARRAY DATA_TYPE_UINT16_ARRAY DATA_TYPE_INT32_ARRAY DATA_TYPE_UINT32_ARRAY DATA_TYPE_INT64_ARRAY DATA_TYPE_UINT64_ARRAY DATA_TYPE_STRING_ARRAY</pre> <p>For nvlist_next_pair():</p> <p>NULL           Reached end of list.</p> <p>otherwise:     Handle to next nvpair in the list.</p>

`nvlist_next_nvpair(9F)`

**CONTEXT** | The functions described here can be called from user or interrupt context.

## nvlist\_remove(9F)

<b>NAME</b>	nvlist_remove, nvlist_remove_all – remove name-value pairs
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  void <b>nvlist_remove</b>(nvlist_t *nvl, char *name, data_type_t type); void <b>nvlist_remove_all</b>(nvlist_t *nvl, char *name);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	<p><i>nvl</i>        The list of name-value pairs (nvlist_t) to be processed.</p> <p><i>name</i>      Name of the name-value pair (nvpair) to be removed.</p> <p><i>type</i>      Data type of the nvpair to be removed.</p>
<b>DESCRIPTION</b>	<p>The <code>nvlist_remove()</code> function removes the first occurrence of <i>nvpair</i> that matches the name and the type.</p> <p>The <code>nvlist_remove_all()</code> function removes all occurrences of <i>nvpair</i> that match the name, regardless of type.</p>
<b>RETURN VALUES</b>	None
<b>CONTEXT</b>	The <code>nvlist_remove()</code> and <code>nvlist_remove_all()</code> functions can be called from user or interrupt context.

<b>NAME</b>	nvpair_value_byte, nvpair_value_nvlist, nvpair_value_nvlist_array, nvpair_value_int16, nvpair_value_uint16, nvpair_value_int32, nvpair_value_uint32, nvpair_value_int64, nvpair_value_uint64, nvpair_value_string, nvpair_value_byte_array, nvpair_value_int16_array, nvpair_value_uint16_array, nvpair_value_int32_array, nvpair_value_uint32_array, nvpair_value_int64_array, nvpair_value_uint64_array, nvpair_value_string_array – retrieve value from a name-value pair
<b>SYNOPSIS</b>	<pre>#include &lt;sys/nvpair.h&gt;  int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val); int nvpair_value_int16(nvpair_t *nvpair, int16_t *val); int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val); int nvpair_value_int32(nvpair_t *nvpair, int32_t *val); int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val); int nvpair_value_int64(nvpair_t *nvpair, int64_t *val); int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val); int nvpair_value_string(nvpair_t *nvpair, char **val); int nvpair_value_nvlist(nvpair_t *nvpair, nvlist_t **val); int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val, uint_t     *nelem); int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val,     uint_t *nelem); int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val,     uint_t *nelem); int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val,     uint_t *nelem); int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val,     uint_t *nelem); int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val,     uint_t *nelem); int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val,     uint_t *nelem); int nvpair_value_string_array(nvpair_t *nvpair, char ***val, uint_t     *nelem); int nvpair_value_nvlist_array(nvpair_t *nvpair, nvlist_t ***val,     uint_t *nelem);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)

## nvpair\_value\_byte(9F)

<b>PARAMETERS</b>	<i>nvpair</i>	Name-value pair ( <i>nvpair</i> ) to be processed.
	<i>nelem</i>	Address to store the number of elements in value.
	<i>val</i>	Address to store the value or starting address of array value.
<b>DESCRIPTION</b>	These functions retrieve the value of <i>nvpair</i> . The data type of <i>nvpair</i> must match the function name for the call to be successful.	
	There is no <code>nvpair_value_boolean()</code> ; the existence of the name implies the value is true.	
<b>RETURN VALUES</b>	For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until <code>nvlist_free()</code> is called on the <code>nvlist_t</code> from which <i>nvpair</i> is obtained. See <code>nvlist_free(9F)</code>	
	0	Success
	EINVAL	Either one of the arguments is NULL or type of <i>nvpair</i> does not match the interface name.
<b>CONTEXT</b>	These functions can be called from user or interrupt context.	

<b>NAME</b>	OTHERQ, otherq – get pointer to queue’s partner queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  queue_t *OTHERQ(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the queue.
<b>DESCRIPTION</b>	The OTHERQ() function returns a pointer to the other of the two queue structures that make up a STREAMS module or driver. If <i>q</i> points to the read queue the write queue will be returned, and vice versa.
<b>RETURN VALUES</b>	OTHERQ() returns a pointer to a queue’s partner.
<b>CONTEXT</b>	OTHERQ() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Setting Queues</p> <p>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.</p> <pre>1 void 2 set_q_params(q, min, max, hi, lo) 3     queue_t *q; 4     short min; 5     short max; 6     ushort_t hi; 7     ushort_t lo; 8 { 9     q-&gt;q_minpsz = min; 10    q-&gt;q_maxpsz = max; 11    q-&gt;q_hiwat = hi; 12    q-&gt;q_lowat = lo; 13    OTHERQ(q)-&gt;q_minpsz = min; 14    OTHERQ(q)-&gt;q_maxpsz = max; 15    OTHERQ(q)-&gt;q_hiwat = hi; 16    OTHERQ(q)-&gt;q_lowat = lo; 17 }</pre>
<b>SEE ALSO</b>	<p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

outb(9F)

<b>NAME</b>	outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>								
<b>INTERFACE LEVEL</b>	The functions described here are obsolete. For the outb(), outw(), and outl() functions use, respectively, ddi_put8(9F), ddi_put16(9F), and ddi_put32(9F) instead. For repoutsb(), repoutsw(), and repoutsl(), use, respectively, ddi_rep_put8(9F), ddi_rep_put16(9F), and ddi_rep_put32(9F) instead.								
<b>PARAMETERS</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>port</i></td> <td>A valid I/O port address.</td> </tr> <tr> <td><i>value</i></td> <td>The data to be written to the I/O port.</td> </tr> <tr> <td><i>addr</i></td> <td>The address of a buffer from which the values will be fetched.</td> </tr> <tr> <td><i>count</i></td> <td>The number of values to be written to the I/O port.</td> </tr> </table>	<i>port</i>	A valid I/O port address.	<i>value</i>	The data to be written to the I/O port.	<i>addr</i>	The address of a buffer from which the values will be fetched.	<i>count</i>	The number of values to be written to the I/O port.
<i>port</i>	A valid I/O port address.								
<i>value</i>	The data to be written to the I/O port.								
<i>addr</i>	The address of a buffer from which the values will be fetched.								
<i>count</i>	The number of values to be written to the I/O port.								
<b>DESCRIPTION</b>	<p>These routines write data of various sizes to the I/O port with the address specified by <i>port</i>.</p> <p>The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by <i>value</i>.</p> <p>The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. <i>count</i> specifies the number of values to be written. <i>addr</i> is a pointer to a buffer from which the output values are fetched.</p>								
<b>CONTEXT</b>	These functions may be called from user or interrupt context.								
<b>ATTRIBUTES</b>	See attributes(5) for descriptions of the following attributes:								
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">ATTRIBUTE TYPE</th> <th style="text-align: center;">ATTRIBUTE VALUE</th> </tr> </thead> <tbody> <tr> <td>Architecture</td> <td>x86</td> </tr> <tr> <td>Stability Level</td> <td>Obsolete</td> </tr> </tbody> </table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Architecture	x86	Stability Level	Obsolete		
ATTRIBUTE TYPE	ATTRIBUTE VALUE								
Architecture	x86								
Stability Level	Obsolete								
<b>SEE ALSO</b>	eisa(4), isa(4), attributes(5), ddi_put8(9F), ddi_put16(9F), ddi_put32(9F), ddi_rep_put8(9F), ddi_rep_put16(9F), ddi_rep_put32(9F), inb(9F)								

outb(9F)

*Writing Device Drivers*

## pci\_config\_get8(9F)

<b>NAME</b>	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						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  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);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).						
<b>PARAMETERS</b>	<table><tr><td><i>handle</i></td><td>The data access handle returned from pci_config_setup(9F).</td></tr><tr><td><i>offset</i></td><td>Byte offset from the beginning of the PCI Configuration space.</td></tr><tr><td><i>value</i></td><td>Output data.</td></tr></table>	<i>handle</i>	The data access handle returned from pci_config_setup(9F).	<i>offset</i>	Byte offset from the beginning of the PCI Configuration space.	<i>value</i>	Output data.
<i>handle</i>	The data access handle returned from pci_config_setup(9F).						
<i>offset</i>	Byte offset from the beginning of the PCI Configuration space.						
<i>value</i>	Output data.						
<b>DESCRIPTION</b>	<p>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 <i>offset</i> argument must be a multiple of the datum size.</p> <p>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.</p> <p>pci_config_setup(9F) must be called before invoking these functions.</p>						
<b>RETURN VALUES</b>	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_tearardown(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
<code>pci_config_getb</code>	<code>pci_config_get8</code>
<code>pci_config_getw</code>	<code>pci_config_get16</code>
<code>pci_config_getl</code>	<code>pci_config_get32</code>
<code>pci_config_getll</code>	<code>pci_config_get64</code>
<code>pci_config_putb</code>	<code>pci_config_put8</code>
<code>pci_config_putw</code>	<code>pci_config_put16</code>
<code>pci_config_putl</code>	<code>pci_config_put32</code>
<code>pci_config_putll</code>	<code>pci_config_put64</code>

## pci\_config\_setup(9F)

<b>NAME</b>	pci_config_setup, pci_config_tearardown – setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle); void pci_config_tearardown(ddi_acc_handle_t *handle);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>dip</i> Pointer to the device's dev_info structure. <i>handle</i> Pointer to a data access handle.				
<b>DESCRIPTION</b>	pci_config_setup() sets up the necessary resources for enabling subsequent data accesses to the PCI Local Bus Configuration space. pci_config_tearardown() reclaims and removes those resources represented by the data access handle returned from pci_config_setup().				
<b>RETURN VALUES</b>	pci_config_setup() returns: DDI_SUCCESS Successfully setup the resources. DDI_FAILURE Unable to allocate resources for setup.				
<b>CONTEXT</b>	pci_config_setup() must be called from user or kernel context. pci_config_tearardown() can be called from any context.				
<b>NOTES</b>	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.				
<b>ATTRIBUTES</b>	See attributes(5) for descriptions of the following attributes: <table border="1"><thead><tr><th>ATTRIBUTE TYPE</th><th>ATTRIBUTE VALUE</th></tr></thead><tbody><tr><td>Architecture</td><td>PCI Local Bus</td></tr></tbody></table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Architecture	PCI Local Bus
ATTRIBUTE TYPE	ATTRIBUTE VALUE				
Architecture	PCI Local Bus				
<b>SEE ALSO</b>	attributes(5) <i>IEEE 1275 PCI Bus Binding</i>				

<b>NAME</b>	pci_report_pmcap – Report Power Management capability of a PCI device
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pci_report_pmcap(dev_info_t *dip, int cap, void *arg);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI)</p> <p><i>dip</i>                    Pointer to the device's dev_info structure</p> <p><i>cap</i>                    Power management capability</p> <p><i>arg</i>                    Argument for the capability</p>
<b>DESCRIPTION</b>	<p>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:</p> <p>PCI_PM_IDLESPPEED — The PCI_PM_IDLESPPEED 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).</p> <p>The special values of arg are:</p> <p>PCI_PM_IDLESPPEED_ANY    The device can tolerate any idle clock speed.</p> <p>PCI_PM_IDLESPPEED_NONE    The device cannot tolerate slowing down of PCI clock even when idle.</p> <p>If the driver doesn't make this call, PCI_PM_IDLESPPEED_NONE is assumed. In this case, one offending device can keep the entire bus from being power managed.</p>
<b>RETURN VALUES</b>	<p>The pci_report_pmcap(9F) function returns:</p> <p>DDI_SUCCESS            Successful reporting of the capability</p> <p>DDI_FAILURE            Failure to report capability because of invalid argument(s)</p>
<b>CONTEXT</b>	The pci_report_pmcap(9F) function can be called from user, kernel and interrupt context.
<b>EXAMPLES</b>	<p>1. A device driver knows that the device it controls works with any clock between DC and 33 MHz as specified in <i>Section 4.2.3.1: Clock Specification of the PCI Bus Specification Revision 2.1</i>. The device driver makes the following call from its attach(9E):</p> <pre>if (pci_report_pmcap(dip, PCI_PM_IDLESPPEED, PCI_PM_IDLESPPEED_ANY) !=     DDI_SUCCESS)     cmn_err(CE_WARN, "%s%d: pci_report_pmcap failed\n",</pre>

## pci\_report\_pmcap(9F)

```
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_IDLESPPEED, (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*

<b>NAME</b>	pci_save_config_regs, pci_restore_config_regs – save and restore the PCI configuration registers
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pci_save_config_regs(dev_info_t *dip); int pci_restore_config_regs(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL ARGUMENTS</b>	<p>Solaris DDI-specific (Solaris DDI).</p> <p><i>dip</i>  Pointer to the device's dev_info structure.</p>
<b>DESCRIPTION</b>	<p>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().</p> <p>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).</p>
<b>RETURN VALUES</b>	<p>pci_save_config_regs() and pci_restore_config_regs() return:</p> <p>DDI_SUCCESS  Operation completed successfully.</p> <p>DDI_FAILURE  Operation failed to complete successfully.</p>
<b>CONTEXT</b>	Both these functions can be called from user or kernel context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Invoking the save and restore functions</p> <pre>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(&amp;xx-&gt;x_mutex);      switch (level) {     case PM_LEVEL_D0:         XX_POWER_ON(xx);         if (pci_restore_config_regs(dip) == DDI_FAILURE) {             /*              * appropriate error path handling here              */             ...         }     } }</pre>

## pci\_save\_config\_regs(9F)

**EXAMPLE 1** Invoking the save and restore functions (Continued)

```
        rval = DDI_FAILURE;
    }
    break;

case PM_LEVEL_D3:
    if (pci_save_config_regs(dip) == DDI_FAILURE) {
        /*
         * appropriate error path handling here
         */
        ...
        rval = DDI_FAILURE;
    }
    else {
        XX_POWER_OFF(xx);
    }
    break;

default:
    rval = DDI_FAILURE;
    break;
}

mutex_exit (&xx->x_mutex);
return (rval);
}
```

**ATTRIBUTES** See attributes(5) for descriptions of the following attributes:

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*

<b>NAME</b>	physio, minphys – perform physical I/O												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/buf.h&gt; #include &lt;sys/uio.h&gt;  int <b>physio</b>(int (*strat) (struct buf *), struct buf *bp, dev_t dev, int             rw, void (*mincnt) (struct buf *), struct uio *uio);  void <b>minphys</b>(struct buf *bp);</pre>												
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).												
<b>PARAMETERS</b>													
<b>physio()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>strat</i></td> <td>Pointer to device strategy routine.</td> </tr> <tr> <td><i>bp</i></td> <td>Pointer to a buf(9S) structure describing the transfer. If <i>bp</i> is set to NULL then <b>physio()</b> allocates one which is automatically released upon completion.</td> </tr> <tr> <td><i>dev</i></td> <td>The device number.</td> </tr> <tr> <td><i>rw</i></td> <td>Read/write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.</td> </tr> <tr> <td><i>mincnt</i></td> <td>Routine which bounds the maximum transfer unit size.</td> </tr> <tr> <td><i>uio</i></td> <td>Pointer to the uio structure which describes the user I/O request.</td> </tr> </table>	<i>strat</i>	Pointer to device strategy routine.	<i>bp</i>	Pointer to a buf(9S) structure describing the transfer. If <i>bp</i> is set to NULL then <b>physio()</b> allocates one which is automatically released upon completion.	<i>dev</i>	The device number.	<i>rw</i>	Read/write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.	<i>mincnt</i>	Routine which bounds the maximum transfer unit size.	<i>uio</i>	Pointer to the uio structure which describes the user I/O request.
<i>strat</i>	Pointer to device strategy routine.												
<i>bp</i>	Pointer to a buf(9S) structure describing the transfer. If <i>bp</i> is set to NULL then <b>physio()</b> allocates one which is automatically released upon completion.												
<i>dev</i>	The device number.												
<i>rw</i>	Read/write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.												
<i>mincnt</i>	Routine which bounds the maximum transfer unit size.												
<i>uio</i>	Pointer to the uio structure which describes the user I/O request.												
<b>minphys()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>bp</i></td> <td>Pointer to a buf structure.</td> </tr> </table>	<i>bp</i>	Pointer to a buf structure.										
<i>bp</i>	Pointer to a buf structure.												
<b>DESCRIPTION</b>	<p><b>physio()</b> performs unbuffered I/O operations between the device <i>dev</i> and the address space described in the <i>uio</i> structure.</p> <p>Prior to the start of the transfer <b>physio()</b> verifies the requested operation is valid by checking the protection of the address space specified in the <i>uio</i> structure. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, <i>strat()</i>, is then called one or more times to perform the physical I/O operations. <b>physio()</b> uses <b>biowait(9F)</b> to block until <i>strat()</i> has completed each transfer. Upon completion, or detection of an error, <b>physio()</b> unlocks the pages and returns the error status.</p> <p><b>physio()</b> uses <b>mincnt()</b> to bound the maximum transfer unit size to the system, or device, maximum length. <b>minphys()</b> is the system <b>mincnt()</b> routine for use with <b>physio()</b> operations. Drivers which do not provide their own local <b>mincnt()</b> routines should call <b>physio()</b> with <b>minphys()</b>.</p> <p><b>minphys()</b> limits the value of <i>bp-&gt;b_bcount</i> to a sensible default for the capabilities of the system. Drivers that provide their own <b>mincnt()</b> routine should also call <b>minphys()</b> to make sure they do not exceed the system limit.</p>												
<b>RETURN VALUES</b>	<b>physio()</b> returns:												

## physio(9F)

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.

<b>NAME</b>	pm_busy_component, pm_idle_component – Control device component availability for Power Management				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pm_busy_component(dev_info_t *dip, int component); int pm_idle_component(dev_info_t *dip, int component);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)				
<b>PARAMETERS</b>					
<b>pm_busy_component</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>dip</i></td> <td>Pointer to the device's dev_info structure.</td> </tr> <tr> <td><i>component</i></td> <td>The number of the component to be power-managed.</td> </tr> </table>	<i>dip</i>	Pointer to the device's dev_info structure.	<i>component</i>	The number of the component to be power-managed.
<i>dip</i>	Pointer to the device's dev_info structure.				
<i>component</i>	The number of the component to be power-managed.				
<b>pm_idle_component</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>dip</i></td> <td>Pointer to the device's dev_info structure.</td> </tr> <tr> <td><i>component</i></td> <td>The number of the component to be power-managed.</td> </tr> </table>	<i>dip</i>	Pointer to the device's dev_info structure.	<i>component</i>	The number of the component to be power-managed.
<i>dip</i>	Pointer to the device's dev_info structure.				
<i>component</i>	The number of the component to be power-managed.				
<b>DESCRIPTION</b>	<p>The pm_busy_component () function sets <i>component</i> of <i>dip</i> 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.</p> <p>The pm_idle_component () function marks <i>component</i> idle, recording the time that <i>component</i> 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.</p>				
<b>RETURN VALUES</b>	<p>The pm_busy_component () and pm_idle_component () functions return:</p> <table border="0"> <tr> <td style="padding-right: 20px;">DDI_SUCCESS</td> <td>Successfully set the indicated component busy or idle.</td> </tr> <tr> <td>DDI_FAILURE</td> <td>Invalid component number <i>component</i> or the device has no components.</td> </tr> </table>	DDI_SUCCESS	Successfully set the indicated component busy or idle.	DDI_FAILURE	Invalid component number <i>component</i> or the device has no components.
DDI_SUCCESS	Successfully set the indicated component busy or idle.				
DDI_FAILURE	Invalid component number <i>component</i> or the device has no components.				
<b>CONTEXT</b>	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.				
<b>ATTRIBUTES</b>	See attributes(5) for descriptions of the following attributes:				

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

pm\_busy\_component(9F)

**SEE ALSO** | power.conf(4), pm(7D), pm\_create\_components(9F),  
pm\_destroy\_components(9F), pm\_raise\_power(9F), pm(9P), pm-components(9P)

*Writing Device Drivers*

<b>NAME</b>	pm_create_components, pm_destroy_components – Create or destroy power-manageable components				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pm_create_components(dev_info_t *dip, int components); void pm_destroy_components(dev_info_t *dip);</pre>				
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i>                    Pointer to the device's dev_info structure</p> <p><i>components</i>            Number of components to create</p>				
<b>DESCRIPTION</b>	<p>The pm_create_components() and pm_destroy_components() functions are now obsolete and will be removed in a future release. It is recommended that the driver use pm-components(9) instead.</p> <p>The pm_create_components() function creates power-manageable components for a device. It should be called from the driver's attach(9E) entry point if the device has power-manageable components.</p> <p>The correspondence of components to parts of the physical device controlled by the driver are the responsibility of the driver.</p> <p>The pm_destroy_components() function removes all components from the device. It should be called from the driver's detach(9E) entry point.</p>				
<b>RETURN VALUES</b>	<p>The pm_create_components() function returns:</p> <p>DDI_SUCCESS      Components are successfully created.</p> <p>DDI_FAILURE      The device already has components.</p>				
<b>CONTEXT</b>	These functions may be called from user or kernel context.				
<b>ATTRIBUTES</b>	See attributes(5) for descriptions of the following attributes:				
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">ATTRIBUTE TYPE</th> <th style="text-align: center;">ATTRIBUTE VALUE</th> </tr> </thead> <tbody> <tr> <td>Interface stability</td> <td>Obsolete</td> </tr> </tbody> </table>	ATTRIBUTE TYPE	ATTRIBUTE VALUE	Interface stability	Obsolete
ATTRIBUTE TYPE	ATTRIBUTE VALUE				
Interface stability	Obsolete				
<b>SEE ALSO</b>	<p>power.conf(4), pm(7D), attach(9E), detach(9E), pm_busy_component(9F), pm_idle_component(9F), pm(9P), pm-components(9P)</p> <p><i>Writing Device Drivers</i></p>				

## pm\_get\_normal\_power(9F)

<b>NAME</b>	pm_get_normal_power, pm_set_normal_power – get or set a device component's normal power level
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pm_get_normal_power(dev_info_t *dip, int component); void pm_set_normal_power(dev_info_t *dip, int component, int level);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)
<b>PARAMETERS</b>	pm_get_normal_power() parameters: <i>dip</i> Pointer to the device's <i>dev_info</i> structure <i>component</i> Number of component from which to get normal power level  pm_set_normal_power() parameters: <i>dip</i> Pointer to the device's <i>dev_info</i> structure <i>component</i> Number of component for which to set normal power level <i>level</i> Component's new normal power level
<b>DESCRIPTION</b>	<p>The <code>pm_get_normal_power()</code> and <code>pm_set_normal_power()</code> functions are now obsolete and will be removed in a future release. It is recommended that device drivers use new automatic device Power Management interfaces.</p> <p>The <code>pm_get_normal_power()</code> function returns the normal power level of <i>component</i> of the device <i>dip</i>.</p> <p>The <code>pm_set_normal_power()</code> function sets the normal power level of <i>component</i> of the device <i>dip</i> to <i>level</i>.</p> <p>When a device has been power managed and is being returned to a state to be used by the system, it will be brought to its normal power level. Except for a power level of 0, which is defined by the system to mean "powered off," the interpretation of the meaning of the power level is entirely up to the driver.</p>
<b>RETURN VALUES</b>	The <code>pm_get_normal_power()</code> function returns: <i>level</i> The normal power level of the specified component (a positive integer).  DDI_FAILURE Invalid component number <i>component</i> or the device has no components.
<b>CONTEXT</b>	These functions can be called from user or kernel context.
<b>ATTRIBUTES</b>	See <code>attributes(5)</code> for descriptions of the following attributes:

pm\_get\_normal\_power(9F)

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Obsolete

**SEE ALSO** `power.conf(4)`, `pm(7D)`, `pm(9P)`, `power(9E)`, `pm_busy_component(9F)`,  
`pm_create_components(9F)`, `pm_destroy_components(9F)`,  
`pm_idle_component(9F)`

*Writing Device Drivers*

## pm\_power\_has\_changed(9F)

<b>NAME</b>	pm_power_has_changed – Notify Power Management framework of autonomous power level change
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pm_power_has_changed(dev_info_t *dip, int component, int level);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI)  <i>dip</i> Pointer to the device dev_info structure <i>component</i> Number of the component that has changed power level <i>level</i> Power level to which the indicated component has changed
<b>DESCRIPTION</b>	<p>The pm_power_has_changed(9) function notifies the Power Management framework that the power level of component of <i>dip</i> has changed to <i>level</i>.</p> <p>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().</p> <p>Because of the asynchronous nature of these events, the Power Management framework might have called power(9E) between the device's autonomous power level change and the driver calling pm_power_has_changed(), or the framework may be in the process of changing the power level when pm_power_has_changed() is called. To handle these situations correctly, the driver should verify that the device is indeed at the level or set the device to the level if it doesn't support inquiring of power levels, before calling pm_power_has_changed(). In addition, the driver should prevent a power(9E) entry point from running in parallel with pm_power_has_changed().</p>
<b>RETURN VALUES</b>	The pm_power_has_changed() function returns:  DDI_SUCCESS        The power level of component was successfully updated to <i>level</i> . DDI_FAILURE        Invalid component <i>component</i> or power level <i>level</i>
<b>CONTEXT</b>	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.
<b>EXAMPLES</b>	A hypothetical driver might include this code to handle pm_power_has_changed(9):  <pre>static int xxusb_intr(struct buf *bp) {     ... }</pre>

## pm\_power\_has\_changed(9F)

```
/*
 * 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)`, `power(9E)`, `pm_busy_components(9F)`,  
`pm_idle_components(9F)`, `pm_raise_power(9F)`, `pm_lower_power(9F)`, `pm(9P)`,  
`pm-components(9P)`

*Writing Device Drivers*

## pm\_raise\_power(9F)

<b>NAME</b>	pm_raise_power, pm_lower_power – Raise or lower power of components						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int pm_raise_power(dev_info_t *dip, int component, int level); int pm_lower_power(dev_info_t *dip, int component, int level);</pre>						
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI)						
<b>PARAMETERS</b>							
<b>pm_raise_power</b>	<table><tr><td><i>dip</i></td><td>Pointer to the device's dev_info structure</td></tr><tr><td><i>component</i></td><td>The number of the <i>component</i> for which a power level change is desired</td></tr><tr><td><i>level</i></td><td>The power level to which the indicated <i>component</i> will be raised</td></tr></table>	<i>dip</i>	Pointer to the device's dev_info structure	<i>component</i>	The number of the <i>component</i> for which a power level change is desired	<i>level</i>	The power level to which the indicated <i>component</i> will be raised
<i>dip</i>	Pointer to the device's dev_info structure						
<i>component</i>	The number of the <i>component</i> for which a power level change is desired						
<i>level</i>	The power level to which the indicated <i>component</i> will be raised						
<b>pm_lower_power</b>	<table><tr><td><i>dip</i></td><td>Pointer to the device's dev_info structure</td></tr><tr><td><i>component</i></td><td>The number of the <i>component</i> for which a power level change is desired</td></tr><tr><td><i>level</i></td><td>The power level to which the indicated <i>component</i> will be lowered</td></tr></table>	<i>dip</i>	Pointer to the device's dev_info structure	<i>component</i>	The number of the <i>component</i> for which a power level change is desired	<i>level</i>	The power level to which the indicated <i>component</i> will be lowered
<i>dip</i>	Pointer to the device's dev_info structure						
<i>component</i>	The number of the <i>component</i> for which a power level change is desired						
<i>level</i>	The power level to which the indicated <i>component</i> will be lowered						
<b>DESCRIPTION</b>	<p>The pm_raise_power(9F) function requests the Power Management framework to raise the power level of <i>component</i> of <i>dip</i> to at least <i>level</i>.</p> <p>The state of the device should be examined before each physical access. The pm_raise_power(9F) function should be called to set a <i>component</i> to the required power level if the operation to be performed requires the <i>component</i> to be at a power level higher than its current power level.</p> <p>When pm_raise_power(9F) returns with success, the <i>component</i> 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).</p> <p>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).</p> <p>The pm_lower_power(9F) function requests the Power Management framework to lower the power level of <i>component</i> of <i>dip</i> to at most <i>level</i>.</p> <p>Normally, transitions to lower power levels are initiated by the Power Management framework based on <i>component</i> 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.</p>						

pm\_raise\_power(9F)

If automatic Power Management is disabled ( see `dtpower(1M)` and `power.conf(4)` ), `pm_lower_power(9F)` returns `DDI_SUCCESS` without changing the power level of the component. Otherwise, when `pm_lower_power(9F)` returns with success, the *component* is guaranteed to be at most at the requested power level. Since the actual device power level may be lower than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from `pm_lower_power(9F)`.

The `pm_lower_power(9F)` may cause re-entry of the driver `power(9E)` to lower the power level. Deadlock may result if the driver locks are held across the call to `pm_raise_power(9F)`.

## RETURN VALUES

The `pm_raise_power(9F)` function returns:

`DDI_SUCCESS`      *Component* is now at the requested power level or higher.

`DDI_FAILURE`      *Component* or *level* is out of range, or the framework was unable to raise the power level of the component to the requested level.

The `pm_lower_power(9F)` function returns:

`DDI_SUCCESS`      *Component* is now at the requested power level or lower, or automatic Power Management is disabled.

`DDI_FAILURE`      *Component* or *level* is out of range, or the framework was unable to lower the power level of the component to the requested level, or the device is not detaching.

## EXAMPLES

A hypothetical disk driver might include this code to handle `pm_raise_power(9F)`:

```
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);
        }
    }
}
```

## pm\_raise\_power(9F)

```

        }
        xsp->pm_busycnt++;
    }
    mutex_exit(xsp->lock);
    if (pm_raise_power(xsp->dip,
        XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
        bioerror(bp, EIO);
        biodone(bp);
        return (0);
    }
    mutex_enter(xsp->lock);

    ....
}

xxdisk_power(dev_info *dip, int comp, int level)
{
    ...

    /*
     * We fail the power() entry point if the device is busy and
     * request is to lower the power level.
     */
    ASSERT(mutex_owned(xsp->lock));
    if (xsp->pm_busycnt >= 1) {
        (level < xsp->cur_level) {
            mutex_exit(xsp->lock);
            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)`, `power(9E)`, `pm_busy_component(9F)`, `pm_idle_component(9F)`, `pm(9P)`, `pm-components(9P)`

*Writing Device Drivers*



pm\_trans\_check(9F)

0 Power cycle is not advised  
-1 Error due to invalid argument.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

**SEE ALSO** `power.conf(4)`, `attributes(5)`, `power(9E)`

*Writing Device Drivers*

*Using Power Management*

<b>NAME</b>	pollwakeuper – inform a process that an event has occurred
<b>SYNOPSIS</b>	<pre>#include &lt;sys/poll.h&gt;  void <b>pollwakeuper</b>(struct pollhead *php, short event);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>php</i>        Pointer to a pollhead structure.</p> <p><i>event</i>      Event to notify the process about.</p>
<b>DESCRIPTION</b>	pollwakeuper() 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.
<b>CONTEXT</b>	pollwakeuper() can be called from user or interrupt context.
<b>SEE ALSO</b>	poll(2), chpoll(9E) <i>Writing Device Drivers</i>
<b>NOTES</b>	Driver defined locks should not be held across calls to this function.

## proc\_signal(9F)

<b>NAME</b>	proc_signal, proc_ref, proc_unref – send a signal to a process														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt; #include &lt;sys/signal.h&gt;  void *proc_ref(void);  void proc_unref(void *pref);  int proc_signal(void *pref, int sig);</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>pref</i> A handle for the process to be signalled. <i>sig</i> Signal number to be sent to the process.														
<b>DESCRIPTION</b>	<p>This set of routines allows a driver to send a signal to a process. The routine <code>proc_ref()</code> 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, <code>proc_unref()</code> should be used to remove it as soon as it is no longer needed. <code>proc_signal()</code> is used to send signal <i>sig</i> to the referenced process. The following set of signals may be sent to a process from a driver:</p> <table><tr><td>SIGHUP</td><td>The device has been disconnected.</td></tr><tr><td>SIGINT</td><td>The interrupt character has been received.</td></tr><tr><td>SIGQUIT</td><td>The quit character has been received.</td></tr><tr><td>SIGPOLL</td><td>A pollable event has occurred.</td></tr><tr><td>SIGKILL</td><td>Kill the process (cannot be caught or ignored).</td></tr><tr><td>SIGWINCH</td><td>Window size change.</td></tr><tr><td>SIGURG</td><td>Urgent data are available.</td></tr></table> <p>See <code>signal(3HEAD)</code> for more details on the meaning of these signals.</p> <p>If the process has exited at the time the signal was sent, <code>proc_signal()</code> returns an error code; the caller should remove the reference on the process by calling <code>proc_unref()</code>.</p> <p>The driver writer must ensure that for each call made to <code>proc_ref()</code>, there is exactly one corresponding call to <code>proc_unref()</code>.</p>	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.
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.														
<b>RETURN VALUES</b>	<p><code>proc_ref()</code> returns the following:</p> <table><tr><td><i>pref</i></td><td>An opaque handle used to refer to the current process.</td></tr></table> <p><code>proc_signal()</code> returns the following:</p> <table><tr><td>0</td><td>The process existed before the signal was sent.</td></tr></table>	<i>pref</i>	An opaque handle used to refer to the current process.	0	The process existed before the signal was sent.										
<i>pref</i>	An opaque handle used to refer to the current process.														
0	The process existed before the signal was sent.														

-1           The process no longer exists; no signal was sent.

**CONTEXT**    proc\_unref () and proc\_signal () can be called from user or interrupt context.  
proc\_ref () should only be called from user context.

**SEE ALSO**    signal(3HEAD), putnextctl1(9F)

*Writing Device Drivers*

## ptob(9F)

<b>NAME</b>	ptob – convert size in pages to size in bytes
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  unsigned long <b>ptob</b>(unsigned long <i>numpages</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>numpages</i> Size in number of pages to convert to size in bytes.
<b>DESCRIPTION</b>	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 <code>ptob(2)</code> returns 4096. <code>ptob(0)</code> returns 0.
<b>RETURN VALUES</b>	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.
<b>CONTEXT</b>	<code>ptob()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>btop(9F)</code> , <code>btopr(9F)</code> , <code>ddi_ptob(9F)</code> <i>Writing Device Drivers</i>

<b>NAME</b>	pullupmsg – concatenate bytes in a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>pullupmsg</b>(mblk_t *mp, ssize_t len);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>mp</i>        Pointer to the message whose blocks are to be concatenated. <i>mblk_t</i> is an instance of the <i>msgb(9S)</i> structure.</p> <p><i>len</i>        Number of bytes to concatenate.</p>
<b>DESCRIPTION</b>	<p><i>pullupmsg</i> () tries to combine multiple data blocks into a single block. <i>pullupmsg</i> () concatenates and aligns the first <i>len</i> data bytes of the message pointed to by <i>mp</i>. If <i>len</i> equals -1, all data are concatenated. If <i>len</i> bytes of the same message type cannot be found, <i>pullupmsg</i> () fails and returns 0.</p>
<b>RETURN VALUES</b>	On success, 1 is returned; on failure, 0 is returned.
<b>CONTEXT</b>	<i>pullupmsg</i> () can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <i>pullupmsg</i> ()</p> <p>This is a driver write <i>srv(9E)</i> (service) routine for a device that does not support scatter/gather DMA. For all <i>M_DATA</i> messages, the data will be transferred to the device with DMA. First, try to pull up the message into one message block with the <i>pullupmsg</i> () 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.</p> <pre> 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-&gt;b_datap-&gt;db_type) { 11            case M_DATA: 12                if (pullupmsg(mp, -1)) { 13                    dma_addr = vtop(mp-&gt;b_rptr); 14                    dma_len = mp-&gt;b_wptr - mp-&gt;b_rptr; 15                    xxx_do_dma(dma_addr, dma_len); 16                    freemsg(mp); 17                    break; 18                } 19                for (tmp = mp; tmp; tmp = tmp-&gt;b_cont) { 20                    dma_addr = vtop(tmp-&gt;b_rptr); 21                    dma_len = tmp-&gt;b_wptr - tmp-&gt;b_rptr; 22                    xxx_do_dma(dma_addr, dma_len);</pre>

## pullupmsg(9F)

**EXAMPLE 1** Using pullupmsg() (Continued)

```
23             }
24             freemsg(mp);
25             break;
26         . . .
27     }
28 }
```

**SEE ALSO** [srv\(9E\)](#), [allocb\(9F\)](#), [msgpullup\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES** pullupmsg() is not included in the DKI and will be removed from the system in a future release. Device driver writers are strongly encouraged to use msgpullup(9F) instead of pullupmsg().

<b>NAME</b>	put – call a STREAMS put procedure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void put(queue_t *q, mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to a STREAMS queue.</p> <p><i>mp</i>           Pointer to message block being passed into queue.</p>
<b>DESCRIPTION</b>	put () calls the put procedure ( put(9E) entry point) for the STREAMS queue specified by <i>q</i> , passing it the message block referred to by <i>mp</i> . It is typically used by a driver or module to call its own put procedure.
<b>CONTEXT</b>	<p>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 <i>q</i> argument.</p> <p>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.</p>
<b>SEE ALSO</b>	<p>put(9E), freezestr(9F), putctl(9F), putctl1(9F), putnext(9F), putnextctl(9F), putnextctl1(9F), qprocson(9F), qreply(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>
<b>NOTES</b>	<p>The caller cannot have the stream frozen when calling this function. See freezestr(9F).</p> <p>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.</p> <p>The put () and putnext () functions should be called only after qprocson () is finished.</p>

## putbq(9F)

<b>NAME</b>	putbq – place a message at the head of a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>putbq</b>(queue_t *q, mblk_t *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to the queue.</p> <p><i>bp</i>            Pointer to the message block.</p>
<b>DESCRIPTION</b>	<p>putbq() places a message at the beginning of the appropriate section of the message queue. There are always sections for high priority and ordinary messages. If other priority bands are used, each will have its own section of the queue, in priority band order, after high priority messages and before ordinary messages. putbq() can be used for ordinary, priority band, and high priority messages. However, unless precautions are taken, using putbq() with a high priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.</p> <p>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.</p>
<b>RETURN VALUES</b>	<p>putbq() returns 1 upon success and 0 upon failure.</p> <p><b>Note</b> – Upon failure, the caller should call freemsg(9F) to free the pointer to the message block.</p>
<b>CONTEXT</b>	putbq() can be called from user or interrupt context.
<b>EXAMPLES</b>	See the bufcall(9F) function page for an example of putbq().
<b>SEE ALSO</b>	bcanput(9F), bufcall(9F), canput(9F), getq(9F), putq(9F)
	<i>Writing Device Drivers</i>
	<i>STREAMS Programming Guide</i>

<b>NAME</b>	putctl1 – send a control message with a one-byte parameter to a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int putctl1(queue_t *q, int type, int p);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i> Queue to which the message is to be sent.</p> <p><i>type</i> Type of message.</p> <p><i>p</i> One-byte parameter.</p>
<b>DESCRIPTION</b>	putctl1(), like putctl(9F), tests the <i>type</i> argument to make sure a data type has not been specified, and attempts to allocate a message block. The <i>p</i> 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 <i>type</i> 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 <i>q</i> with the newly allocated and initialized message.
<b>RETURN VALUES</b>	On success, 1 is returned. 0 is returned if <i>type</i> is a data type, or if a message block cannot be allocated.
<b>CONTEXT</b>	putctl1() can be called from user or interrupt context.
<b>EXAMPLES</b>	See the putctl(9F) function page for an example of putctl1().
<b>SEE ALSO</b>	<p>put(9E), allocb(9F), datamsg(9F), putctl(9F), putnextctl1(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## putctl(9F)

<b>NAME</b>	putctl – send a control message to a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int putctl(queue_t *q, int type);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>q</i> Queue to which the message is to be sent. <i>type</i> Message type (must be control, not data type).
<b>DESCRIPTION</b>	putctl() tests the <i>type</i> argument to make sure a data type has not been specified, and then attempts to allocate a message block. putctl() fails if <i>type</i> 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 <i>q</i> with the newly allocated and initialized messages.
<b>RETURN VALUES</b>	On success, 1 is returned. If <i>type</i> is a data type, or if a message block cannot be allocated, 0 is returned.
<b>CONTEXT</b>	putctl() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using putctl()</p> <p>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 <i>parm</i> 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).</p> <pre>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-&gt;q_next, M_BREAK)) 12                num_alloc_fail++; 13            break; 14 15        case M_DELAY: 16            if (!putctl1(wrq-&gt;q_next, M_DELAY, parm)) 17                num_alloc_fail++; 18            break; 19 20        default: 21            cmn_err(CE_PANIC, "send_ctl: bad message type passed"); 22            break;</pre>

**EXAMPLE 1** Using putctl() (Continued)

```
23         }  
24     }
```

**SEE ALSO** put(9E), cmn\_err(9F), datamsg(9F), putctl1(9F), putnextctl(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

## putnext(9F)

<b>NAME</b>	putnext – send a message to the next queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>putnext</b> (queue_t *q, mblk_t *mp) ;</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to the queue from which the message <i>mp</i> will be sent.</p> <p><i>mp</i>           Message to be passed.</p>
<b>DESCRIPTION</b>	putnext () is used to pass a message to the put(9E) routine of the next queue in the stream.
<b>RETURN VALUES</b>	None.
<b>CONTEXT</b>	putnext () can be called from user or interrupt context.
<b>EXAMPLES</b>	See allocb(9F) for an example of using putnext () .
<b>SEE ALSO</b>	put(9E), allocb(9F), put(9F), qprocson(9F)  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>
<b>NOTES</b>	The put () and putnext () functions should be called only after qprocson () is finished.

<b>NAME</b>	putnextctl1 – send a control message with a one-byte parameter to a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>putnextctl1</b>(queue_t *q, int type, int p);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>q</i> Queue to which the message is to be sent.</p> <p><i>type</i> Type of message.</p> <p><i>p</i> One-byte parameter.</p>
<b>DESCRIPTION</b>	<p><code>putnextctl1()</code>, like <code>putctl1(9F)</code>, tests the <i>type</i> argument to make sure a data type has not been specified, and attempts to allocate a message block. The <i>p</i> parameter can be used, for example, to specify how long the delay will be when an <code>M_DELAY</code> message is being sent. <code>putnextctl1()</code> fails if <i>type</i> is <code>M_DATA</code>, <code>M_PROTO</code>, or <code>M_PCPROTO</code>, or if a message block cannot be allocated. If successful, <code>putnextctl1()</code> calls the <code>put(9E)</code> routine of the queue pointed to by <i>q</i> with the newly allocated and initialized message.</p> <p>A call to <code>putnextctl1(q,type,p)</code> is an atomic equivalent of <code>putctl1(q-&gt;q_next, type,p)</code>. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing <i>q</i> through its <code>q_next</code> field and then invoking <code>putctl1(9F)</code> proceeds without interference from other threads.</p> <p><code>putnextctl1()</code> should always be used in preference to <code>putctl1(9F)</code></p>
<b>RETURN VALUES</b>	On success, 1 is returned. 0 is returned if <i>type</i> is a data type, or if a message block cannot be allocated.
<b>CONTEXT</b>	<code>putnextctl1()</code> can be called from user or interrupt context.
<b>EXAMPLES</b>	See the <code>putnextctl1(9F)</code> function page for an example of <code>putnextctl1()</code> .
<b>SEE ALSO</b>	<p><code>put(9E)</code>, <code>alloca(9F)</code>, <code>datamsg(9F)</code>, <code>putctl1(9F)</code>, <code>putnextctl1(9F)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## putnextctl(9F)

<b>NAME</b>	putnextctl – send a control message to a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int putnextctl(queue_t *q, int type);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i> Queue to which the message is to be sent.</p> <p><i>type</i> Message type (must be control, not data type).</p>
<b>DESCRIPTION</b>	<p>putnextctl() tests the <i>type</i> argument to make sure a data type has not been specified, and then attempts to allocate a message block. putnextctl() fails if <i>type</i> 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 <i>q</i> with the newly allocated and initialized messages.</p> <p>A call to putnextctl(<i>q</i>, <i>type</i>) is an atomic equivalent of putctl(<i>q</i>-&gt;q_next, <i>type</i>). The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing <i>q</i> through its q_next field and then invoking putctl(9F) proceeds without interference from other threads.</p> <p>putnextctl() should always be used in preference to putctl(9F)</p>
<b>RETURN VALUES</b>	On success, 1 is returned. If <i>type</i> is a data type, or if a message block cannot be allocated, 0 is returned.
<b>CONTEXT</b>	putnextctl() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p>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 <i>parm</i> 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).</p> <pre>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 12    case M_DELAY: 13        if (!putnextctl1(wrq, M_DELAY, parm)) 14            num_alloc_fail++; 15        break; 16 17    default:</pre>

putnextctl(9F)

```
18             cmn_err(CE_PANIC, "send_ctl: bad message type passed");
19             break;
20         }
21     }
```

**SEE ALSO** put(9E), cmn\_err(9F), datamsg(9F), putctl(9F), putnextctl1(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

## putq(9F)

<b>NAME</b>	putq – put a message on a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int putq(queue_t *q, mblk_t *bp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to the queue to which the message is to be added.</p> <p><i>bp</i>           Message to be put on the queue.</p>
<b>DESCRIPTION</b>	putq() is used to put messages on a driver's queue after the module's put routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. If QNOENB is not set, the service routine is enabled. If no other processing is done, putq() can be used as the module's put routine.
<b>RETURN VALUES</b>	putq() returns 1 on success and 0 on failure.
	<b>Note</b> – Upon failure, the caller should call freemsg(9F) to free the pointer to the message block.
<b>CONTEXT</b>	putq() can be called from user or interrupt context.
<b>EXAMPLES</b>	See the datamsg(9F) function page for an example of putq().
<b>SEE ALSO</b>	datamsg(9F), putbq(9F), qenable(9F), rmvq(9F)
	<i>Writing Device Drivers</i>
	<i>STREAMS Programming Guide</i>

<b>NAME</b>	qassociate – associate STREAMS queue with driver instance
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/stream.h&gt; #include &lt;sys/stropts.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  int <b>qassociate</b>(queue_t *q, int instance);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI). This entry point is required for drivers which export <code>cb_ops(9S)</code> entry points.
<b>PARAMETERS</b>	<p><code>queue_t *q</code> Pointer to a <code>queue(9S)</code> structure. Either the read or write queue can be used.</p> <p><code>int instance</code> Driver instance number or -1.</p>
<b>DESCRIPTION</b>	<p>The <code>qassociate()</code> 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.</p> <p>A DLPI style-2 driver calls <code>qassociate()</code> while processing the <code>DL_ATTACH_REQ</code> message. The driver is also expected to call this interface while performing stream associations through other means, such as <code>ndd(1M)</code> ioctl commands.</p> <p>If <code>instance</code> is -1, the stream is left unassociated with any hardware instance.</p> <p>If the interface returns failure, the stream is not associated with the specified instance. Any prior association is left untouched.</p> <p>The interface typically fails because of failure to locate and attach the device instance. The interface never fails if the specified instance is -1.</p>
<b>CONTEXT</b>	<code>qassociate()</code> can be called from the stream's <code>put(9E)</code> entry point.
<b>RETURN VALUES</b>	<p>0 Success.</p> <p>-1 Failure.</p>
<b>EXAMPLES</b>	<p>A Style-2 network driver's <code>DL_ATTACH_REQ</code> code would specify:</p> <pre>if (qassociate(q, instance) != 0)     goto fail;</pre> <p>The association prevents Dynamic Reconfiguration (DR) from detaching the instance.</p> <p>A Style-2 network driver's <code>DL_DETACH</code> code would specify:</p> <pre>(void) qassociate(q, -1);</pre>

qassociate(9F)

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)

<b>NAME</b>	qbufcall – call a function when a buffer becomes available
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  bufcall_id_t <b>qbufcall</b>(queue_t *q, size_t size, uint_t pri,     void*funcvoid *arg, void *arg);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>q</i> Pointer to STREAMS queue structure.</p> <p><i>size</i> Number of bytes required for the buffer.</p> <p><i>pri</i> Priority of the allocb(9F) allocation request (not used).</p> <p><i>func</i> Function or driver routine to be called when a buffer becomes available.</p> <p><i>arg</i> Argument to the function to be called when a buffer becomes available.</p>
<b>DESCRIPTION</b>	<p>qbufcall () serves as a qtimeout(9F) call of indeterminate length. When a buffer allocation request fails, qbufcall () can be used to schedule the routine <i>func</i> to be called with the argument <i>arg</i> when a buffer becomes available. <i>func</i> may call allocb () or it may do something else.</p> <p>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, qtimeout(9F) and qunbufcall(9F)) before a driver close routine can block and before the close routine calls qprocsoff(9F).</p> <p>qprocson(9F) must be called before calling either qbufcall () or qtimeout(9F).</p>
<b>RETURN VALUES</b>	If successful, qbufcall () returns a qbufcall ID that can be used in a call to qunbufcall(9F) to cancel the request. If the qbufcall () scheduling fails, <i>func</i> is never called and 0 is returned.
<b>CONTEXT</b>	qbufcall () can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>allocb(9F), mt-streams(9F), qprocson(9F), qtimeout(9F), qunbufcall(9F), qtimeout(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>
<b>WARNINGS</b>	Even when <i>func</i> is called by qbufcall (), allocb(9F) can fail if another module or driver had allocated the memory before <i>func</i> was able to call allocb(9F).

## qenable(9F)

<b>NAME</b>	qenable – enable a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>qenable</b>(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the queue to be enabled.
<b>DESCRIPTION</b>	qenable() adds the queue pointed to by <i>q</i> to the list of queues whose service routines are ready to be called by the STREAMS scheduler.
<b>CONTEXT</b>	qenable() can be called from user or interrupt context.
<b>EXAMPLES</b>	See the dupb(9F) function page for an example of the qenable().
<b>SEE ALSO</b>	dupb(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	qprocson, qprocsoff – enable, disable put and service routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>qprocson</b>(queue_t *q); void <b>qprocsoff</b>(queue_t *q);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>q</i> Pointer to the RD side of a STREAMS queue pair.
<b>DESCRIPTION</b>	<p><b>qprocson</b>( ) enables the put and service routines of the driver or module whose read queue is pointed to by <i>q</i>. Threads cannot enter the module instance through the put and service routines while they are disabled.</p> <p><b>qprocson</b>( ) 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.</p> <p><b>qprocson</b>( ) must be called before calling <b>put</b>(9F), <b>putnext</b>(9F), <b>qbufcall</b>(9F), <b>qtimeout</b>(9F), <b>qwait</b>(9F), or <b>qwait_sig</b>(9F).</p> <p><b>qprocsoff</b>( ) 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.</p> <p>The module or driver instance is guaranteed to be single-threaded before <b>qprocson</b>( ) is called and after <b>qprocsoff</b>( ) is called, except for threads executing asynchronous events such as interrupt handlers and callbacks, which must be handled separately.</p>
<b>CONTEXT</b>	These routines can be called from user or interrupt context.
<b>SEE ALSO</b>	<b>close</b> (9E), <b>open</b> (9E), <b>put</b> (9E), <b>srv</b> (9E), <b>put</b> (9F), <b>putnext</b> (9F), <b>qbufcall</b> (9F), <b>qtimeout</b> (9F), <b>qwait</b> (9F), <b>qwait_sig</b> (9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>
<b>NOTES</b>	The caller may not have the STREAM frozen during either of these calls.

## qreply(9F)

<b>NAME</b>	qreply – send a message on a stream in the reverse direction
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void <b>qreply</b>(queue_t *q, mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to the queue.</p> <p><i>mp</i>           Pointer to the message to be sent in the opposite direction.</p>
<b>DESCRIPTION</b>	qreply() sends messages in the reverse direction of normal flow. That is, qreply( <i>q</i> , <i>mp</i> ) is equivalent to putnext(OTHERQ( <i>q</i> ), <i>mp</i> ).
<b>CONTEXT</b>	qreply() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Canonical Flushing Code for STREAMS Drivers.</p> <p>This example depicts the canonical flushing code for STREAMS drivers. Assume that the driver has service procedures so that there may be messages on its queues. See srv(9E). Its write-side put procedure handles M_FLUSH messages by first checking the FLUSHW bit in the first byte of the message, then the write queue is flushed (line 8) and the FLUSHW bit is turned off (line 9). See put(9E). If the FLUSHR bit is on, then the read queue is flushed (line 12) and the message is sent back up the read side of the stream with the qreply(9F) function (line 13). If the FLUSHR bit is off, then the message is freed (line 15). See the example for flushq(9F) for the canonical flushing code for modules.</p> <pre>1 xxxwput(q, mp) 2   queue_t *q; 3   mblk_t *mp; 4   { 5       switch(mp-&gt;b_datap-&gt;db_type) { 6           case M_FLUSH: 7               if (*mp-&gt;b_rptr &amp; FLUSHW) { 8                   flushq(q, FLUSHALL); 9                   *mp-&gt;b_rptr &amp;= ~FLUSHW; 10              } 11              if (*mp-&gt;b_rptr &amp; FLUSHR) { 12                  flushq(RD(q), FLUSHALL); 13                  qreply(q, mp); 14              } else { 15                  freemsg(mp); 16              } 17              break; 18          . . . 19      }</pre>
<b>SEE ALSO</b>	put(9E), srv(9E), flushq(9F), OTHERQ(9F), putnext(9F) <i>Writing Device Drivers</i>

qreply(9F)

*STREAMS Programming Guide*

## qsize(9F)

<b>NAME</b>	qsize – find the number of messages on a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>qsize</b>(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Queue to be evaluated.
<b>DESCRIPTION</b>	qsize() evaluates the queue <i>q</i> and returns the number of messages it contains.
<b>RETURN VALUES</b>	If there are no message on the queue, qsize() returns 0. Otherwise, it returns the integer representing the number of messages on the queue.
<b>CONTEXT</b>	qsize() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	qtimeout – execute a function after a specified length of time
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  timeout_id_t qtimeout(queue_t *q, void *funcvoid *, void *arg,                     clock_t ticks);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>q</i>            Pointer to STREAMS queue structure.</p> <p><i>func</i>        Kernel function to invoke when the time increment expires.</p> <p><i>arg</i>         Argument to the function.</p> <p><i>ticks</i>       Number of clock ticks to wait before the function is called. Use <code>drv_usec2ohz(9F)</code> to convert microseconds to clock ticks.</p>
<b>DESCRIPTION</b>	<p>The <code>qtimeout()</code> function schedules the specified function <i>func</i> to be called after a specified time interval. <i>func</i> is called with <i>arg</i> 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.</p> <p>The <code>qtimeout()</code> function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See <code>mt-streams(9F)</code>.) <code>qtimeout()</code> schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to <code>qtimeout()</code>. All outstanding timeouts and bufcalls must be cancelled (using, respectively, <code>quntimeout(9F)</code> and <code>qunbufcall(9F)</code>) before a driver close routine can block and before the close routine calls <code>qprocsoff(9F)</code>.</p> <p><code>qprocson(9F)</code> must be called before calling <code>qtimeout()</code>.</p>
<b>RETURN VALUES</b>	<code>qtimeout()</code> returns an opaque non-zero timeout identifier that can be passed to <code>quntimeout(9F)</code> to cancel the request. Note: No value is returned from the called function.
<b>CONTEXT</b>	<code>qtimeout()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<p><code>drv_usec2ohz(9F)</code>, <code>mt-streams(9F)</code>, <code>qbufcall(9F)</code>, <code>qprocson(9F)</code>, <code>qunbufcall(9F)</code>, <code>quntimeout(9F)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## qunbufcall(9F)

<b>NAME</b>	qunbufcall – cancel a pending qbufcall request
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>qunbufcall</b>(queue_t *q, bufcall_id_t id);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>q</i> Pointer to STREAMS queue_t structure.</p> <p><i>id</i> Identifier returned from qbufcall(9F)</p>
<b>DESCRIPTION</b>	<p>qunbufcall () cancels a pending qbufcall () request. The argument <i>id</i> is a non-zero identifier of the request to be cancelled. <i>id</i> is returned from the qbufcall () function used to issue the cancel request.</p> <p>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).</p>
<b>CONTEXT</b>	qunbufcall () can be called from user or interrupt context.
<b>SEE ALSO</b>	mt-streams(9F), qbufcall(9F), qtimeout(9F), quntimeout(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	quntimeout – cancel previous qtimeout function call
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  clock_t <b>quntimeout</b>(queue_t *q, timeout_id_t id);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>q</i>            Pointer to a STREAMS queue structure.</p> <p><i>id</i>            Opaque timeout ID a previous qtimeout(9F) call.</p>
<b>DESCRIPTION</b>	<p>quntimeout () cancels a pending qtimeout(9F) request. The quntimeout () function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See mt-streams(9F).) quntimeout () returns when the timeout has been cancelled or finished executing. The timeout will be cancelled even if it is blocked at the perimeters associated with the queue. quntimeout () should be executed for all outstanding timeouts before a driver or module close returns. All outstanding timeouts and bufcalls must be cancelled before a driver close routine can block and before the close routine calls qprocsoff(9F).</p>
<b>RETURN VALUES</b>	quntimeout () returns -1 if the id is not found. Otherwise, quntimeout () returns a 0 or positive value.
<b>CONTEXT</b>	quntimeout () can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>mt-streams(9F), qbufcall(9F), qtimeout(9F), qunbufcall(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## qwait(9F)

<b>NAME</b>	qwait, qwait_sig – STREAMS wait routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void qwait(queue_t *q); int qwait_sig(queue_t *q);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>qp</i> Pointer to the queue that is being opened or closed.
<b>DESCRIPTION</b>	<p>qwait() and qwait_sig() are used to wait for a message to arrive to the put(9E) or srv(9E) procedures. qwait() and qwait_sig() can also be used to wait for qbufcall(9F) or qtimeout(9F) callback procedures to execute. These routines can be used in the open(9E) and close(9E) procedures in a STREAMS driver or module. qwait() and qwait_sig() atomically exit the inner and outer perimeters associated with the queue, and wait for a thread to leave the module's put(9E), srv(9E), or qbufcall(9F) / qtimeout(9F) callback procedures. Upon return they re-enter the inner and outer perimeters.</p> <p>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.</p> <p>qprocson(9F) must be called before calling qwait() or qwait_sig().</p> <p>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.</p> <p>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).</p>
<b>RETURN VALUES</b>	0 For qwait_sig(), indicates that the condition was not necessarily signaled, and the function returned because a signal was pending.
<b>CONTEXT</b>	These functions can only be called from an open(9E) or close(9E) routine.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using qwait()</p> <p>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.</p> <pre>xxopen(qp, . . .)     queue_t *qp; {     struct xxdata *xx;     /* Allocate xxdata structure */</pre>

**EXAMPLE 1** Using qwait() (Continued)

```

    qprocson(qp);
    /* Format T_INFO_ACK in mp */
    putnext(qp, mp);
    xx->xx_flags |= WAIT_INFO_ACK;
    while (xx->xx_flags & WAIT_INFO_ACK)
        qwait(qp);
    return (0);
}
xxrput(qp, mp)
queue_t *qp;
mblk_t *mp;
{
    struct xxdata *xx = (struct xxdata *)q->q_ptr;

    ...

    case T_INFO_ACK:
        if (xx->xx_flags & WAIT_INFO_ACK) {
            /* Record information from info ack */
            xx->xx_flags &= ~WAIT_INFO_ACK;
            freemsg(mp);
            return;
        }

    ...
}

```

**SEE ALSO** close(9E), open(9E), put(9E), srv(9E), condvar(9F), mt-streams(9F), qbufcall(9F), qprocson(9F), qtimeout(9F)

*STREAMS Programming Guide*

*Writing Device Drivers*

## qwriter(9F)

<b>NAME</b>	qwriter – asynchronous STREAMS perimeter upgrade
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  void <b>qwriter</b>(queue_t *qp, mblk_t *mp, void (*func)(), int perimeter);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI).  <i>qp</i> Pointer to the queue.  <i>mp</i> Pointer to a message that will be passed in to the callback function.  <i>func</i> A function that will be called when exclusive (writer) access has been acquired at the specified perimeter.  <i>perimeter</i> Either PERIM_INNER or PERIM_OUTER.
<b>DESCRIPTION</b>	<p>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:</p> <pre>(*func)(queue_t *qp, mblk_t *mp);</pre> <p>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.</p> <p>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.</p>
<b>CONTEXT</b>	qwriter() can only be called from an put(9E) or srv(9E) routine, or from a qwriter(), qtimeout(9F), or qbufcall(9F) callback function.
<b>SEE ALSO</b>	put(9E), srv(9E), mt-streams(9F), qbufcall(9F), qtimeout(9F)  <i>STREAMS Programming Guide</i>  <i>Writing Device Drivers</i>

<b>NAME</b>	RD, rd – get pointer to the read queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  queue_t *RD(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the write queue whose read queue is to be returned.
<b>DESCRIPTION</b>	<p>The RD() function accepts a write queue pointer as an argument and returns a pointer to the read queue of the same module.</p> <p><b>CAUTION:</b> 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.</p>
<b>RETURN VALUES</b>	The pointer to the read queue.
<b>CONTEXT</b>	RD() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Function page reference</p> <p>See the qreply(9F) function page for an example of RD().</p>
<b>SEE ALSO</b>	<p>qreply(9F), WR(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## rmalloc(9F)

<b>NAME</b>	rmalloc – allocate space from a resource map								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/map.h&gt; #include &lt;sys/ddi.h&gt;  unsigned long <b>rmalloc</b>(struct map *mp, size_t size);</pre>								
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).								
<b>PARAMETERS</b>	<p><i>mp</i> Resource map from where the resource is drawn.</p> <p><i>size</i> Number of units of the resource.</p>								
<b>DESCRIPTION</b>	<p><code>rmalloc()</code> 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 <code>rmallocmap(9F)</code>. <code>rmalloc()</code> is one of five functions used for resource map management. The other functions include:</p> <table><tr><td><code>rmalloc_wait(9F)</code></td><td>Allocate space from a resource map, wait if necessary.</td></tr><tr><td><code>rmfree(9F)</code></td><td>Return previously allocated space to a map.</td></tr><tr><td><code>rmallocmap(9F)</code></td><td>Allocate a resource map and initialize it.</td></tr><tr><td><code>rmfreemap(9F)</code></td><td>Deallocate a resource map.</td></tr></table> <p><code>rmalloc()</code> 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 <code>unsigned long</code> set to the value of the index where sufficient free space in the resource was found.</p>	<code>rmalloc_wait(9F)</code>	Allocate space from a resource map, wait if necessary.	<code>rmfree(9F)</code>	Return previously allocated space to a map.	<code>rmallocmap(9F)</code>	Allocate a resource map and initialize it.	<code>rmfreemap(9F)</code>	Deallocate a resource map.
<code>rmalloc_wait(9F)</code>	Allocate space from a resource map, wait if necessary.								
<code>rmfree(9F)</code>	Return previously allocated space to a map.								
<code>rmallocmap(9F)</code>	Allocate a resource map and initialize it.								
<code>rmfreemap(9F)</code>	Deallocate a resource map.								
<b>RETURN VALUES</b>	Under normal conditions, <code>rmalloc()</code> returns the base index of the allocated space. Otherwise, <code>rmalloc()</code> returns a 0 if all resource map entries are already allocated.								
<b>CONTEXT</b>	<code>rmalloc()</code> can be called from user or interrupt context.								
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Illustrating the principles of map management</p> <p>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 <code>rmallocmap(9F)</code> and <code>rmfree(9F)</code> functions. <code>rmallocmap(9F)</code> is called to establish the number of slots or entries in the map, and <code>rmfree(9F)</code> 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:</p> <ul style="list-style-type: none"><li>■ Panics the system if the required amount of memory can not be allocated (lines 11–15).</li><li>■ Uses <code>rmallocmap(9F)</code> to configure the total number of entries in the map, and <code>rmfree(9F)</code> to initialize the total resource area.</li></ul> <pre>1  #define XX_MAPSIZE      12 2  #define XX_BUFSIZE     2560</pre>								

**EXAMPLE 1** Illustrating the principles of map management (Continued)

```

3  static struct map *xx_mp;          /* Private buffer space map */
4  . . .
5  xxstart( )
6      /*
7       * Allocate private buffer.  If insufficient memory,
8       * display message and halt system.
9       */
10 {
11     register caddr_t bp;
12     . . .
13     if ((bp = kmem_alloc(XX_BUFSIZE, KM_NOSLEEP) == 0) {
14         cmn_err(CE_PANIC, "xxstart: kmem_alloc failed before %d buffer"
15                "allocation", XX_BUFSIZE);
16     }
17     /*
18      * Initialize the resource map with number
19      * of slots in map.
20      */
21     xx_mp = rmallocmap(XX_MAPSIZE);
22
23     /*
24      * Initialize space management map with total
25      * buffer area it is to manage.
26      */
27     rmfree(xx_mp, XX_BUFSIZE, bp);
28     . . .

```

**EXAMPLE 2** Allocating buffers

The `rmalloc()` function is then used by the driver's read or write routine to allocate buffers for specific data transfers. The `uiomove(9F)` function is used to move the data between user space and local driver memory. The device then moves data between itself and local driver memory through DMA.

The next example illustrates the following procedures:

- The size of the I/O request is calculated and stored in the *size* variable (line 10).
- Buffers are allocated through the `rmalloc()` function using the *size* value (line 15). If the allocation fails the system will panic.
- The `uiomove(9F)` function is used to move data to the allocated buffer (line 23).
- If the address passed to `uiomove(9F)` is invalid, `rmfree(9F)` is called to release the previously allocated buffer, and an `EFAULT` error is returned.

```

1  #define XX_BUFSIZE  2560
2  #define XX_MAXSIZE (XX_BUFSIZE / 4)
3
4  static struct map *xx_mp;          /* Private buffer space map */
5  . . .
6  xxread(dev_t dev, uio_t *uiop, cred_t *credp)

```

## rmalloc(9F)

### EXAMPLE 2 Allocating buffers (Continued)

```
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         rmtree(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*

<b>NAME</b>	rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  struct map *rmallocmap(size_t mapsize); struct map *rmallocmap_wait(size_t mapsize); void rmfreemap(struct map *mp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>mapsize</i>            Number of entries for the map.</p> <p><i>mp</i>                    A pointer to the map structure to be deallocated.</p>
<b>DESCRIPTION</b>	<p><code>rmallocmap()</code> dynamically allocates a resource map structure. The argument <i>mapsize</i> 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.</p> <p><code>rmallocmap()</code> 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 <code>rmfree(9F)</code> 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 <code>rmallocmap()</code> is followed by a call to <code>rmfree(9F)</code>, passing the address of the map returned from <code>rmallocmap()</code>, the total size of the resource, and the first index into the actual resource.</p> <p>The resource map allocated by <code>rmallocmap()</code> 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 <code>rmalloc(9F)</code>, <code>rmalloc_wait(9F)</code>, and <code>rmfree(9F)</code>.</p> <p><code>rmallocmap_wait()</code> is similar to <code>rmallocmap()</code>, with the exception that it will wait for space to become available if necessary.</p> <p><code>rmfreemap()</code> deallocates a resource map structure previously allocated by <code>rmallocmap()</code> or <code>rmallocmap_wait()</code>. The argument <i>mp</i> is a pointer to the map structure to be deallocated.</p>
<b>RETURN VALUES</b>	Upon successful completion, <code>rmallocmap()</code> and <code>rmallocmap_wait()</code> return a pointer to the newly allocated map structure. Upon failure, <code>rmallocmap()</code> returns a NULL pointer.
<b>CONTEXT</b>	<p><code>rmallocmap()</code> and <code>rmfreemap()</code> can be called from user, kernel, or interrupt context.</p> <p><code>rmallocmap_wait()</code> can only be called from user or kernel context.</p>
<b>SEE ALSO</b>	<p><code>rmalloc(9F)</code>, <code>rmalloc_wait(9F)</code>, <code>rmfree(9F)</code></p> <p><i>Writing Device Drivers</i></p>

## rmalloc\_wait(9F)

<b>NAME</b>	rmalloc_wait – allocate space from a resource map, wait if necessary
<b>SYNOPSIS</b>	<pre>#include &lt;sys/map.h&gt; #include &lt;sys/ddi.h&gt;  unsigned long <b>rmalloc_wait</b>(struct map *mp, size_t size);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>mp</i>            Pointer to the resource map from which space is to be allocated.</p> <p><i>size</i>          Number of units of space to allocate.</p>
<b>DESCRIPTION</b>	<p>rmalloc_wait () requests an allocation of space from a resource map. rmalloc_wait () is similar to the rmalloc(9F) function with the exception that it will wait for space to become available if necessary.</p>
<b>RETURN VALUES</b>	rmalloc_wait () returns the base of the allocated space.
<b>CONTEXT</b>	This function can be called from user or interrupt context. However, in most cases rmalloc_wait () should be called from user context only.
<b>SEE ALSO</b>	rmalloc(9F), rmallocmap(9F), rmtree(9F), rmtree(9F)
	<i>Writing Device Drivers</i>

<b>NAME</b>	rmfree – free space back into a resource map
<b>SYNOPSIS</b>	<pre>#include &lt;sys/map.h&gt; #include &lt;sys/ddi.h&gt;  void <b>rmfree</b>(struct map *mp, size_t size, ulong_t index);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>mp</i>            Pointer to the map structure.</p> <p><i>size</i>           Number of units being freed.</p> <p><i>index</i>          Index of the first unit of the allocated resource.</p>
<b>DESCRIPTION</b>	<p>rmfree() releases space back into a resource map. It is the opposite of rmalloc(9F), which allocates space that is controlled by a resource map structure.</p> <p>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.</p> <p>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.</p> <p>rmfree() can also be used to initialize a resource map, in which case the size and index should cover the entire resource area.</p>
<b>CONTEXT</b>	rmfree() can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>rmalloc(9F), rmalloc_wait(9F), rmallocmap(9F), rmfreemap(9F)</p> <p><i>Writing Device Drivers</i></p>

rmvb(9F)

<b>NAME</b>	rmvb – remove a message block from a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; mblk_t *rmvb(mblk_t *mp, mblk_t *bp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>mp</i> Message from which a block is to be removed. <i>mblk_t</i> is an instance of the <i>msgb(9S)</i> structure. <i>bp</i> Message block to be removed.
<b>DESCRIPTION</b>	<i>rmvb()</i> removes a message block ( <i>bp</i> ) from a message ( <i>mp</i> ), 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.
<b>RETURN VALUES</b>	If successful, a pointer to the message (minus the removed block) is returned. The pointer is <i>NULL</i> if <i>bp</i> was the only block of the message before <i>rmvb()</i> was called. If the designated message block ( <i>bp</i> ) does not exist, -1 is returned.
<b>CONTEXT</b>	<i>rmvb()</i> can be called from user or interrupt context.
<b>EXAMPLES</b>	This routine removes all zero-length <i>M_DATA</i> 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 <i>M_DATA</i> 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). <pre>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-&gt;b_cont; 11        if ((tmp-&gt;b_datap-&gt;db_type == M_DATA) &amp;&amp; 12            (tmp-&gt;b_rptr == tmp-&gt;b_wptr)) { 13            (void) rmbv(mp, tmp); 14            freeb(tmp); 15        } 16        tmp = nmp; 17    }</pre>
<b>SEE ALSO</b>	<i>freeb(9F)</i> , <i>msgb(9S)</i>  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>

<b>NAME</b>	rmvq – remove a message from a queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void <b>rmvq</b>(queue_t *q, mblk_t *mp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>q</i> Queue containing the message to be removed.</p> <p><i>mp</i> Message to remove.</p>
<b>DESCRIPTION</b>	<p>rmvq() removes a message from a queue. A message can be removed from anywhere on a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either rmvq() or getq(9F) should be used to remove a message from a queue.</p>
<b>CONTEXT</b>	rmvq() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p>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).</p> <pre> 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-&gt;q_first; 7  while (mp) { 8      if (mp-&gt;b_datap-&gt;db_type == M_PROTO) { 9          tp = (union T_primitives *)mp-&gt;b_rptr; 10         if (tp-&gt;type == T_DATA_IND) { 11             nmp = mp-&gt;b_next; 12             rmvq(q, mp); 13             freemsg(mp); 14             mp = nmp; 15         } else { 16             mp = mp-&gt;b_next; 17         } 18     } else { 19         mp = mp-&gt;b_next; 20     } 21 } 22 /* End of region that must be protected */</pre> <p>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.</p>
<b>SEE ALSO</b>	freemsg(9F), getq(9F), insq(9F)
	<i>Writing Device Drivers</i>

rmvq(9F)

*STREAMS Programming Guide*

**WARNINGS** Make sure that the message *mp* is linked onto *q* to avoid a possible system panic.

<b>NAME</b>	rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers/writer lock functions										
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ksynch.h&gt;  void <b>rw_init</b>(krwlock_t *rwlp, char *name, krw_type_t type, void              *arg);  void <b>rw_destroy</b>(krwlock_t *rwlp);  void <b>rw_enter</b>(krwlock_t *rwlp, krw_t enter_type);  void <b>rw_exit</b>(krwlock_t *rwlp);  int <b>rw_tryenter</b>(krwlock_t *rwlp, krw_t enter_type);  void <b>rw_downgrade</b>(krwlock_t *rwlp);  int <b>rw_tryupgrade</b>(krwlock_t *rwlp);  int <b>rw_read_locked</b>(krwlock_t *rwlp);</pre>										
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <table border="0"> <tr> <td style="padding-right: 20px;"><i>rwlp</i></td> <td>Pointer to a <code>krwlock_t</code> readers/writer lock.</td> </tr> <tr> <td><i>name</i></td> <td>Descriptive string. This is obsolete and should be <code>NULL</code>. (Non-null strings are legal, but they're a waste of kernel memory.)</td> </tr> <tr> <td><i>type</i></td> <td>Type of readers/writer lock.</td> </tr> <tr> <td><i>arg</i></td> <td>Type-specific argument for initialization function.</td> </tr> <tr> <td><i>enter_type</i></td> <td>One of the values <code>RW_READER</code> or <code>RW_WRITER</code>, indicating whether the lock is to be acquired non-exclusively (<code>RW_READER</code>) or exclusively (<code>RW_WRITER</code>).</td> </tr> </table>	<i>rwlp</i>	Pointer to a <code>krwlock_t</code> readers/writer lock.	<i>name</i>	Descriptive string. This is obsolete and should be <code>NULL</code> . (Non-null strings are legal, but they're a waste of kernel memory.)	<i>type</i>	Type of readers/writer lock.	<i>arg</i>	Type-specific argument for initialization function.	<i>enter_type</i>	One of the values <code>RW_READER</code> or <code>RW_WRITER</code> , indicating whether the lock is to be acquired non-exclusively ( <code>RW_READER</code> ) or exclusively ( <code>RW_WRITER</code> ).
<i>rwlp</i>	Pointer to a <code>krwlock_t</code> readers/writer lock.										
<i>name</i>	Descriptive string. This is obsolete and should be <code>NULL</code> . (Non-null strings are legal, but they're a waste of kernel memory.)										
<i>type</i>	Type of readers/writer lock.										
<i>arg</i>	Type-specific argument for initialization function.										
<i>enter_type</i>	One of the values <code>RW_READER</code> or <code>RW_WRITER</code> , indicating whether the lock is to be acquired non-exclusively ( <code>RW_READER</code> ) or exclusively ( <code>RW_WRITER</code> ).										
<b>DESCRIPTION</b>	<p>A multiple-readers, single-writer lock is represented by the <code>krwlock_t</code> 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.</p> <p>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.</p> <p><code>rw_init()</code> initializes a readers/writer lock. It is an error to initialize a lock more than once. The <i>type</i> argument should be set to <code>RW_DRIVER</code>. If the lock is used by the interrupt handler, the type-specific argument, <i>arg</i>, should be the <code>ddi_iblock_cookie</code> returned from <code>ddi_get_iblock_cookie(9F)</code> or <code>ddi_get_soft_iblock_cookie(9F)</code>. If the lock is not used by any interrupt handler, the argument should be <code>NULL</code>.</p>										

## rwlock(9F)

`rw_destroy()` releases any resources that might have been allocated by `rw_init()`. It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

`rw_enter()` acquires the lock, and blocks if necessary. If `enter_type` is `RW_READER`, the caller blocks if there is a writer or a thread attempting to enter for writing. If `enter_type` is `RW_WRITER`, the caller blocks if any thread holds the lock.

**NOTE:** It is a programming error for any thread to acquire an `rwlock` it already holds, even as a reader. Doing so can deadlock the system: if thread *R* acquires the lock as a reader, then thread *W* tries to acquire the lock as a writer, *W* will set write-wanted and block. When *R* tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for *W*; but *W* cannot run until *R* drops the lock. Thus threads *R* and *W* deadlock.

`rw_exit()` releases the lock and may wake up one or more threads waiting on the lock.

`rw_tryenter()` attempts to enter the lock, like `rw_enter()`, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with `RW_WRITER`), may call `rw_downgrade()` to convert to holding the lock non-exclusively (as if entered with `RW_READER`). One or more waiting readers may be unblocked.

`rw_tryupgrade()` can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

`rw_read_locked()` returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if `rw_read_locked()` is called for a lock that isn't held by the caller.

### RETURN VALUES

0	<code>rw_tryenter()</code> could not obtain the lock without blocking.
0	<code>rw_tryupgrade()</code> was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
0	<code>rw_read_locked()</code> returns 0 if the lock is held by the caller for write.
non-zero	from <code>rw_read_locked()</code> if the lock is held by the caller for read.
non-zero	successful return from <code>rw_tryenter()</code> or <code>rw_tryupgrade()</code> .

### CONTEXT

These functions can be called from user or interrupt context, except for `rw_init()` and `rw_destroy()`, which can be called from user context only.

rwlock(9F)

**SEE ALSO** condvar(9F), ddi\_add\_intr(9F), ddi\_get\_iblock\_cookie(9F),  
ddi\_get\_soft\_iblock\_cookie(9F), mutex(9F), semaphore(9F)

*Writing Device Drivers*

**NOTES** Compiling with `_LOCKTEST` or `_MPSTATS` defined no longer has any effect. To gather lock statistics, see `lockstat(1M)`.

## SAMESTR(9F)

<b>NAME</b>	SAMESTR, samestr – test if next queue is in the same stream
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int SAMESTR(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the queue.
<b>DESCRIPTION</b>	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 <i>q_next</i> field of queue(9S) to see if the stream continues beyond <i>q</i> .
<b>RETURN VALUES</b>	SAMESTR() returns 1 if the next queue is the same type as the current queue. It returns 0 if the next queue does not exist or if it is not the same type.
<b>CONTEXT</b>	SAMESTR() can be called from user or interrupt context.
<b>SEE ALSO</b>	OTHERQ(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	scsi_abort – abort a SCSI command
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_abort</b>(struct scsi_address *ap, struct scsi_pkt *pkt);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>ap</i>            Pointer to a scsi_address structure.</p> <p><i>pkt</i>            Pointer to a scsi_pkt(9S) structure.</p>
<b>DESCRIPTION</b>	scsi_abort () terminates a command that has been transported to the host adapter driver. A NULL <i>pkt</i> causes all outstanding packets to be aborted. On a successful abort, the <i>pkt_reason</i> is set to CMD_ABORTED and <i>pkt_statistics</i> is OR'ed with STAT_ABORTED.
<b>RETURN VALUES</b>	<p>scsi_abort () returns:</p> <p>1            on success.</p> <p>0            on failure.</p>
<b>CONTEXT</b>	scsi_abort () can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Terminating a command.</p> <pre>if (scsi_abort(&amp;devp-&gt;sd_address, pkt) == 0) {     (void) scsi_reset(&amp;devp-&gt;sd_address, RESET_ALL); }</pre>
<b>SEE ALSO</b>	<p>tran_abort(9E), scsi_reset(9F), scsi_pkt(9S)</p> <p><i>Writing Device Drivers</i></p>

## scsi\_alloc\_consistent\_buf(9F)

<b>NAME</b>	scsi_alloc_consistent_buf – allocate an I/O buffer for SCSI DMA												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  struct buf *scsi_alloc_consistent_buf(struct scsi_address*ap,     struct buf *bp, size_t datalen, uint_t bflags, int     (*callback)(caddr_t), caddr_t arg);</pre>												
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).												
<b>PARAMETERS</b>	<table><tr><td><i>ap</i></td><td>Pointer to the <code>scsi_address(9S)</code> structure.</td></tr><tr><td><i>bp</i></td><td>Pointer to the <code>buf(9S)</code> structure.</td></tr><tr><td><i>datalen</i></td><td>Number of bytes for the data buffer.</td></tr><tr><td><i>bflags</i></td><td>Flags setting for the allocated buffer header. This should either be <code>B_READ</code> or <code>B_WRITE</code>.</td></tr><tr><td><i>callback</i></td><td>A pointer to a callback function, <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code>.</td></tr><tr><td><i>arg</i></td><td>The callback function argument.</td></tr></table>	<i>ap</i>	Pointer to the <code>scsi_address(9S)</code> structure.	<i>bp</i>	Pointer to the <code>buf(9S)</code> structure.	<i>datalen</i>	Number of bytes for the data buffer.	<i>bflags</i>	Flags setting for the allocated buffer header. This should either be <code>B_READ</code> or <code>B_WRITE</code> .	<i>callback</i>	A pointer to a callback function, <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code> .	<i>arg</i>	The callback function argument.
<i>ap</i>	Pointer to the <code>scsi_address(9S)</code> structure.												
<i>bp</i>	Pointer to the <code>buf(9S)</code> structure.												
<i>datalen</i>	Number of bytes for the data buffer.												
<i>bflags</i>	Flags setting for the allocated buffer header. This should either be <code>B_READ</code> or <code>B_WRITE</code> .												
<i>callback</i>	A pointer to a callback function, <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code> .												
<i>arg</i>	The callback function argument.												
<b>DESCRIPTION</b>	<p><code>scsi_alloc_consistent_buf()</code> allocates a buffer header and the associated data buffer for direct memory access (DMA) transfer. This buffer is allocated from the <code>iobp</code> space, which is considered consistent memory. For more details, see <code>ddi_dma_mem_alloc(9F)</code> and <code>ddi_dma_sync(9F)</code>.</p> <p>For buffers allocated via <code>scsi_alloc_consistent_buf()</code>, and marked with the <code>PKT_CONSISTENT</code> flag via <code>scsi_init_pkt(9F)</code>, 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.</p> <p>If <i>bp</i> is <code>NULL</code>, a new buffer header will be allocated using <code>getrbuf(9F)</code>. In addition, if <i>datalen</i> is non-zero, a new buffer will be allocated using <code>ddi_dma_mem_alloc(9F)</code>.</p> <p><i>callback</i> indicates what the allocator routines should do when direct memory access (DMA) resources are not available; the valid values are:</p> <table><tr><td><code>NULL_FUNC</code></td><td>Do not wait for resources. Return a <code>NULL</code> pointer.</td></tr><tr><td><code>SLEEP_FUNC</code></td><td>Wait indefinitely for resources.</td></tr><tr><td>Other Values</td><td><i>callback</i> points to a function that is called when resources may become available. <i>callback</i> 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 <i>arg</i> is supplied to the <i>callback</i> function when it is invoked.</td></tr></table>	<code>NULL_FUNC</code>	Do not wait for resources. Return a <code>NULL</code> pointer.	<code>SLEEP_FUNC</code>	Wait indefinitely for resources.	Other Values	<i>callback</i> points to a function that is called when resources may become available. <i>callback</i> 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 <i>arg</i> is supplied to the <i>callback</i> function when it is invoked.						
<code>NULL_FUNC</code>	Do not wait for resources. Return a <code>NULL</code> pointer.												
<code>SLEEP_FUNC</code>	Wait indefinitely for resources.												
Other Values	<i>callback</i> points to a function that is called when resources may become available. <i>callback</i> 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 <i>arg</i> is supplied to the <i>callback</i> function when it is invoked.												

**RETURN VALUES** `scsi_alloc_consistent_buf()` returns a pointer to a `buf(9S)` structure on success. It returns `NULL` if resources are not available even if `waitfunc` was not `SLEEP_FUNC`.

**CONTEXT** If `callback` is `SLEEP_FUNC`, then this routine may be called only from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block.

**EXAMPLES**

**EXAMPLE 1** Allocate a request sense packet with consistent DMA resources attached.

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
    SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
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*

## scsi\_cname(9F)

<b>NAME</b>	scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name												
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  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);</pre>												
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <table><tr><td><i>cmd</i></td><td>A SCSI command value.</td></tr><tr><td><i>cmdvec</i></td><td>Pointer to an array of command strings.</td></tr><tr><td><i>dtype</i></td><td>Device type.</td></tr><tr><td><i>msg</i></td><td>A message value.</td></tr><tr><td><i>reason</i></td><td>A packet reason value.</td></tr><tr><td><i>sense_key</i></td><td>A SCSI sense key value.</td></tr></table>	<i>cmd</i>	A SCSI command value.	<i>cmdvec</i>	Pointer to an array of command strings.	<i>dtype</i>	Device type.	<i>msg</i>	A message value.	<i>reason</i>	A packet reason value.	<i>sense_key</i>	A SCSI sense key value.
<i>cmd</i>	A SCSI command value.												
<i>cmdvec</i>	Pointer to an array of command strings.												
<i>dtype</i>	Device type.												
<i>msg</i>	A message value.												
<i>reason</i>	A packet reason value.												
<i>sense_key</i>	A SCSI sense key value.												
<b>DESCRIPTION</b>	<p><i>scsi_cname()</i> decodes SCSI commands. <i>cmdvec</i> 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.</p> <p><i>scsi_dname()</i> decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.</p> <p><i>scsi_mname()</i> decodes SCSI messages.</p> <p><i>scsi_rname()</i> decodes packet completion reasons.</p> <p><i>scsi_sname()</i> decodes SCSI sense keys.</p>												
<b>RETURN VALUES</b>	These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.												
<b>CONTEXT</b>	These functions can be called from user or interrupt context.												
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Decoding SCSI tape commands.</p> <p><i>scsi_cname()</i> decodes SCSI tape commands as follows:</p> <pre>static char *st_cmds[] = {     "\000test unit ready",     "\001rewind",     "\003request sense",     "\010read",     "\012write",</pre>												

**EXAMPLE 1** Decoding SCSI tape commands. *(Continued)*

```
        "\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*

## scsi\_destroy\_pkt(9F)

<b>NAME</b>	scsi_destroy_pkt – free an allocated SCSI packet and its DMA resource
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_destroy_pkt</b>(struct scsi_pkt *pktp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>pktp</i> Pointer to a scsi_pkt(9S) structure.
<b>DESCRIPTION</b>	scsi_destroy_pkt() releases all necessary resources, typically at the end of an I/O transfer. The data is synchronized to memory, then the DMA resources are deallocated and <i>pktp</i> is freed.
<b>CONTEXT</b>	scsi_destroy_pkt() may be called from user or interrupt context.
<b>EXAMPLES</b>	<b>EXAMPLE 1</b> Releasing resources. <pre>scsi_destroy_pkt(un-&gt;un_rqs);</pre>
<b>SEE ALSO</b>	tran_destroy_pkt(9E), scsi_init_pkt(9F), scsi_pkt(9S) <i>Writing Device Drivers</i>

<b>NAME</b>	scsi_dmaget, scsi_dmafree – SCSI dma utility routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  struct scsi_pkt *<b>scsi_dmaget</b>(struct scsi_pkt *pkt, opaque_t     dmatoken, int (*callback) (void));  void <b>scsi_dmafree</b>(struct scsi_pkt *pkt);</pre>
<b>INTERFACE LEVEL</b>	These interfaces are obsolete. Use <code>scsi_init_pkt(9F)</code> instead of <code>scsi_dmaget()</code> . Use <code>scsi_destroy_pkt(9F)</code> instead of <code>scsi_dmafree()</code> .
<b>PARAMETERS</b>	<p><i>pkt</i>                    A pointer to a <code>scsi_pkt(9S)</code> structure.</p> <p><i>dmatoken</i>               Pointer to an implementation dependent object</p> <p><i>callback</i>               Pointer to a callback function, or <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code>.</p>
<b>DESCRIPTION</b>	<p><code>scsi_dmaget()</code> allocates DMA resources for an already allocated SCSI packet. <i>pkt</i> is a pointer to the previously allocated SCSI packet (see <code>scsi_pktalloc(9F)</code>).</p> <p><i>dmatoken</i> 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 <i>dmatoken</i> must be a pointer to a <code>buf(9S)</code> structure. If <i>dmatoken</i> is <code>NULL</code>, no resources are allocated.</p> <p><i>callback</i> indicates what <code>scsi_dmaget()</code> should do when resources are not available:</p> <p><code>NULL_FUNC</code>            Do not wait for resources. Return a <code>NULL</code> pointer.</p> <p><code>SLEEP_FUNC</code>           Wait indefinitely for resources.</p> <p>Other Values           <i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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.</p> <p><code>scsi_dmafree()</code> frees the DMA resources associated with the SCSI packet. The packet itself remains allocated.</p>
<b>RETURN VALUES</b>	<code>scsi_dmaget()</code> returns a pointer to a <code>scsi_pkt</code> on success. It returns <code>NULL</code> if resources are not available.
<b>CONTEXT</b>	<p>If <i>callback</i> is <code>SLEEP_FUNC</code>, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The <i>callback</i> function may not block or call routines that block.</p> <p><code>scsi_dmafree()</code> can be called from user or interrupt context.</p>

scsi\_dmaget(9F)

**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)`.

<b>NAME</b>	scsi_errmsg – display a SCSI request sense message																
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_errmsg</b>(struct scsi_device *devp, struct scsi_pkt *pktp,     char *drv_name, int severity, daddr_t blkno, daddr_t err_blkno,     struct scsi_key_strings *cmdlist, struct scsi_extended_sense     *sensep);</pre>																
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <table border="0"> <tr> <td><i>devp</i></td> <td>Pointer to the <i>scsi_device</i>(9S) structure.</td> </tr> <tr> <td><i>pktp</i></td> <td>Pointer to a <i>scsi_pkt</i>(9S) structure.</td> </tr> <tr> <td><i>drv_name</i></td> <td>String used by <i>scsi_log</i>(9F).</td> </tr> <tr> <td><i>severity</i></td> <td>Error severity level, maps to severity strings below.</td> </tr> <tr> <td><i>blkno</i></td> <td>Requested block number.</td> </tr> <tr> <td><i>err_blkno</i></td> <td>Error block number.</td> </tr> <tr> <td><i>cmdlist</i></td> <td>An array of SCSI command description strings.</td> </tr> <tr> <td><i>sensep</i></td> <td>A pointer to a <i>scsi_extended_sense</i>(9S) structure.</td> </tr> </table>	<i>devp</i>	Pointer to the <i>scsi_device</i> (9S) structure.	<i>pktp</i>	Pointer to a <i>scsi_pkt</i> (9S) structure.	<i>drv_name</i>	String used by <i>scsi_log</i> (9F).	<i>severity</i>	Error severity level, maps to severity strings below.	<i>blkno</i>	Requested block number.	<i>err_blkno</i>	Error block number.	<i>cmdlist</i>	An array of SCSI command description strings.	<i>sensep</i>	A pointer to a <i>scsi_extended_sense</i> (9S) structure.
<i>devp</i>	Pointer to the <i>scsi_device</i> (9S) structure.																
<i>pktp</i>	Pointer to a <i>scsi_pkt</i> (9S) structure.																
<i>drv_name</i>	String used by <i>scsi_log</i> (9F).																
<i>severity</i>	Error severity level, maps to severity strings below.																
<i>blkno</i>	Requested block number.																
<i>err_blkno</i>	Error block number.																
<i>cmdlist</i>	An array of SCSI command description strings.																
<i>sensep</i>	A pointer to a <i>scsi_extended_sense</i> (9S) structure.																
<b>DESCRIPTION</b>	<p><i>scsi_errmsg</i> () interprets the request sense information in the <i>sensep</i> pointer and generates a standard message that is displayed using <i>scsi_log</i>(9F). The first line of the message is always a <i>CE_WARN</i>, with the continuation lines being <i>CE_CONT</i>. <i>sensep</i> may be <i>NULL</i>, in which case no sense key or vendor information is displayed.</p> <p>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.</p> <p>The <i>scsi_device</i>(9S) structure denoted by <i>devp</i> supplies the identification of the device that requested the display. <i>severity</i> selects which string is used in the "Error Level:" reporting, according to the following table:</p> <table border="0"> <thead> <tr> <th style="text-align: left;"><b>Severity Value:</b></th> <th style="text-align: left;"><b>String:</b></th> </tr> </thead> <tbody> <tr> <td>SCSI_ERR_ALL</td> <td>All</td> </tr> <tr> <td>SCSI_ERR_UNKNOWN</td> <td>Unknown</td> </tr> <tr> <td>SCSI_ERR_INFO</td> <td>Informational</td> </tr> <tr> <td>SCSI_ERR_RECOVERE</td> <td>Recovered</td> </tr> <tr> <td>SCSI_ERR_RETRYABL</td> <td>Retryable</td> </tr> <tr> <td>SCSI_ERR_FATAL</td> <td>Fatal</td> </tr> </tbody> </table>	<b>Severity Value:</b>	<b>String:</b>	SCSI_ERR_ALL	All	SCSI_ERR_UNKNOWN	Unknown	SCSI_ERR_INFO	Informational	SCSI_ERR_RECOVERE	Recovered	SCSI_ERR_RETRYABL	Retryable	SCSI_ERR_FATAL	Fatal		
<b>Severity Value:</b>	<b>String:</b>																
SCSI_ERR_ALL	All																
SCSI_ERR_UNKNOWN	Unknown																
SCSI_ERR_INFO	Informational																
SCSI_ERR_RECOVERE	Recovered																
SCSI_ERR_RETRYABL	Retryable																
SCSI_ERR_FATAL	Fatal																

## scsi\_errmsg(9F)

*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** `scsi_errmsg()` may be called from user or interrupt 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);
```

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
```

scsi\_errmsg(9F)

**SEE ALSO** | `cmn_err(9F)`, `scsi_log(9F)`, `scsi_device(9S)`, `scsi_extended_sense(9S)`,  
`scsi_pkt(9S)`

*Writing Device Drivers*

## scsi\_free\_consistent\_buf(9F)

<b>NAME</b>	scsi_free_consistent_buf – free a previously allocated SCSI DMA I/O buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_free_consistent_buf</b>(struct buf *bp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>bp</i> Pointer to the buf(9S) structure.
<b>DESCRIPTION</b>	scsi_free_consistent_buf() frees a buffer header and consistent data buffer that was previously allocated using scsi_alloc_consistent_buf(9F).
<b>CONTEXT</b>	scsi_free_consistent_buf() may be called from either the user or the interrupt levels.
<b>SEE ALSO</b>	freerbuf(9F), scsi_alloc_consistent_buf(9F), buf(9S)  <i>Writing Device Drivers</i>
<b>WARNING</b>	scsi_free_consistent_buf() will call freerbuf(9F) to free the buf(9S) that was allocated before or during the call to scsi_alloc_consistent_buf(9F).  If consistent memory is bound to a scsi_pkt(9S), the pkt should be destroyed before freeing the consistent memory.

<b>NAME</b>	scsi_get_device_type_scsi_options – look up per-device-type scsi-options property
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_get_device_type_scsi_options</b>(dev_info_t *dip, struct     scsi_device *devp, int default_scsi_options);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i> Pointer to the device info node for this HBA driver.</p> <p><i>devp</i> Pointer to a <code>scsi_device(9S)</code> structure of the target.</p> <p><i>default_scsi_options</i> Value returned if no match is found.</p>
<b>DESCRIPTION</b>	<p>The <code>scsi_get_device_type_scsi_options()</code> function looks up the property <code>device-type-scsi-options-list</code>, which can be specified in the HBA's <code>driver.conf(4)</code> file. This property allows specification of <code>scsi-options</code> on a per-device-type basis.</p> <p>The formal syntax is:</p> <pre>device-type-scsi-options-list = &lt;duplet&gt; [, &lt;duplet&gt; *];</pre> <p>where:</p> <pre>&lt;duplet&gt; := &lt;vid+pid&gt;, &lt;scsi-options-property-name&gt;</pre> <p>and:</p> <pre>&lt;scsi-options-property-name&gt; = &lt;value&gt;;</pre> <p>The string <code>&lt;vid+pid&gt;</code> 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.</p> <p>For example:</p> <pre>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;</pre> <p>The <code>scsi_get_device_type_scsi_options()</code> function searches the list of duplets for a matching INQUIRY string. If a match is found, <code>scsi_get_device_type_scsi_options()</code> returns the corresponding value.</p>

scsi\_get\_device\_type\_scsi\_options(9F)

**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*

<b>NAME</b>	scsi_hba_attach_setup, scsi_hba_attach, scsi_hba_detach – SCSI HBA attach and detach routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_hba_attach_setup</b>(dev_info_t *dip, ddi_dma_attr_t     *hba_dma_attr, scsi_hba_tran_t *hba_tran, int hba_flags);  int <b>scsi_hba_attach</b>(dev_info_t *dip, ddi_dma_lim_t *hba_lim,     scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);  int <b>scsi_hba_detach</b>(dev_info_t *dip);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris architecture specific (Solaris DDI).</p> <p><i>dip</i>                    A pointer to the dev_info_t structure, referring to the instance of the HBA device.</p> <p><i>hba_lim</i>                A pointer to a ddi_dma_lim(9S) structure.</p> <p><i>hba_tran</i>               A pointer to a scsi_hba_tran(9S) structure.</p> <p><i>hba_flags</i>              Flag modifiers. The only defined flag value is SCSI_HBA_TRAN_CLONE.</p> <p><i>hba_options</i>            Optional features provided by the HBA driver for future extensions; must be NULL.</p> <p><i>hba_dma_attr</i>          A pointer to a ddi_dma_attr(9S) structure.</p>
<b>DESCRIPTION</b>	<p>scsi_hba_attach_setup() is the recommended interface over scsi_hba_attach().</p> <p>For scsi_hba_attach_setup() and scsi_hba_attach():</p> <p>scsi_hba_attach() registers the DMA limits <i>hba_lim</i> and the transport vectors <i>hba_tran</i> of each instance of the HBA device defined by <i>dip</i>.</p> <p>scsi_hba_attach_setup() registers the DMA attributes <i>hba_dma_attr</i> and the transport vectors <i>hba_tran</i> of each instance of the HBA device defined by <i>dip</i>. 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.</p> <p>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().</p> <p>If SCSI_HBA_TRAN_CLONE is requested in <i>hba_flags</i>, the <i>hba_tran</i> 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</p>

## scsi\_hba\_attach\_setup(9F)

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 Queuing 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 SCSI mode.

`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\_hba\_attach\_setup(9F)

`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()`.

## scsi\_hba\_init(9F)

<b>NAME</b>	scsi_hba_init, scsi_hba_fini – SCSI Host Bus Adapter system initialization and completion routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_hba_init</b>(struct modlinkage *modlp); void <b>scsi_hba_fini</b>(struct modlinkage *modlp);</pre>
<b>INTERFACE LEVEL</b>	Solaris architecture specific (Solaris DDI).
<b>PARAMETERS</b>	<i>modlp</i> Pointer to the Host Bus Adapters module linkage structure.
<b>DESCRIPTION</b>	
<b>scsi_hba_init()</b>	<i>scsi_hba_init()</i> is the system-provided initialization routine for SCSI HBA drivers. The <i>scsi_hba_init()</i> function registers the HBA in the system and allows the driver to accept configuration requests on behalf of SCSI target drivers. The <i>scsi_hba_init()</i> routine must be called in the HBA's <i>_init(9E)</i> routine before <i>mod_install(9F)</i> is called. If <i>mod_install(9F)</i> fails, the HBA's <i>_init(9E)</i> should call <i>scsi_hba_fini()</i> before returning failure.
<b>scsi_hba_fini()</b>	<i>scsi_hba_fini()</i> is the system provided completion routine for SCSI HBA drivers. <i>scsi_hba_fini()</i> removes all of the system references for the HBA that were created in <i>scsi_hba_init()</i> . The <i>scsi_hba_fini()</i> routine should be called in the HBA's <i>_fini(9E)</i> routine if <i>mod_remove(9F)</i> is successful.
<b>RETURN VALUES</b>	<i>scsi_hba_init()</i> returns 0 if successful, and a non-zero value otherwise. If <i>scsi_hba_init()</i> fails, the HBA's <i>_init()</i> entry point should return the value returned by <i>scsi_hba_init()</i> .
<b>CONTEXT</b>	<i>scsi_hba_init()</i> and <i>scsi_hba_fini()</i> should be called from <i>_init(9E)</i> or <i>_fini(9E)</i> , respectively.
<b>SEE ALSO</b>	<i>_fini(9E)</i> , <i>_init(9E)</i> , <i>mod_install(9F)</i> , <i>mod_remove(9F)</i> , <i>scsi_pktalloc(9F)</i> , <i>scsi_pktfree(9F)</i> , <i>scsi_hba_tran(9S)</i>
	<i>Writing Device Drivers</i>
<b>NOTES</b>	The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once <i>scsi_hba_fini()</i> is called.

<b>NAME</b>	scsi_hba_lookup_capstr – return index matching capability string
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_hba_lookup_capstr</b>(char *capstr);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris architecture specific (Solaris DDI). <i>capstr</i> Pointer to a string.
<b>DESCRIPTION</b>	<p><code>scsi_hba_lookup_capstr()</code> attempts to match <i>capstr</i> against a known set of capability strings, and returns the defined index for the matched capability, if found.</p> <p>The set of indices and capability strings is:</p> <pre>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"</pre>

## scsi\_hba\_lookup\_capstr(9F)

```
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"
```

**RETURN VALUES** `scsi_hba_lookup_capstr()` returns a non-negative index value corresponding to the capability string, or -1 if the string does not match any known capability.

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

**SEE ALSO** `tran_getcap(9E)`, `tran_setcap(9E)`, `scsi_ifgetcap(9F)`, `scsi_ifsetcap(9F)`, `scsi_reset_notify(9F)`

*Writing Device Drivers*

<b>NAME</b>	scsi_hba_pkt_alloc, scsi_hba_pkt_free – allocate and free a scsi_pkt structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris architecture specific (Solaris DDI).</p> <p><i>dip</i> Pointer to a dev_info_t structure, defining the HBA driver instance.</p> <p><i>ap</i> Pointer to a scsi_address(9S) structure, defining the target instance.</p> <p><i>cmdlen</i> Length in bytes to be allocated for the SCSI command descriptor block (CDB).</p> <p><i>statuslen</i> Length in bytes to be allocated for the SCSI status completion block (SCB).</p> <p><i>tgtlen</i> Length in bytes to be allocated for a private data area for the target driver's exclusive use.</p> <p><i>hbalen</i> Length in bytes to be allocated for a private data area for the HBA driver's exclusive use.</p> <p><i>callback</i> Indicates what scsi_hba_pkt_alloc() should do when resources are not available:</p> <p style="padding-left: 2em;">NULL_FUNC Do not wait for resources. Return a NULL pointer.</p> <p style="padding-left: 2em;">SLEEP_FUNC Wait indefinitely for resources.</p> <p><i>arg</i> Must be NULL.</p> <p><i>pkt</i> A pointer to a scsi_pkt(9S) structure.</p>
<b>DESCRIPTION</b>	<p>For scsi_hba_pkt_alloc():</p> <p>scsi_hba_pkt_alloc() 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.</p> <p>If <i>callback</i> is NULL_FUNC, scsi_hba_pkt_alloc() may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.</p> <p>scsi_hba_pkt_alloc() copies the scsi_address(9S) structure pointed to by <i>ap</i> to the pkt_address field in the scsi_pkt(9S).</p>

## scsi\_hba\_pkt\_alloc(9F)

`scsi_hba_pkt_alloc()` also allocates memory for these `scsi_pkt(9S)` data areas, and sets these fields to point to the allocated memory:

<code>pkt_ha_private</code>	HBA private data area.
<code>pkt_private</code>	Target driver private data area.
<code>pkt_scbp</code>	SCSI status completion block.
<code>pkt_cdbp</code>	SCSI command descriptor block.

For `scsi_hba_pkt_free()`:

`scsi_hba_pkt_free()` frees the space allocated for the `scsi_pkt(9S)` structure.

### RETURN VALUES

`scsi_hba_pkt_alloc()` returns a pointer to the `scsi_pkt` structure, or `NULL` if no space is available.

### CONTEXT

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

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

### SEE ALSO

`tran_init_pkt(9E)`, `scsi_address(9S)`, `scsi_pkt(9S)`

*Writing Device Drivers*

<b>NAME</b>	scsi_hba_probe – default SCSI HBA probe function
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_hba_probe</b>(struct scsi_device *sd, int (*waitfunc)(void));</pre>
<b>INTERFACE LEVEL</b>	Solaris architecture specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>sd</i> Pointer to a scsi_device(9S) structure describing the target.</p> <p><i>waitfunc</i> NULL_FUNC or SLEEP_FUNC.</p>
<b>DESCRIPTION</b>	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.
<b>RETURN VALUES</b>	See scsi_probe(9F) for the return values from scsi_hba_probe().
<b>CONTEXT</b>	scsi_hba_probe() should only be called from the HBA's tran_tgt_probe(9E) entry point.
<b>SEE ALSO</b>	tran_tgt_probe(9E), scsi_probe(9F), scsi_device(9S)
	<i>Writing Device Drivers</i>

## scsi\_hba\_tran\_alloc(9F)

<b>NAME</b>	scsi_hba_tran_alloc, scsi_hba_tran_free – allocate and free transport structures						
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  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);</pre>						
<b>INTERFACE LEVEL</b>	Solaris architecture specific (Solaris DDI).						
<b>PARAMETERS</b>	<table><tr><td><i>dip</i></td><td>Pointer to a dev_info structure, defining the HBA driver instance.</td></tr><tr><td><i>flag</i></td><td>Flag modifiers. The only possible flag value is SCSI_HBA_CANSLEEP (memory allocation may sleep).</td></tr><tr><td><i>hba_tran</i></td><td>Pointer to a scsi_hba_tran(9S) structure.</td></tr></table>	<i>dip</i>	Pointer to a dev_info structure, defining the HBA driver instance.	<i>flag</i>	Flag modifiers. The only possible flag value is SCSI_HBA_CANSLEEP (memory allocation may sleep).	<i>hba_tran</i>	Pointer to a scsi_hba_tran(9S) structure.
<i>dip</i>	Pointer to a dev_info structure, defining the HBA driver instance.						
<i>flag</i>	Flag modifiers. The only possible flag value is SCSI_HBA_CANSLEEP (memory allocation may sleep).						
<i>hba_tran</i>	Pointer to a scsi_hba_tran(9S) structure.						
<b>DESCRIPTION</b>	<p>For scsi_hba_tran_alloc():</p> <p>scsi_hba_tran_alloc() allocates a scsi_hba_tran(9S) structure for a HBA driver. The HBA must use this structure to register its transport vectors with the system by using scsi_hba_attach_setup(9F).</p> <p>If the flag SCSI_HBA_CANSLEEP is set in <i>flags</i>, scsi_hba_tran_alloc() may sleep when allocating resources; otherwise it may not sleep, and callers should be prepared to deal with allocation failures.</p> <p>For scsi_hba_tran_free():</p> <p>scsi_hba_tran_free() is used to free the scsi_hba_tran(9S) structure allocated by scsi_hba_tran_alloc().</p>						
<b>RETURN VALUES</b>	scsi_hba_tran_alloc() returns a pointer to the allocated transport structure, or NULL if no space is available.						
<b>CONTEXT</b>	scsi_hba_tran_alloc() can be called from user or interrupt context. Drivers must not allow scsi_hba_tran_alloc() to sleep if called from an interrupt routine.	scsi_hba_tran_free() can be called from user or interrupt context.					
<b>SEE ALSO</b>	scsi_hba_attach_setup(9F), scsi_hba_tran(9S)	<i>Writing Device Drivers</i>					

<b>NAME</b>	scsi_ifgetcap, scsi_ifsetcap – get/set SCSI transport capability																		
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_ifgetcap</b>(struct scsi_address *ap, char *cap, int whom);  int <b>scsi_ifsetcap</b>(struct scsi_address *ap, char *cap, int value, int     whom);</pre>																		
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).																		
<b>PARAMETERS</b>	<p><i>ap</i> Pointer to the <code>scsi_address</code> structure.</p> <p><i>cap</i> Pointer to the string capability identifier.</p> <p><i>value</i> Defines the new state of the capability.</p> <p><i>whom</i> Determines if all targets or only the specified target is affected.</p>																		
<b>DESCRIPTION</b>	<p>The target drivers use <code>scsi_ifsetcap()</code> to set the capabilities of the host adapter driver. A <i>cap</i> is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using <code>scsi_ifgetcap()</code>. If <i>whom</i> is 0 all targets are affected, else the target specified by the <code>scsi_address</code> structure pointed to by <i>ap</i> is affected.</p> <p>A device may support only a subset of the capabilities listed below. It is the responsibility of the driver to make sure that these functions are called with a <i>cap</i> supported by the device.</p> <p>The following capabilities have been defined:</p> <table border="0"> <tr> <td style="padding-right: 20px;"><code>dma-max</code></td> <td>Maximum dma transfer size supported by host adapter.</td> </tr> <tr> <td><code>msg-out</code></td> <td>Message out capability supported by host adapter: 0 disables, 1 enables.</td> </tr> <tr> <td><code>disconnect</code></td> <td>Disconnect capability supported by host adapter: 0 disables, 1 enables.</td> </tr> <tr> <td><code>synchronous</code></td> <td>Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.</td> </tr> <tr> <td><code>wide-xfer</code></td> <td>Wide transfer capability supported by host adapter: 0 disables, 1 enables.</td> </tr> <tr> <td><code>parity</code></td> <td>Parity checking by host adapter: 0 disables, 1 enables.</td> </tr> <tr> <td><code>initiator-id</code></td> <td>The host's bus address is returned.</td> </tr> <tr> <td><code>untagged-qing</code></td> <td>The host adapter's capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.</td> </tr> <tr> <td><code>tagged-qing</code></td> <td>The host adapter's capability to support tagged queueing: 0 disables, 1 enables.</td> </tr> </table>	<code>dma-max</code>	Maximum dma transfer size supported by host adapter.	<code>msg-out</code>	Message out capability supported by host adapter: 0 disables, 1 enables.	<code>disconnect</code>	Disconnect capability supported by host adapter: 0 disables, 1 enables.	<code>synchronous</code>	Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.	<code>wide-xfer</code>	Wide transfer capability supported by host adapter: 0 disables, 1 enables.	<code>parity</code>	Parity checking by host adapter: 0 disables, 1 enables.	<code>initiator-id</code>	The host's bus address is returned.	<code>untagged-qing</code>	The host adapter's capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.	<code>tagged-qing</code>	The host adapter's capability to support tagged queueing: 0 disables, 1 enables.
<code>dma-max</code>	Maximum dma transfer size supported by host adapter.																		
<code>msg-out</code>	Message out capability supported by host adapter: 0 disables, 1 enables.																		
<code>disconnect</code>	Disconnect capability supported by host adapter: 0 disables, 1 enables.																		
<code>synchronous</code>	Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.																		
<code>wide-xfer</code>	Wide transfer capability supported by host adapter: 0 disables, 1 enables.																		
<code>parity</code>	Parity checking by host adapter: 0 disables, 1 enables.																		
<code>initiator-id</code>	The host's bus address is returned.																		
<code>untagged-qing</code>	The host adapter's capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.																		
<code>tagged-qing</code>	The host adapter's capability to support tagged queueing: 0 disables, 1 enables.																		

## scsi\_ifgetcap(9F)

<code>auto-rqsense</code>	The host adapter's capability to support auto request sense on check conditions: 0 disables, 1 enables.
<code>sector-size</code>	The target driver sets this capability to inform the HBA of the granularity, in bytes, of DMA breakup; the HBA's DMA limit structure will be set to reflect this limit (see <code>ddi_dma_lim_sparc(9S)</code> or <code>ddi_dma_lim_x86(9S)</code> ). It should be set to the physical disk sector size. This capability defaults to 512.
<code>total-sectors</code>	The target driver sets this capability to inform the HBA of the total number of sectors on the device, as returned from the SCSI get capacity command. This capability must be set before the target driver "gets" the geometry capability.
<code>geometry</code>	<p>This capability returns the HBA geometry of a target disk. The target driver must set the <code>total-sectors</code> capability before "getting" the <code>geometry</code> 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 <code>geometry</code> capability cannot be "set."</p> <p>If <code>geometry</code> is not relevant or appropriate for this target disk, because (for example) the HBA BIOS supports Logical Block Addressing for this drive, it is acceptable for <code>scsi_ifgetcap()</code> to return -1, indicating that the geometry is not defined. This will cause failure of attempts to retrieve the "virtual geometry" from the target driver (the <code>DKIOCG_VIRTGEOM</code> ioctl will fail). See <code>dkio(7I)</code> for more information about <code>DKIOCG_VIRTGEOM</code>.</p>
<code>reset-notification</code>	The host adapter's capability to support bus reset notification: 0 disables, 1 enables. Refer to <code>scsi_reset_notify(9F)</code> .
<code>linked -cmds</code>	The host adapter's capability to support linked commands: 0 disables, 1 enables.
<code>qfull-retries</code>	This capability enables/disables <code>QUEUE FULL</code> handling. If 0, the HBA will not retry a command when a <code>QUEUE FULL</code> status is returned. If greater than 0, then the HBA driver will retry the command at specified number of times at an interval determined by the " <code>qfull-retry-interval</code> ". The range for <code>qfull-retries</code> is 0-255.

`qfull-retry-interval` This capability sets the retry interval (in ms) for commands that were completed with a `QUEUE FULL` status. The range for `qfull-retry-intervals` is 0-1000 ms.

`lun-reset` This capability is created with a value of zero by HBA drivers that support the `RESET_LUN` flag in their `tran_reset(9E)` routine. If it exists, its value can be set to 1 by target drivers that want to allow use of LOGICAL UNIT RESET on a specific target instance. If `lun-reset` does not exist or has a value of zero, `scsi_reset(9F)` is prevented from passing the `RESET_LUN` flag to the HBA driver's `tran_reset()` routine. If `lun-reset` exists and has a value of 1, then the HBA driver's `tran_reset()` routine can be called with the `RESET_LUN` flag.

**RETURN VALUES**

`scsi_ifsetcap()` returns:

- 1 If the capability was successfully set to the new value.
- 0 If the capability is not variable.
- 1 If the capability was not defined, or setting the capability to a new value failed.

`scsi_ifgetcap()` returns the current value of a capability, or:

- 1 If the capability was not defined.

**CONTEXT**

These functions can be called from user or interrupt context.

**EXAMPLES**

**EXAMPLE 1** Using `scsi_ifgetcap()`

```
if (scsi_ifgetcap(&sd->sd_address, "auto-rqsense", 1) == 1) {
    un->un_arq_enabled = 1;
} else {
    un->un_arq_enabled =
        ((scsi_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;
}
```

scsi\_ifgetcap(9F)

**EXAMPLE 1** Using `scsi_ifgetcap()`     *(Continued)*

**SEE ALSO** `tran_reset(9E)`, `scsi_hba_lookup_capstr(9F)`, `scsi_reset(9F)`,  
`scsi_reset_notify(9F)`, `ddi_dma_lim_sparc(9S)`, `ddi_dma_lim_x86(9S)`,  
`scsi_address(9S)`, `scsi_arq_status(9S)`

*Writing Device Drivers*

<b>NAME</b>	scsi_init_pkt – prepare a complete SCSI packet
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  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);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>ap</i> Pointer to a scsi_address(9S) structure.</p> <p><i>pktp</i> A pointer to a scsi_pkt(9S) structure.</p> <p><i>bp</i> Pointer to a buf(9S) structure.</p> <p><i>cmdlen</i> The required length for the SCSI command descriptor block (CDB) in bytes.</p> <p><i>statuslen</i> The required length for the SCSI status completion block (SCB) in bytes. Valid values are:</p> <ul style="list-style-type: none"> <li>0 No status back.</li> <li>1 Return SCSI status byte.</li> <li>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.</li> <li>EXTCMDS_STATUS_SIZE Same as preceding.</li> </ul> <p><i>privatelen</i> The required length for the <i>pkt_private</i> area.</p> <p><i>flags</i> Flags modifier.</p> <p><i>callback</i> A pointer to a callback function, NULL_FUNC, or SLEEP_FUNC.</p> <p><i>arg</i> The <i>callback</i> function argument.</p>
<b>DESCRIPTION</b>	<p>Target drivers use <code>scsi_init_pkt()</code> to request the transport layer to allocate and initialize a packet for a SCSI command which possibly includes a data transfer. If <i>pktp</i> is NULL, a new <code>scsi_pkt(9S)</code> is allocated using the HBA driver's packet allocator. The <i>bp</i> is a pointer to a <code>buf(9S)</code> structure. If <i>bp</i> is non-NULL and contains a valid byte count, the <code>buf(9S)</code> structure is also set up for DMA transfer using the HBA driver DMA resources allocator. When <i>bp</i> is allocated by <code>scsi_alloc_consistent_buf(9F)</code>, the <code>PKT_CONSISTENT</code> bit must be set in the <i>flags</i> argument to ensure proper operation. If <i>privatelen</i> is non-zero then additional</p>

## scsi\_init\_pkt(9F)

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 *dmr\_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	<i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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** `scsi_init_pkt()` returns `NULL` if the packet or DMA resources could not be allocated. Otherwise, it returns a pointer to an initialized `scsi_pkt(9S)`. If `pktp` was not `NULL` the return value will be `pktp` on successful initialization of the packet.

**CONTEXT** If `callback` is `SLEEP_FUNC`, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block.

**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);
```

scsi\_init\_pkt(9F)

**EXAMPLE 5** Allocating a Packet with Partial DMA Resources Attached (Continued)

**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`.

<b>NAME</b>	scsi_log – display a SCSI-device-related message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt; #include &lt;sys/cmn_err.h&gt;  void <b>scsi_log</b>(dev_info_t *dip, char *drv_name, uint_t level, const char *fmt, ...);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>dip</i>                    Pointer to the dev_info structure.</p> <p><i>drv_name</i>               String naming the device.</p> <p><i>level</i>                   Error level.</p> <p><i>fmt</i>                     Display format.</p>
<b>DESCRIPTION</b>	<p>scsi_log() is a utility function that displays a message via the cmn_err(9F) routine. The error levels that can be passed in to this function are CE_PANIC, CE_WARN, CE_NOTE, CE_CONT, and SCSI_DEBUG. The last level is used to assist in displaying debug messages to the console only. <i>drv_name</i> is the short name by which this device is known; example disk driver names are sd and cmdk. If the <i>dev_info_t</i> pointer is NULL, then the <i>drv_name</i> will be used with no unit or long name.</p> <p>If the first character in <i>format</i> is:</p> <ul style="list-style-type: none"> <li>■ An exclamation mark (!), the message goes only to the system buffer.</li> <li>■ A caret (^), the message goes only to the console.</li> <li>■ A question mark (?) and <i>level</i> 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.</li> </ul> <p>All formatting conversions in use by cmn_err() also work with scsi_log().</p>
<b>CONTEXT</b>	scsi_log() may be called from user or interrupt context.
<b>EXAMPLES</b>	<pre>scsi_log(dev, "Disk Unit ", CE_PANIC, "Bad Value %d\n", foo); generates:  PANIC: /eisa/aha@330,0/cmdk@0,0 (Disk Unit 0): Bad Value 5  This is followed by a PANIC.  scsi_log(dev, "sd", CE_WARN, "Label Bad\n"); generates:  WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1): Label Bad  scsi_log((dev_info_t *) NULL, "Disk Unit ", CE_NOTE, "Disk Ejected\n"); generates:  Disk Unit: Disk Ejected</pre>

scsi\_log(9F)

```
scsi_log(cmdk_unit, "Disk Unit ", CE_CONT, "Disk Inserted\n");  
generates:
```

```
Disk Inserted
```

```
scsi_log(sd_unit, "sd", SCSI_DEBUG, "We really got here\n");  
generates (only to the console):
```

```
DEBUG: sd1: We really got here
```

**SEE ALSO** kernel(1M), sd(7D), cmn\_err(9F), scsi\_errmsg(9F)

*Writing Device Drivers*

<b>NAME</b>	scsi_pktalloc, scsi_realloc, scsi_pktfree, scsi_resfree – SCSI packet utility routines
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  struct scsi_pkt *scsi_pktalloc(struct scsi_address*ap, intcmdlen,                                intstatuslen, int (*callback) (void));  struct scsi_pkt *scsi_realloc(struct scsi_address*ap, intcmdlen,                               intstatuslen, opaque_tdmatoken, int (*callback) (void));  voidscsi_pktfree(struct scsi_pkt*pkt);  voidscsi_resfree(struct scsi_pkt*pkt);</pre>
<b>INTERFACE LEVEL</b>	The <code>scsi_pktalloc()</code> , <code>scsi_pktfree()</code> , <code>scsi_realloc()</code> , and <code>scsi_resfree()</code> functions are obsolete. The <code>scsi_pktalloc()</code> and <code>scsi_realloc()</code> functions have been replaced by <code>scsi_init_pkt(9F)</code> . The <code>scsi_pktfree()</code> and <code>scsi_resfree()</code> functions have been replaced by <code>scsi_destroy_pkt(9F)</code> .
<b>PARAMETERS</b>	<p><i>ap</i> Pointer to a <code>scsi_address</code> structure.</p> <p><i>cmdlen</i> The required length for the SCSI command descriptor block (CDB) in bytes.</p> <p><i>statuslen</i> The required length for the SCSI status completion block (SCB) in bytes.</p> <p><i>dmatoken</i> Pointer to an implementation-dependent object.</p> <p><i>callback</i> A pointer to a callback function, or <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code>.</p> <p><i>pkt</i> Pointer to a <code>scsi_pkt(9S)</code> structure.</p>
<b>DESCRIPTION</b>	<p><code>scsi_pktalloc()</code> requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, <code>scsi_realloc()</code> should be used.</p> <p><i>ap</i> is a pointer to a <code>scsi_address</code> structure. Allocator routines use it to determine the associated host adapter.</p> <p><i>cmdlen</i> is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the <code>pkt_cdbp</code> field of the allocated <code>scsi_pkt</code> structure.</p> <p><i>statuslen</i> is the required length for the SCSI status completion block. The address of the allocated block is placed into the <code>pkt_scbp</code> field of the <code>scsi_pkt</code> structure.</p> <p><i>dmatoken</i> 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 <i>dmatoken</i> must be a pointer to a <code>buf(9S)</code> structure. If <i>dmatoken</i> is <code>NULL</code>, 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</p>

## scsi\_pktalloc(9F)

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_DEERR`. `dmatoken` provides the information to determine if the transfer count is correct.

`callback` indicates what the allocator routines should do when resources are not available:

`NULL_FUNC` Do not wait for resources. Return a `NULL` pointer.

`SLEEP_FUNC` Wait indefinitely for resources.

**Other Values** `callback` points to a function which is called when resources may have become available. `callback` must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

`scsi_pktfree()` frees the packet.

`scsi_resfree()` free all resources held by the packet and the packet itself.

**RETURN VALUES** Both allocation routines return a pointer to a `scsi_pkt` structure on success, or `NULL` on failure.

**CONTEXT** If `callback` is `SLEEP_FUNC`, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

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

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_realloc()`, and `scsi_resfree()` functions are obsolete and will be discontinued in a future release. The `scsi_pktalloc()` and `scsi_realloc()` functions have been replaced by `scsi_init_pkt(9F)`. The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by `scsi_destroy_pkt(9F)`.

<b>NAME</b>	scsi_poll – run a polled SCSI command on behalf of a target driver
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_poll</b>(struct scsi_pkt *pkt);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>pkt</i> Pointer to the scsi_pkt(9S) structure.
<b>DESCRIPTION</b>	scsi_poll() requests the host adapter driver to run a polled command. Unlike scsi_transport(9F) which runs commands asynchronously, scsi_poll() runs commands to completion before returning. If the pkt_time member of <i>pkt</i> is 0, the value of pkt_time is defaulted to SCSI_POLL_TIMEOUT to prevent an indefinite hang of the system.
<b>RETURN VALUES</b>	scsi_poll() returns: 0 command completed successfully. -1 command failed.
<b>CONTEXT</b>	scsi_poll() can be called from user or interrupt level. This function should not be called when the caller is executing timeout(9F) in the context of a thread.
<b>SEE ALSO</b>	makecom(9F), scsi_transport(9F), scsi_pkt(9S)  <i>Writing Device Drivers</i>
<b>WARNINGS</b>	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.

## scsi\_probe(9F)

<b>NAME</b>	scsi_probe – utility for probing a scsi device														
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_probe</b>(struct scsi_device *devp, int (*waitfunc) );</pre>														
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>devp</i> Pointer to a scsi_device(9S) structure <i>waitfunc</i> NULL_FUNC or SLEEP_FUNC														
<b>DESCRIPTION</b>	<p>scsi_probe() determines whether a target/lun is present and sets up the scsi_device structure with inquiry data.</p> <p>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 SCSI_PROBE_EXISTS. If scsi_probe() is unsuccessful, it returns SCSI_PROBE_NOMEM in spite of callback set to SLEEP_FUNC.</p> <p>scsi_unprobe(9F) is used to undo the effect of scsi_probe().</p> <p>If the target is a non-CCS device, SCSI_PROBE_NONCCS will be returned.</p> <p><i>waitfunc</i> indicates what the allocator routines should do when resources are not available; the valid values are:</p> <table><tr><td>NULL_FUNC</td><td>Do not wait for resources. Return SCSI_PROBE_NOMEM or SCSI_PROBE_FAILURE</td></tr><tr><td>SLEEP_FUNC</td><td>Wait indefinitely for resources.</td></tr></table>	NULL_FUNC	Do not wait for resources. Return SCSI_PROBE_NOMEM or SCSI_PROBE_FAILURE	SLEEP_FUNC	Wait indefinitely for resources.										
NULL_FUNC	Do not wait for resources. Return SCSI_PROBE_NOMEM or SCSI_PROBE_FAILURE														
SLEEP_FUNC	Wait indefinitely for resources.														
<b>RETURN VALUES</b>	scsi_probe() returns: <table><tr><td>SCSI_PROBE_BUSY</td><td>Device exists but is currently busy.</td></tr><tr><td>SCSI_PROBE_EXISTS</td><td>Device exists and inquiry data is valid.</td></tr><tr><td>SCSI_PROBE_FAILURE</td><td>Polled command failure.</td></tr><tr><td>SCSI_PROBE_NOMEM</td><td>No space available for structures.</td></tr><tr><td>SCSI_PROBE_NOMEM_CB</td><td>No space available for structures but callback request has been queued.</td></tr><tr><td>SCSI_PROBE_NONCCS</td><td>Device exists but inquiry data is not valid.</td></tr><tr><td>SCSI_PROBE_NORESP</td><td>Device does not respond to an INQUIRY.</td></tr></table>	SCSI_PROBE_BUSY	Device exists but is currently busy.	SCSI_PROBE_EXISTS	Device exists and inquiry data is valid.	SCSI_PROBE_FAILURE	Polled command failure.	SCSI_PROBE_NOMEM	No space available for structures.	SCSI_PROBE_NOMEM_CB	No space available for structures but callback request has been queued.	SCSI_PROBE_NONCCS	Device exists but inquiry data is not valid.	SCSI_PROBE_NORESP	Device does not respond to an INQUIRY.
SCSI_PROBE_BUSY	Device exists but is currently busy.														
SCSI_PROBE_EXISTS	Device exists and inquiry data is valid.														
SCSI_PROBE_FAILURE	Polled command failure.														
SCSI_PROBE_NOMEM	No space available for structures.														
SCSI_PROBE_NOMEM_CB	No space available for structures but callback request has been queued.														
SCSI_PROBE_NONCCS	Device exists but inquiry data is not valid.														
SCSI_PROBE_NORESP	Device does not respond to an INQUIRY.														
<b>CONTEXT</b>	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 SCSI_PROBE_NORESP:
case SCSI_PROBE_NONCCS:
case SCSI_PROBE_NOMEM:
case SCSI_PROBE_FAILURE:
case SCSI_PROBE_BUSY:
    break;
case SCSI_PROBE_EXISTS:
    switch (devp->sd_inq->inq_dtype) {
    case DTYPE_DIRECT:
        rval = DDI_PROBE_SUCCESS;
        break;
    case DTYPE_RODIRECT:
        rval = DDI_PROBE_SUCCESS;
        break;
    case DTYPE_NOTPRESENT:
    default:
        break;
    }
}
scsi_unprobe(devp);

```

**SEE ALSO**

`attach(9E)`, `probe(9E)`, `scsi_slave(9F)`, `scsi_unprobe(9F)`, `scsi_unslave(9F)`, `scsi_device(9S)`, `scsi_inquiry(9S)`

*ANSI Small Computer System Interface-2 (SCSI-2)*

*Writing Device Drivers*

**NOTES**

A *waitfunc* function other than `NULL_FUNC` or `SLEEP_FUNC` is not supported and may have unexpected results.

## scsi\_reset(9F)

<b>NAME</b>	<code>scsi_reset</code> – reset a SCSI bus or target
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_reset</b>(struct scsi_address *ap, int level);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>ap</i> Pointer to the <code>scsi_address</code> structure. <i>level</i> The level of reset required.
<b>DESCRIPTION</b>	<p><code>scsi_reset()</code> asks the host adapter driver to reset the SCSI bus or a SCSI target as specified by <i>level</i>. If <i>level</i> equals <code>RESET_ALL</code>, the SCSI bus is reset. If it equals <code>RESET_TARGET</code>, <i>ap</i> is used to determine the target to be reset. If it equals <code>RESET_LUN</code>, <i>ap</i> is used to determine the logical unit to be reset.</p> <p>When given the <code>RESET_LUN</code> level, <code>scsi_reset()</code> 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.</p> <p>Note that, at the point when <code>scsi_reset()</code> resets the logical unit (case <code>RESET_LUN</code>), or the target (case <code>RESET_TARGET</code>), or the bus (case <code>RESET_ALL</code>), there might be one or more command packets outstanding. That is, packets have been passed to <code>scsi_transport()</code>, and queued or possibly transported, but the commands have not been completed and the target completion routine has not been called for those packets.</p> <p>The successful call to <code>scsi_reset()</code> has the side effect that any such commands currently outstanding are aborted, at which point the packets are marked with <code>pkt_reason</code> set to <code>CMD_RESET</code>, and the appropriate bit -- either <code>STAT_BUS_RESET</code> or <code>STAT_DEV_RESET</code> -- is set in <code>pkt_statistics</code>. Once thus appropriately marked, the aborted command packets are passed to the target driver command completion routine.</p> <p>Also note that, at the moment that a thread executing <code>scsi_reset()</code> actually resets the target or the bus, it is possible that a second thread may have already called <code>scsi_transport()</code>, 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.</p>
<b>RETURN VALUES</b>	<code>scsi_reset()</code> returns: 1 Upon success. 0 Upon failure.
<b>CONTEXT</b>	<code>scsi_reset()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>tran_reset(9E)</code> , <code>tran_reset_notify(9E)</code> , <code>scsi_abort(9F)</code>  <i>Writing Device Drivers</i>

<b>NAME</b>	scsi_reset_notify – notify target driver of bus resets								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_reset_notify</b>(struct scsi_address *ap, int flag, void     (*callback) (caddr_t), caddr_t arg);</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>ap</i> Pointer to the <code>scsi_address</code> structure.</p> <p><i>flag</i> A flag indicating registration or cancellation of the notification request.</p> <p><i>callback</i> A pointer to the target driver's reset notification function.</p> <p><i>arg</i> The callback function argument.</p>								
<b>DESCRIPTION</b>	<p><code>scsi_reset_notify()</code> 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.</p> <p>The argument <i>flag</i> is used to register or cancel the notification. The supported values for <i>flag</i> are as follows:</p> <table border="0"> <tr> <td>SCSI_RESET_NOTIFY</td> <td>Register <i>callback</i> as the reset notification function for the target driver.</td> </tr> <tr> <td>SCSI_RESET_CANCEL</td> <td>Cancel the reset notification request.</td> </tr> </table> <p>Target drivers can find out whether the HBA driver and controller support reset notification by checking the <code>reset-notification</code> capability using the <code>scsi_ifgetcap(9F)</code> function.</p>	SCSI_RESET_NOTIFY	Register <i>callback</i> as the reset notification function for the target driver.	SCSI_RESET_CANCEL	Cancel the reset notification request.				
SCSI_RESET_NOTIFY	Register <i>callback</i> as the reset notification function for the target driver.								
SCSI_RESET_CANCEL	Cancel the reset notification request.								
<b>RETURN VALUES</b>	<p>If <i>flag</i> is <code>SCSI_RESET_NOTIFY</code>, <code>scsi_reset_notify()</code> returns:</p> <table border="0"> <tr> <td>DDI_SUCCESS</td> <td>The notification request has been accepted.</td> </tr> <tr> <td>DDI_FAILURE</td> <td>The transport layer does not support reset notification or could not accept this request.</td> </tr> </table> <p>If <i>flag</i> is <code>SCSI_RESET_CANCEL</code>, <code>scsi_reset_notify()</code> returns:</p> <table border="0"> <tr> <td>DDI_SUCCESS</td> <td>The notification request has been canceled.</td> </tr> <tr> <td>DDI_FAILURE</td> <td>No notification request was registered.</td> </tr> </table>	DDI_SUCCESS	The notification request has been accepted.	DDI_FAILURE	The transport layer does not support reset notification or could not accept this request.	DDI_SUCCESS	The notification request has been canceled.	DDI_FAILURE	No notification request was registered.
DDI_SUCCESS	The notification request has been accepted.								
DDI_FAILURE	The transport layer does not support reset notification or could not accept this request.								
DDI_SUCCESS	The notification request has been canceled.								
DDI_FAILURE	No notification request was registered.								
<b>CONTEXT</b>	<code>scsi_reset_notify()</code> can be called from user or interrupt context.								
<b>SEE ALSO</b>	<p><code>scsi_address(9S)</code>, <code>scsi_ifgetcap(9F)</code></p> <p><i>Writing Device Drivers</i></p>								

## scsi\_setup\_cdb(9F)

<b>NAME</b>	scsi_setup_cdb – setup SCSI command descriptor block (CDB)										
<b>SYNOPSIS</b>	<pre>int <b>scsi_setup_cdb</b>(union scsi_cdb *cdbp, uchar_t cmd, uint_t addr,     uint_t cnt, uint_t othr_cdb_data);</pre>										
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).										
<b>PARAMETERS</b>	<table><tr><td><i>cdbp</i></td><td>Pointer to command descriptor block.</td></tr><tr><td><i>cmd</i></td><td>The first byte of the SCSI group 0, 1, 2, 4, or 5 CDB.</td></tr><tr><td><i>addr</i></td><td>Pointer to the location of the data.</td></tr><tr><td><i>cnt</i></td><td>Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.</td></tr><tr><td><i>othr_cdb_data</i></td><td>Additional CDB data.</td></tr></table>	<i>cdbp</i>	Pointer to command descriptor block.	<i>cmd</i>	The first byte of the SCSI group 0, 1, 2, 4, or 5 CDB.	<i>addr</i>	Pointer to the location of the data.	<i>cnt</i>	Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.	<i>othr_cdb_data</i>	Additional CDB data.
<i>cdbp</i>	Pointer to command descriptor block.										
<i>cmd</i>	The first byte of the SCSI group 0, 1, 2, 4, or 5 CDB.										
<i>addr</i>	Pointer to the location of the data.										
<i>cnt</i>	Data transfer length in units defined by the SCSI device type. For sequential devices <i>cnt</i> is the number of bytes. For block devices, <i>cnt</i> is the number of blocks.										
<i>othr_cdb_data</i>	Additional CDB data.										
<b>DESCRIPTION</b>	<p><code>scsi_setup_cdb()</code> function initializes a group 0, 1, 2, 4, or 5 type of command descriptor block pointed to by <i>cdbp</i> using <i>cmd</i>, <i>addr</i>, <i>cnt</i>, <i>othr_cdb_data</i>.</p> <p><i>addr</i> should be set to 0 for commands having no addressing information (for example, group 0 READ command for sequential access devices). <i>othr_cdb_data</i> should be additional CDB data for Group 4 commands; otherwise, it should be set to 0.</p> <p><code>scsi_setup_cdb()</code> function does not set the LUN bits in CDB[1] as the <code>makecom(9F)</code> functions do. Also, the fixed bit for sequential access device commands is not set.</p>										
<b>RETURN VALUES</b>	<p><code>scsi_setup_cdb()</code> returns:</p> <table><tr><td>1</td><td>Upon success.</td></tr><tr><td>0</td><td>Upon failure.</td></tr></table>	1	Upon success.	0	Upon failure.						
1	Upon success.										
0	Upon failure.										
<b>CONTEXT</b>	These functions can be called from a user or interrupt context.										
<b>SEE ALSO</b>	<p><code>makecom(9F)</code>, <code>scsi_pkt(9S)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>American National Standard Small Computer System Interface-2 (SCSI-2)</i></p> <p><i>American National Standard SCSI-3 Primary Commands (SPC)</i></p>										

<b>NAME</b>	scsi_slave – utility for SCSI target drivers to establish the presence of a target
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_slave</b>(struct scsi_device *devp, int (*callback)(void));</pre>
<b>INTERFACE LEVEL</b>	The <code>scsi_slave()</code> function is obsolete. This function has been replaced by <code>scsi_probe(9F)</code> .
<b>PARAMETERS</b>	<p><i>devp</i>                    Pointer to a <code>scsi_device(9S)</code> structure.</p> <p><i>callback</i>                Pointer to a callback function, <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code>.</p>
<b>DESCRIPTION</b>	<p><code>scsi_slave()</code> checks for the presence of a SCSI device. Target drivers may use this function in their <code>probe(9E)</code> routines. <code>scsi_slave()</code> determines if the device is present by using a Test Unit Ready command followed by an Inquiry command. If <code>scsi_slave()</code> is successful, it will fill in the <code>scsi_inquiry</code> structure, which is the <code>sd_inq</code> member of the <code>scsi_device(9S)</code> structure, and return <code>SCSI_PROBE_EXISTS</code>. This information can be used to determine if the target driver has probed the correct SCSI device type. <i>callback</i> indicates what the allocator routines should do when DMA resources are not available:</p> <p><code>NULL_FUNC</code>                Do not wait for resources. Return a <code>NULL</code> pointer.</p> <p><code>SLEEP_FUNC</code>                Wait indefinitely for resources.</p> <p>Other Values                <i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> 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.</p>
<b>RETURN VALUES</b>	<p><code>scsi_slave()</code> returns:</p> <p><code>SCSI_PROBE_NOMEM</code>                No space available for structures.</p> <p><code>SCSI_PROBE_EXISTS</code>                Device exists and inquiry data is valid.</p> <p><code>SCSI_PROBE_NONCCS</code>                Device exists but inquiry data is not valid.</p> <p><code>SCSI_PROBE_FAILURE</code>                Polled command failure.</p> <p><code>SCSI_PROBE_NORESP</code>                No response to TEST UNIT READY.</p>
<b>CONTEXT</b>	<code>scsi_slave()</code> is normally called from the target driver's <code>probe(9E)</code> or <code>attach(9E)</code> routine. In any case, this routine should not be called from interrupt context, because it can sleep waiting for memory to be allocated.
<b>ATTRIBUTES</b>	See <code>attributes(5)</code> for a description of the following attributes:

scsi\_slave(9F)

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)`.

<b>NAME</b>	scsi_sync_pkt – synchronize CPU and I/O views of memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_sync_pkt</b>(struct scsi_pkt *pktp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Solaris DDI specific (Solaris DDI). <i>pktp</i> Pointer to a scsi_pkt(9S) structure.
<b>DESCRIPTION</b>	<p>scsi_sync_pkt () is used to selectively synchronize a CPU's or device's view of the data associated with the SCSI packet that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.</p> <p>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.</p> <p>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.</p>
<b>CONTEXT</b>	scsi_sync_pkt () may be called from user or interrupt context.
<b>SEE ALSO</b>	<p>tran_sync_pkt(9E), ddi_dma_sync(9F), scsi_destroy_pkt(9F), scsi_init_pkt(9F), scsi_transport(9F), scsi_pkt(9S)</p> <p><i>Writing Device Drivers</i></p>

## scsi\_transport(9F)

<b>NAME</b>	scsi_transport – request by a SCSI target driver to start a command
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  int <b>scsi_transport</b>(struct scsi_pkt *pkt);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>pkt</i> Pointer to a scsi_pkt(9S) structure.
<b>DESCRIPTION</b>	Target drivers use <code>scsi_transport()</code> to request the host adapter driver to transport a command to the SCSI target device specified by <i>pkt</i> . The target driver must obtain resources for the packet using <code>scsi_init_pkt(9F)</code> prior to calling this function. The packet may be initialized using one of the <code>makecom(9F)</code> functions. <code>scsi_transport()</code> does not wait for the SCSI command to complete. See <code>scsi_poll(9F)</code> 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 <code>pkt_comp</code> member of the <code>scsi_pkt</code> pointed to by <i>pkt</i> .
<b>RETURN VALUES</b>	<code>scsi_transport()</code> 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.
<b>CONTEXT</b>	<code>scsi_transport()</code> can be called from user or interrupt context.
<b>EXAMPLES</b>	<b>EXAMPLE 1</b> Using <code>scsi_transport()</code> <pre>if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {     scsi_log(devp, sd_label, CE_WARN,             "transport of request sense pkt fails (0x%x)\n", status); }</pre>
<b>SEE ALSO</b>	<code>tran_start(9E)</code> , <code>makecom(9F)</code> , <code>scsi_init_pkt(9F)</code> , <code>scsi_pktalloc(9F)</code> , <code>scsi_poll(9F)</code> , <code>scsi_pkt(9S)</code>  <i>Writing Device Drivers</i>

<b>NAME</b>	scsi_unprobe, scsi_unslave – free resources allocated during initial probing
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_unslave</b>(struct scsi_device *devp); void <b>scsi_unprobe</b>(struct scsi_device *devp);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI). The <code>scsi_unslave()</code> interface is obsolete. Use <code>scsi_unprobe()</code> instead.
<b>PARAMETERS</b>	<i>devp</i> Pointer to a <code>scsi_device(9S)</code> structure.
<b>DESCRIPTION</b>	<code>scsi_unprobe()</code> and <code>scsi_unslave()</code> are used to free any resources that were allocated on the driver's behalf during <code>scsi_slave(9F)</code> and <code>scsi_probe(9F)</code> activity.
<b>CONTEXT</b>	<code>scsi_unprobe()</code> and <code>scsi_unslave()</code> must not be called from an interrupt context.
<b>SEE ALSO</b>	<code>scsi_probe(9F)</code> , <code>scsi_slave(9F)</code> , <code>scsi_device(9S)</code> <i>Writing Device Drivers</i>
<b>NOTES</b>	The <code>scsi_unslave()</code> function is obsolete and will be discontinued in a future release. This function has been replaced by <code>scsi_unprobe()</code> .

## scsi\_vu\_errmsg(9F)

<b>NAME</b>	scsi_vu_errmsg – display a SCSI request sense message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  void <b>scsi_vu_errmsg</b>(struct scsi_pkt *pkt, 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_frustruct     scsi_device*, char *, int, char);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	The following parameters are supported:  <i>devp</i> Pointer to the <code>scsi_device(9S)</code> structure.  <i>pkt</i> Pointer to a <code>scsi_pkt(9S)</code> structure.  <i>drv_name</i> String used by <code>scsi_log(9F)</code> .  <i>severity</i> Error severity level, maps to severity strings below.  <i>blkno</i> Requested block number.  <i>err_blkno</i> Error block number.  <i>cmdlist</i> An array of SCSI command description strings.  <i>sensep</i> A pointer to a <code>scsi_extended_sense(9S)</code> structure.  <i>asc_list</i> A pointer to a array of asc and ascq message list. The list must be terminated with -1 asc value.  <i>decode_fru</i> This is a function pointer that will be called after the entire sense information has been decoded. The parameters will be the <code>scsi_device</code> 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. <code>decode_fru</code> might be NULL if no special decoding is required. <code>decode_fru</code> is expected to return pointer to a char string if decoding possible and NULL if no decoding is possible.
<b>DESCRIPTION</b>	This function is very similar to <code>scsi_errmsg(9F)</code> but allows decoding of vendor-unique ASC/ASCQ and FRU information.

`scsi_vu_errmsg()` interprets the request sense information in the *sensep* pointer and generates a standard message that is displayed using `scsi_log(9F)`. It first searches the list array for a matching vendor unique code if supplied. If it does not find one in the list then the standard list is searched. The first line of the message is always a `CE_WARN`, with the continuation lines being `CE_CONT`. *sensep* may be `NULL`, in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The `scsi_device(9S)` structure denoted by *devp* supplies the identification of the device that requested the display. *severity* selects which string is used in the "Error Level:" reporting, according to the table below:

Severity Value:	String:
<code>SCSI_ERR_ALL</code>	All
<code>SCSI_ERR_UNKNOWN</code>	Unknown
<code>SCSI_ERR_INFO</code>	Information
<code>SCSI_ERR_RECOVERED</code>	Recovered
<code>SCSI_ERR_RETRYABLE</code>	Retryable
<code>SCSI_ERR_FATAL</code>	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",
```

## scsi\_vu\_errmsg(9F)

```
        0x37, "read defect data",
        0x3b, "write buffer",
        -1, NULL
    };
```

**CONTEXT** `scsi_vu_errmsg()` may be called from user or interrupt context.

**EXAMPLES** **EXAMPLE 1** Using `scsi_vu_errmsg()`

```
struct scsi_asc_key_strings cd_slist[] = {
    0x81, 0, "Logical Unit is inaccessible",
    -1, 0, NULL,
};

scsi_vu_errmsg(devp, pkt, "sd",
               SCSI_ERR_INFO, bp->b_blkno, err_blkno,
               sd_cmds, rqsense, cd_list,
               my_decode_fru);
```

This generates the following console warning:

```
WARNING: /sbus@1,f8000000/esp@0,800000/sd@1,0 (sd1):
Error for Command: read          Error Level: Informational
Requested Block: 23936          Error Block: 23936
Vendor: XYZ                     Serial Number: 123456
Sense Key: Unit Attention
ASC: 0x81 (Logical Unit is inaccessible), ASCQ: 0x0
FRU: 0x11 (replace LUN 1, located in slot 1)
```

**SEE ALSO** `cmn_err(9F)`, `scsi_errmsg(9F)`, `scsi_log(9F)`, `scsi_errmsg(9F)`,  
`scsi_asc_key_strings(9S)`, `scsi_device(9S)`, `scsi_extended_sense(9S)`,  
`scsi_pkt(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*

<b>NAME</b>	semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_try_p – semaphore functions
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ksynch.h&gt;  void <b>sema_init</b>(ksema_t *sp, uint_t val, char *name, ksema_type_t     type, void *arg);  void <b>sema_destroy</b>(ksema_t *sp);  void <b>sema_p</b>(ksema_t *sp);  void <b>sema_v</b>(ksema_t *sp);  int <b>sema_p_sig</b>(ksema_t *sp);  int <b>sema_try_p</b>(ksema_t *sp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Solaris DDI specific (Solaris DDI).</p> <p><i>sp</i>                    A pointer to a semaphore, type ksema_t.</p> <p><i>val</i>                    Initial value for semaphore.</p> <p><i>name</i>                   Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)</p> <p><i>type</i>                   Variant type of the semaphore. Currently, only SEMA_DRIVER is supported.</p> <p><i>arg</i>                    Type-specific argument; should be NULL.</p>
<b>DESCRIPTION</b>	<p>These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by <code>sema_p()</code> and atomically incremented by <code>sema_v()</code>. The value must always be greater than or equal to zero. If <code>sema_p()</code> is called and the value is zero, the calling thread is blocked until another thread performs a <code>sema_v()</code> operation on the semaphore.</p> <p>Semaphores are initialized by calling <code>sema_init()</code>. The argument, <code>val</code>, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by <code>sema_init()</code>. For this reason, <code>sema_destroy()</code> should be called before deallocating the storage containing the semaphore.</p> <p><code>sema_p_sig()</code> decrements the semaphore, as does <code>sema_p()</code>. However, if the semaphore value is zero, <code>sema_p_sig()</code> will return without decrementing the value if a signal (that is, from <code>kill(2)</code>) is pending for the thread.</p> <p><code>sema_try_p()</code> will decrement the semaphore value only if it is greater than zero, and will not block.</p>
<b>RETURN VALUES</b>	<p>0                    <code>sema_try_p()</code> could not decrement the semaphore value because it was zero.</p>

## semaphore(9F)

1        `sema_p_sig()` was not able to decrement the semaphore value and detected a pending signal.

**CONTEXT**    These functions can be called from user or interrupt context, except for `sema_init()` and `sema_destroy()`, which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, `sema_v()` and `sema_p()` should not be called from any interrupt context.

If `sema_p()` is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the `sema_v()` becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the `sema_v()` may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

**SEE ALSO**    `kill(2)`, `condvar(9F)`, `mutex(9F)`

*Writing Device Drivers*

<b>NAME</b>	sprintf, snprintf – format characters in memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  char *<b>sprintf</b>(char *buf, const char *fmt, ...);  size_t <b>snprintf</b>(char *buf, size_t n, const char *fmt, ...);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>buf</i>                    Pointer to a character string.</p> <p><i>fmt</i>                    Pointer to a character string.</p>
<b>DESCRIPTION</b>	<p><code>sprintf()</code> builds a string in <i>buf</i> under the control of the format <i>fmt</i>. The format is a character string with either plain characters, which are simply copied into <i>buf</i>, or conversion specifications, each of which converts zero or more arguments, again copied into <i>buf</i>. 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 <i>buf</i>.</p> <p>The <code>snprintf()</code> function is identical to <code>sprintf()</code> with the addition of the argument <i>n</i>, which specifies the size of the buffer referred to by <i>buf</i>. The buffer is always terminated with the null byte.</p>
<b>Conversion Specifications</b>	<p>Each conversion specification is introduced by the % character, after which the following appear in sequence:</p> <p>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.</p> <p>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.</p> <p>A character indicating the type of conversion to be applied:</p> <p>d,D,o,O,x,X,u</p> <p>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.</p> <p>c</p> <p>The character value of argument is copied.</p> <p>b</p> <p>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 &lt;base&gt; [&lt;arg&gt; . . . ]. The base supplies the conversion base for the first argument as a binary value; \10 gives</p>

## sprintf(9F)

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

`sprintf()` returns its first argument, *buf*.

`snprintf()` returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. If the value of *n* is less than or equal to 0 on a call to `snprintf()`, the function simply returns the number of characters formatted.

### CONTEXT

`sprintf()` and `snprintf()` can be called from user or interrupt context.

### SEE ALSO

*Writing Device Drivers*

<b>NAME</b>	stoi, numtos – convert between an integer and a decimal string
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; int stoi(char **str); void numtos(unsigned long num, char *s);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>str</i>        Pointer to a character string to be converted.</p> <p><i>num</i>        Decimal number to be converted to a character string.</p> <p><i>s</i>            Character buffer to hold converted decimal number.</p>
<b>DESCRIPTION</b>	
<b>stoi()</b>	stoi() returns the integer value of a string of decimal numeric characters beginning at <i>**str</i> . No overflow checking is done. <i>*str</i> is updated to point at the last character examined.
<b>numtos()</b>	numtos() converts a long into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.
<b>RETURN VALUES</b>	stoi() returns the integer value of the string <i>str</i> .
<b>CONTEXT</b>	stoi() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>
<b>NOTES</b>	stoi() handles only positive integers; it does not handle leading minus signs.

## strchr(9F)

<b>NAME</b>	strchr, strrchr – find a character in a string
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  char *<b>strchr</b>(const char *str, int chr); char *<b>strrchr</b>(const char *str, int chr);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>str</i>            Pointer to a string to be searched.</p> <p><i>chr</i>            The character to search for.</p>
<b>DESCRIPTION</b>	<p><b>strchr()</b> The <b>strchr()</b> function returns a pointer to the first occurrence of <i>chr</i> in the string pointed to by <i>str</i>.</p> <p><b>strrchr()</b> The <b>strrchr()</b> function returns a pointer to the last occurrence of <i>chr</i> in the string pointed to by <i>str</i>.</p>
<b>RETURN VALUES</b>	<b>strchr()</b> and <b>strrchr()</b> return a pointer to a character, or NULL, if the search fails.
<b>CONTEXT</b>	These functions can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>strcmp(9F)</p> <p><i>Writing Device Drivers</i></p>

<b>NAME</b>	strcmp, strcasecmp, strncasecmp, strncmp – compare two null-terminated strings.
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  int <b>strcmp</b>(const char *s1, const char *s2); int <b>strcasecmp</b>(const char *s1, const char *s2); int <b>strncasecmp</b>(const char *s1, const char *s2, size_t n); int <b>strncmp</b>(const char *s1, const char *s2, size_t n);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>s1, s2</i>                    Pointers to character strings.</p> <p><i>n</i>                            Count of characters to be compared.</p>
<b>DESCRIPTION</b>	
<b>strcmp()</b>	strcmp() returns 0 if the strings are the same, or the integer value of the expression (*s1 - *s2) for the last characters compared if they differ.
<b>strcasecmp(), strncasecmp()</b>	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.
<b>strncmp()</b>	strncmp() returns 0 if the first <i>n</i> characters of <i>s1</i> and <i>s2</i> are the same, or (*s1 - *s2) for the last characters compared if they differ.
<b>RETURN VALUES</b>	<p>strcmp() returns 0 if the strings are the same, or (*s1 - *s2) for the last characters compared if they differ.</p> <p>strcasecmp() and strncasecmp() return values in the same fashion as strcmp() and strncmp(), respectively.</p> <p>strncmp() returns 0 if the first <i>n</i> characters of strings are the same, or (*s1 - *s2) for the last characters compared if they differ.</p>
<b>CONTEXT</b>	These functions can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

## strcpy(9F)

<b>NAME</b>	strcpy, strncpy – copy a string from one location to another.				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  char *strcpy(char *dst, const char *srs); char *strncpy(char *dst, const char *srs, size_t n);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).				
<b>PARAMETERS</b>	<table><tr><td><i>dst</i>, <i>srs</i></td><td>Pointers to character strings.</td></tr><tr><td><i>n</i></td><td>Count of characters to be copied.</td></tr></table>	<i>dst</i> , <i>srs</i>	Pointers to character strings.	<i>n</i>	Count of characters to be copied.
<i>dst</i> , <i>srs</i>	Pointers to character strings.				
<i>n</i>	Count of characters to be copied.				
<b>DESCRIPTION</b>					
<b>strcpy()</b>	strcpy() copies characters in the string <i>srs</i> to <i>dst</i> , terminating at the first null character in <i>srs</i> , and returns <i>dst</i> to the caller. No bounds checking is done.				
<b>strncpy()</b>	strncpy() copies <i>srs</i> to <i>dst</i> , null-padding or truncating at <i>n</i> bytes, and returns <i>dst</i> . No bounds checking is done.				
<b>RETURN VALUES</b>	strcpy() and strncpy() return <i>dst</i> .				
<b>CONTEXT</b>	strcpy() can be called from user or interrupt context.				
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>				

<b>NAME</b>	strlen – determine the number of non-null bytes in a string
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt;  size_t <b>strlen</b>(const char *s);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<i>s</i> Pointer to a character string.
<b>DESCRIPTION</b>	strlen() returns the number of non-null bytes in the string argument <i>s</i> .
<b>RETURN VALUES</b>	strlen() returns the number of non-null bytes in <i>s</i> .
<b>CONTEXT</b>	strlen() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>

## strlog(9F)

<b>NAME</b>	strlog – submit messages to the log driver																								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/strlog.h&gt; #include &lt;sys/log.h&gt;  int <b>strlog</b>(short <i>mid</i>, short <i>sid</i>, char <i>level</i>, unsigned short <i>flags</i>, char             *<i>fmt</i>, ...);</pre>																								
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI).																								
	<table><tr><td><i>mid</i></td><td>Identification number of the module or driver submitting the message (in the case of a module, its <code>mi_idnum</code> value from <code>module_info(9S)</code>).</td></tr><tr><td><i>sid</i></td><td>Identification number for a particular minor device.</td></tr><tr><td><i>level</i></td><td>Tracing level for selective screening of low priority messages. Larger values imply less important information.</td></tr><tr><td><i>flags</i></td><td>Valid flag values are: <table><tr><td><code>SL_ERROR</code></td><td>Message is for error logger.</td></tr><tr><td><code>SL_TRACE</code></td><td>Message is for trace.</td></tr><tr><td><code>SL_NOTIFY</code></td><td>Mail copy of message to system administrator.</td></tr><tr><td><code>SL_CONSOLE</code></td><td>Log message to console.</td></tr><tr><td><code>SL_FATAL</code></td><td>Error is fatal.</td></tr><tr><td><code>SL_WARN</code></td><td>Error is a warning.</td></tr><tr><td><code>SL_NOTE</code></td><td>Error is a notice.</td></tr></table></td></tr><tr><td><i>fmt</i></td><td><code>printf(3C)</code> style format string. <code>%e</code>, <code>%g</code>, and <code>%G</code> formats are not allowed but <code>%s</code> is supported.</td></tr></table>	<i>mid</i>	Identification number of the module or driver submitting the message (in the case of a module, its <code>mi_idnum</code> value from <code>module_info(9S)</code> ).	<i>sid</i>	Identification number for a particular minor device.	<i>level</i>	Tracing level for selective screening of low priority messages. Larger values imply less important information.	<i>flags</i>	Valid flag values are: <table><tr><td><code>SL_ERROR</code></td><td>Message is for error logger.</td></tr><tr><td><code>SL_TRACE</code></td><td>Message is for trace.</td></tr><tr><td><code>SL_NOTIFY</code></td><td>Mail copy of message to system administrator.</td></tr><tr><td><code>SL_CONSOLE</code></td><td>Log message to console.</td></tr><tr><td><code>SL_FATAL</code></td><td>Error is fatal.</td></tr><tr><td><code>SL_WARN</code></td><td>Error is a warning.</td></tr><tr><td><code>SL_NOTE</code></td><td>Error is a notice.</td></tr></table>	<code>SL_ERROR</code>	Message is for error logger.	<code>SL_TRACE</code>	Message is for trace.	<code>SL_NOTIFY</code>	Mail copy of message to system administrator.	<code>SL_CONSOLE</code>	Log message to console.	<code>SL_FATAL</code>	Error is fatal.	<code>SL_WARN</code>	Error is a warning.	<code>SL_NOTE</code>	Error is a notice.	<i>fmt</i>	<code>printf(3C)</code> style format string. <code>%e</code> , <code>%g</code> , and <code>%G</code> formats are not allowed but <code>%s</code> is supported.
<i>mid</i>	Identification number of the module or driver submitting the message (in the case of a module, its <code>mi_idnum</code> value from <code>module_info(9S)</code> ).																								
<i>sid</i>	Identification number for a particular minor device.																								
<i>level</i>	Tracing level for selective screening of low priority messages. Larger values imply less important information.																								
<i>flags</i>	Valid flag values are: <table><tr><td><code>SL_ERROR</code></td><td>Message is for error logger.</td></tr><tr><td><code>SL_TRACE</code></td><td>Message is for trace.</td></tr><tr><td><code>SL_NOTIFY</code></td><td>Mail copy of message to system administrator.</td></tr><tr><td><code>SL_CONSOLE</code></td><td>Log message to console.</td></tr><tr><td><code>SL_FATAL</code></td><td>Error is fatal.</td></tr><tr><td><code>SL_WARN</code></td><td>Error is a warning.</td></tr><tr><td><code>SL_NOTE</code></td><td>Error is a notice.</td></tr></table>	<code>SL_ERROR</code>	Message is for error logger.	<code>SL_TRACE</code>	Message is for trace.	<code>SL_NOTIFY</code>	Mail copy of message to system administrator.	<code>SL_CONSOLE</code>	Log message to console.	<code>SL_FATAL</code>	Error is fatal.	<code>SL_WARN</code>	Error is a warning.	<code>SL_NOTE</code>	Error is a notice.										
<code>SL_ERROR</code>	Message is for error logger.																								
<code>SL_TRACE</code>	Message is for trace.																								
<code>SL_NOTIFY</code>	Mail copy of message to system administrator.																								
<code>SL_CONSOLE</code>	Log message to console.																								
<code>SL_FATAL</code>	Error is fatal.																								
<code>SL_WARN</code>	Error is a warning.																								
<code>SL_NOTE</code>	Error is a notice.																								
<i>fmt</i>	<code>printf(3C)</code> style format string. <code>%e</code> , <code>%g</code> , and <code>%G</code> formats are not allowed but <code>%s</code> is supported.																								
<b>DESCRIPTION</b>	<p><code>strlog()</code> expands the <code>printf(3C)</code> 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 <code>NLOGARGS</code>) 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.</p> <p>The messages can be retrieved with the <code>getmsg(2)</code> system call. The <i>flags</i> argument specifies the type of the message and where it is to be sent. <code>strace(1M)</code> receives messages from the log driver and sends them to the standard output. <code>strerr(1M)</code> receives error messages from the log driver and appends them to a file called <code>/var/adm/streams/error.mm-dd</code>, where <i>mm-dd</i> identifies the date of the error message.</p>																								
<b>RETURN VALUES</b>	<code>strlog()</code> returns 0 if it fails to submit the message to the <code>log(7D)</code> driver and 1 otherwise.																								

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

**FILES** | /var/adm/streams/error.*mm-dd*  
Error messages dated *mm-dd* appended by strerr(1M) from the log driver

**SEE ALSO** | strace(1M), strerr(1M), getmsg(2), log(7D), module\_info(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*

## strqget(9F)

<b>NAME</b>	strqget – get information about a queue or band of the queue
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>strqget</b>(queue_t *q, qfields_t what, unsigned char pri, void *valp);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI).
	<i>q</i> Pointer to the queue.
	<i>what</i> 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.
	<i>pri</i> Priority band of interest.
	<i>valp</i> The address of where to store the value of the requested field.
<b>DESCRIPTION</b>	<code>strqget()</code> 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.
<b>RETURN VALUES</b>	On success, 0 is returned and the value of the requested field is stored in the location pointed to by <i>valp</i> . An error number is returned on failure.
<b>CONTEXT</b>	<code>strqget()</code> can be called from user or interrupt context.
<b>SEE ALSO</b>	<code>strqset(9F)</code> , <code>queue(9S)</code> <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

<b>NAME</b>	strqset – change information about a queue or band of the queue								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int <b>strqset</b>(queue_t *q, qfields_t what, unsigned char pri, intptr_t val);</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>q</i> Pointer to the queue.</p> <p><i>what</i> Field of the queue structure (or the specified priority band) to return information about. Valid values are one of:</p> <table border="0"> <tr> <td>QHIWAT</td> <td>High water mark.</td> </tr> <tr> <td>QLOWAT</td> <td>Low water mark.</td> </tr> <tr> <td>QMAXPSZ</td> <td>Largest packet accepted.</td> </tr> <tr> <td>QMINPSZ</td> <td>Smallest packet accepted.</td> </tr> </table> <p><i>pri</i> Priority band of interest.</p> <p><i>val</i> The value for the field to be changed.</p>	QHIWAT	High water mark.	QLOWAT	Low water mark.	QMAXPSZ	Largest packet accepted.	QMINPSZ	Smallest packet accepted.
QHIWAT	High water mark.								
QLOWAT	Low water mark.								
QMAXPSZ	Largest packet accepted.								
QMINPSZ	Smallest packet accepted.								
<b>DESCRIPTION</b>	strqset () gives drivers and modules a way to change information about a queue or a particular band of a queue without directly accessing STREAMS data structures.								
<b>RETURN VALUES</b>	On success, 0 is returned. EINVAL is returned if an undefined attribute is specified.								
<b>CONTEXT</b>	strqset () can be called from user or interrupt context.								
<b>SEE ALSO</b>	strqget(9F), queue(9S) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>								
<b>NOTES</b>	When lowering existing values, set QMINPSZ before setting QMAXPSZ; when raising existing values, set QMAXPSZ before setting QMINPSZ.								

## STRUCT\_DECL(9F)

<b>NAME</b>	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
<b>SYNOPSIS</b>	<pre>#include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  <b>STRUCT_DECL</b>(<i>structname</i>, <i>handle</i>); <b>STRUCT_HANDLE</b>(<i>structname</i>, <i>handle</i>);  void <b>STRUCT_INIT</b>(<i>handle</i>, <i>model_t umodel</i>); void <b>STRUCT_SET_HANDLE</b>(<i>handle</i>, <i>model_t umodel</i>, void *<i>addr</i>);  <b>STRUCT_FGET</b>(<i>handle</i>, <i>field</i>); <b>STRUCT_FGETP</b>(<i>handle</i>, <i>field</i>); <b>STRUCT_FSET</b>(<i>handle</i>, <i>field</i>, <i>val</i>); <b>STRUCT_FSETP</b>(<i>handle</i>, <i>field</i>, <i>val</i>);  &lt;typeof field&gt; *<b>STRUCT_FADDR</b>(<i>handle</i>, <i>field</i>); struct <i>structname</i> *<b>STRUCT_BUF</b>(<i>handle</i>); size_t <b>SIZEOF_STRUCT</b>(<i>structname</i>, <i>umodel</i>); size_t <b>SIZEOF_PTR</b>(<i>umodel</i>); size_t <b>STRUCT_SIZE</b>(<i>handle</i>);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	The macros take the following parameters: <i>structname</i> The structure name (as would appear <i>after</i> the C keyword “struct”) of the native form. <i>umodel</i> A bit field containing either ILP32 model bit (DATAMODEL_ILP32), or the LP64 model get (DATAMODEL_LP64). In an <code>ioctl(9E)</code> , these bits will be present in the flag parameter; in a <code>devmap(9E)</code> , they will be present in the model parameter <code>mmap(9E)</code> and can call <code>ddi_mmap_get_model(9F)</code> to get the data model of the current thread. <i>handle</i> The variable name used to refer to a particular instance of a structure which is handled by these macros. <i>field</i> The field name within the structure contain substructures. If the structures contain substructures, unions, or arrays, then <i>field</i> can be whether complex expression could

**DESCRIPTION**

occur after the first "." or "->".

The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.

The macros can be broken up into two main categories, the macros that declare and initialize structure handles and the macros that operate on these structures using the structure handles.

**Declaration and Initialization Macros**

The macros `STRUCT_DECL()` and `STRUCT_HANDLE()` declare structure handles on the stack, whereas the macros `STRUCT_INIT()` and `STRUCT_SET_HANDLE()` initialize the structure handles to point to an instance of the native form structure.

The macros `STRUCT_HANDLE()` and `STRUCT_SET_HANDLE()` are used to declare and initialize a structure handle to an existing data structure, for example, `ioctl`s within a `STREAMS` module.

The macros `STRUCT_DECL()` and `STRUCT_INIT()`, on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by `STRUCT_DECL()`, that is, any standard character or block device driver `ioctl(9E)` routine that needs to copy in data from a user-mode program.

`STRUCT_DECL(structname, handle)`

Declares a "structure handle" for a "struct" and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. *handle* is a variable name and is declared as a variable by this macro.

`void STRUCT_INIT(handle, model_t umodel)`

Initializes *handle* to point to the instance allocated by `STRUCT_DECL()`, it also sets data model for *handle* to *umodel*, and must be called before any access is made through the macros that operate on these structures. When used in an `ioctl(9E)` routine *umodel* is the flag parameter; in a `adevmap(9E)` routine *umodel* is the model parameter and in a `mmap(9E)` routine, is the return value of `ddi_mmap_get_model(9F)`. This macro is intended for handles created with `STRUCT_DECL()` only.

`STRUCT_HANDLE(structname, handle)`

Declares a "structure handle" *handle* but unlike `STRUCT_DECL()` does not allocate an instance of "struct".

`void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)`

Initializes to point to the native form instance at *addr*, it also sets the data model for *handle* to *umodel*. This is intended for handles created with `STRUCT_HANDLE()`. Fields cannot be referenced via the *handle* until this macro has been invoked. Typically, *addr* is the address of the native form structure containing the user-mode programs data. When used in an `ioctl(9E)` *umodel* is the flag parameter, in a `devmap(9E)` routine is the model parameter and in a `mmap(9E)` routine, *umodel* is the return value of `ddi_mmap_get_model(9F)`.

## STRUCT\_DECL(9F)

### Operation Macros

size\_t STRUCT\_SIZE(handle)

Returns size of the structure referred to by *handle*. It will return the size depending upon the data model associated with *handle*. If the data model stored by STRUCT\_INIT() or STRUCT\_SET\_HANDLE() was DATAMODEL\_ILP32, it will return the size of the ILP32 form, else it will return the size of the native form.

STRUCT\_FGET(handle, field)

Returns the contents of *field* in the structure described by *handle* according to the data model associated with *handle*.

STRUCT\_FGETP(handle, field)

This is the same as STRUCT\_FGET() except that the *field* in question is a pointer of some kind. This macro will cast caddr32\_t to a (void \*) when it is accessed. Failure to use this macro for a pointer will lead to compiler warnings or failures.

STRUCT\_FSET(handle, field, val)

Assigns *val* to the (non pointer) in the structure described by *handle*. It should not be used within any other expression, but rather only as a statement.

STRUCT\_FSETP(handle, field, val)

Returns a pointer to the in the structure described by *handle*.

struct structname \*STRUCT\_BUF(handle)

Returns a pointer to the native mode instance of the structure described by *handle*.

### Macros Not Using Handles

size\_t SIZEOF\_STRUCT(structname, umodel)

Returns size of *structname* based on *umodel*.

size\_t SIZEOF\_PTR(umodel)

Returns the size of a pointer based on *umodel*.

### EXAMPLES

#### EXAMPLE 1 Copying a Structure

The following example uses an ioctl(9E) on a regular character device that copies a data structure that looks like this into the kernel:

```
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, intp_t arg, int mode,
        cred_t *cr, int *rval_p);
{
    STRUCT_DECL(opdata, op);

    if (cmd != OPONE)
        return (ENOTTY);

    STRUCT_INIT(op, mode);

    if (copyin((void *)data,
              STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);

    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles ioctl(9E) data (M\_IOCTL) 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_ioctldata(queue_t *q, mblk_t *msgp)
{
    mblk_t *data; /* message block descriptor */
    STRUCT_HANDLE(strbuf, sb);

    /* copyin the data */
    if (mi_copy_state(q, mp, &data) == -1) {
        return;
    }
}
```

## STRUCT\_DECL(9F)

**EXAMPLE 4** Using STRUCT\_HANDLE() and STRUCT\_SET\_HANDLE() (Continued)

```
        STRUCT_SET_HANDLE(sb, ((struct iocblk *)msgp->b_rptr)->ioc_flag,  
                           (void *)data->b_rptr);  
        if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {  
            mi_copy_done(q, msgp, EINVAL);  
            return;  
        }  
    }  
}
```

**SEE ALSO** devmap(9E), ioctl(9E), mmap(9E), ddi\_mmap\_get\_model(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

<b>NAME</b>	swab – swap bytes in 16-bit halfwords
<b>SYNOPSIS</b>	<pre>#include &lt;sys/sunddi.h&gt; void swab(void *src, void *dst, size_t nbytes);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	<p>Architecture independent level 1 (DDI/DKI).</p> <p><i>src</i>        A pointer to the buffer containing the bytes to be swapped.</p> <p><i>dst</i>        A pointer to the destination buffer where the swapped bytes will be written. If <i>dst</i> is the same as <i>src</i> the buffer will be swapped in place.</p> <p><i>nbytes</i>     Number of bytes to be swapped, rounded down to the nearest half-word.</p>
<b>DESCRIPTION</b>	swab() copies the bytes in the buffer pointed to by <i>src</i> to the buffer pointer to by <i>dst</i> , swapping the order of adjacent bytes in half-word pairs as the copy proceeds. A total of <i>nbytes</i> bytes are copied, rounded down to the nearest half-word.
<b>CONTEXT</b>	swab() can be called from user or interrupt context.
<b>SEE ALSO</b>	<i>Writing Device Drivers</i>
<b>NOTES</b>	Since swab() operates byte-by-byte, it can be used on non-aligned buffers.

## testb(9F)

<b>NAME</b>	testb – check for an available buffer
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  int testb(size_t size, uint_t pri);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>size</i> Size of the requested buffer. <i>pri</i> Priority of the allocb request.
<b>DESCRIPTION</b>	testb() checks to see if an allocb(9F) call is likely to succeed if a buffer of <i>size</i> bytes at priority <i>pri</i> is requested. Even if testb() returns successfully, the call to allocb(9F) can fail. The <i>pri</i> argument is no longer used, but is retained for compatibility.
<b>RETURN VALUES</b>	Returns 1 if a buffer of the requested size is available, and 0 if one is not.
<b>CONTEXT</b>	testb() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> testb() example</p> <p>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.</p> <p>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.</p> <pre>1 xxxsrv(q) 2     queue_t *q; 3     { 4     mblk_t *mp; 5     mblk_t *nmp; 6     . . . 7     if ((nmp = copymsg(mp)) == NULL) { 8         putbq(q, mp); 9         timeout(tryagain, (intptr_t)q, drv_usectohz(100000)); 10    } 11    . . . 12 }</pre>

**EXAMPLE 1** testb() example *(Continued)*

```

12
13 tryagain(q)
14     queue_t *q;
15 {
16     register int can_alloc = 0;
17     register int num_blks = 0;
18     register mblk_t *mp;
19
20     if (!q->q_first)
21         return;
22     for (mp = q->q_first; mp; mp = mp->b_cont) {
23         num_blks++;
24         can_alloc += testb((mp->b_datap->db_lim -
25             mp->b_datap->db_base), BPRI_MED);
26     }
27     if (num_blks == can_alloc)
28         qenable(q);
29     else
30         timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
31 }

```

**SEE ALSO** allocb(9F), bufcall(9F), copymsg(9F), timeout(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES** The *pri* argument is provided for compatibility only. Its value is ignored.

## timeout(9F)

<b>NAME</b>	timeout – execute a function after a specified length of time
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/conf.h&gt;  timeout_id_t timeout(void (*func)(void*), void *arg, clock_t ticks);</pre>
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI). <i>func</i> Kernel function to invoke when the time increment expires. <i>arg</i> Argument to the function. <i>ticks</i> Number of clock ticks to wait before the function is called. Use <code>drv_usec2ohz(9F)</code> to convert microseconds to clock ticks.
<b>DESCRIPTION</b>	<p>The <code>timeout()</code> 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.</p> <p>The function called by <code>timeout()</code> must adhere to the same restrictions as a driver soft interrupt handler.</p> <p>The function called by <code>timeout()</code> is run in interrupt context and must not sleep or call other functions that might sleep.</p> <p>The <code>delay(9F)</code> function calls <code>timeout()</code>. Because <code>timeout()</code> is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use <code>cv_timedwait(9F)</code> rather than <code>delay()</code>.</p>
<b>RETURN VALUES</b>	<code>timeout()</code> returns an opaque non-zero timeout identifier that can be passed to <code>untimeout(9F)</code> to cancel the request.
<b>CONTEXT</b>	<code>timeout()</code> can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using <code>timeout()</code></p> <p>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.</p> <pre>static void xtimeout_handler(void *arg) {     struct xxstate *xsp = (struct xxstate *)arg;     mutex_enter(&amp;xsp-&gt;lock);     cv_signal(&amp;xsp-&gt;cv);     xsp-&gt;flags  = TIMED_OUT;     mutex_exit(&amp;xsp-&gt;lock);     xsp-&gt;timeout_id = 0; } static uint_t xxintr(caddr_t arg) {</pre>

**EXAMPLE 1** Using `timeout()` (Continued)

```

    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(xstimeout_handler,
        xsp, (5 * drv_usectohz(1000000)));
    mutex_enter(&xsp->lock);
    while (/* Waiting for interrupt or timeout */)
        cv_wait(&xsp->cv, &xsp->lock);
    if (xsp->flags & TIMED_OUT)
        cmn_err(CE_WARN, "Device not responding");
    .
    .
    .
    mutex_exit(&xsp->lock);
    .
    .
    .
}

```

**SEE ALSO** [bufcall\(9F\)](#), [cv\\_timedwait\(9F\)](#), [ddi\\_in\\_panic\(9F\)](#), [delay\(9F\)](#), [drv\\_usectohz\(9F\)](#), [untimeout\(9F\)](#)

*Writing Device Drivers*

## uiomove(9F)

<b>NAME</b>	uiomove – copy kernel data using uio structure								
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/uio.h&gt;  int uiomove(caddr_t address, size_t nbytes, enum uio_rw rwflag, uio_t             *uio_p);</pre>								
<b>INTERFACE LEVEL PARAMETERS</b>	Architecture independent level 1 (DDI/DKI).  <table><tr><td><i>address</i></td><td>Source/destination kernel address of the copy.</td></tr><tr><td><i>nbytes</i></td><td>Number of bytes to copy.</td></tr><tr><td><i>rwflag</i></td><td>Flag indicating read or write operation. Possible values are UIO_READ and UIO_WRITE.</td></tr><tr><td><i>uio_p</i></td><td>Pointer to the uio structure for the copy.</td></tr></table>	<i>address</i>	Source/destination kernel address of the copy.	<i>nbytes</i>	Number of bytes to copy.	<i>rwflag</i>	Flag indicating read or write operation. Possible values are UIO_READ and UIO_WRITE.	<i>uio_p</i>	Pointer to the uio structure for the copy.
<i>address</i>	Source/destination kernel address of the copy.								
<i>nbytes</i>	Number of bytes to copy.								
<i>rwflag</i>	Flag indicating read or write operation. Possible values are UIO_READ and UIO_WRITE.								
<i>uio_p</i>	Pointer to the uio structure for the copy.								
<b>DESCRIPTION</b>	<p>The uiomove () function copies <i>nbytes</i> of data to or from the space defined by the uio structure (described in uio(9S)) and the driver.</p> <p>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.</p> <p><i>rwflag</i> indicates the direction of the transfer. If UIO_READ is set, the data will be transferred from <i>address</i> to the buffer(s) described by <i>uio_p</i>. If UIO_WRITE is set, the data will be transferred from the buffer(s) described by <i>uio_p</i> to <i>address</i>.</p> <p>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.</p> <p>This function automatically handles page faults. <i>nbytes</i> does not have to be word-aligned.</p>								
<b>RETURN VALUES</b>	uiomove () returns 0 upon success or EFAULT on failure.								
<b>CONTEXT</b>	User context only, if uio_segflg is set to UIO_USERSPACE. User or interrupt context, if uio_segflg is set to UIO_SYSSPACE.								
<b>SEE ALSO</b>	ureadc(9F), uwritec(9F), iovec(9S), uio(9S)  <i>Writing Device Drivers</i>								
<b>WARNINGS</b>	If uio_segflg is set to UIO_SYSSPACE and <i>address</i> is selected from user space, the system may panic.								

<b>NAME</b>	unbufcall – cancel a pending bufcall request
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  void unbufcall(bufcall_id_t id);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>id</i> Identifier returned from bufcall(9F) or esbbcall(9F).
<b>DESCRIPTION</b>	unbufcall cancels a pending bufcall() or esbbcall() request. The argument <i>id</i> is a non-zero identifier for the request to be cancelled. <i>id</i> 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.
<b>RETURN VALUES</b>	None.
<b>CONTEXT</b>	unbufcall() can be called from user or interrupt context.
<b>SEE ALSO</b>	<p>bufcall(9F), esbbcall(9F)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

## unlinkb(9F)

<b>NAME</b>	unlinkb – remove a message block from the head of a message
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt;  mblk_t *unlinkb(mblk_t *mp);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>mp</i> Pointer to the message.
<b>DESCRIPTION</b>	unlinkb() removes the first message block from the message pointed to by <i>mp</i> . A new message, minus the removed message block, is returned.
<b>RETURN VALUES</b>	If successful, unlinkb() returns a pointer to the message with the first message block removed. If there is only one message block in the message, NULL is returned.
<b>CONTEXT</b>	unlinkb() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> unlinkb() example</p> <p>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.</p> <pre>1 mblk_t * 2 makedata(mp) 3     mblk_t *mp; 4 { 5     mblk_t *nmp; 6 7     nmp = unlinkb(mp); 8     freeb(mp); 9     return(nmp); 10 }</pre>
<b>SEE ALSO</b>	linkb(9F)  <i>Writing Device Drivers</i>  <i>STREAMS Programming Guide</i>

<b>NAME</b>	untimeout – cancel previous timeout function call
<b>SYNOPSIS</b>	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/conf.h&gt;  clock_t untimeout(timeout_id_t id);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>id</i> Opaque timeout ID from a previous timeout(9F) call.
<b>DESCRIPTION</b>	<p>untimeout () 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.</p> <p>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.</p>
<b>RETURN VALUES</b>	untimeout () returns -1 if the <i>id</i> is not found. Otherwise, it returns an integer value greater than or equal to 0.
<b>CONTEXT</b>	untimeout () can be called from user or interrupt context.
<b>EXAMPLES</b>	<p>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.</p> <pre>static void xxtimeout_handler(void *arg) {     struct xxstate *xsp = (struct xxstate *)arg;     mutex_enter(&amp;xsp-&gt;lock);     cv_signal(&amp;xsp-&gt;cv);     xsp-&gt;flags  = TIMED_OUT;     mutex_exit(&amp;xsp-&gt;lock);     xsp-&gt;timeout_id = 0; }  static uint_t xxintr(caddr_t arg) {     struct xxstate *xsp = (struct xxstate *)arg;     .     .     .     mutex_enter(&amp;xsp-&gt;lock);     /* Service interrupt */     cv_signal(&amp;xsp-&gt;cv);     mutex_exit(&amp;xsp-&gt;lock);</pre>

## untimeout(9F)

```
        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(xtimeout_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*

<b>NAME</b>	ureadc – add character to a uio structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/uio.h&gt; #include &lt;sys/types.h&gt;  int <b>ureadc</b>(int <i>c</i>, uio_t *<i>uio_p</i>);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<p><i>c</i>           The character added to the uio(9S) structure.</p> <p><i>uio_p</i>        Pointer to the uio(9S) structure.</p>
<b>DESCRIPTION</b>	ureadc() transfers the character <i>c</i> into the address space of the uio(9S) structure pointed to by <i>uio_p</i> , and updates the uio structure as for uiomove(9F).
<b>RETURN VALUES</b>	0 is returned on success and EFAULT on failure.
<b>CONTEXT</b>	ureadc() can be called from user or interrupt context.
<b>SEE ALSO</b>	uiomove(9F), uwritec(9F), iovec(9S), uio(9S)
	<i>Writing Device Drivers</i>

## uwritec(9F)

<b>NAME</b>	uwritec – remove a character from a uio structure
<b>SYNOPSIS</b>	<pre>#include &lt;sys/uio.h&gt;  int <b>uwritec</b>(uio_t *uio_p);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI)
<b>PARAMETERS</b>	<i>uio_p</i> Pointer to the uio(9S) structure
<b>DESCRIPTION</b>	uwritec() returns a character from the uio structure pointed to by <i>uio_p</i> and updates the uio structure. See uiomove(9F).
<b>RETURN VALUES</b>	The next character for processing is returned on success, and -1 is returned if uio is empty or if there is an error.
<b>CONTEXT</b>	uwritec() can be called from user or interrupt context.
<b>SEE ALSO</b>	uiomove(9F), ureadc(9F), iovec(9S), uio(9S) <i>Writing Device Drivers</i>

<b>NAME</b>	va_arg, va_start, va_copy, va_end – handle variable argument list				
<b>SYNOPSIS</b>	<pre>#include &lt;sys/varargs.h&gt;  void <b>va_start</b>(va_list <i>pvar</i>, void <i>parmN</i>);  (<i>type</i> *) <b>va_arg</b>(va_list <i>pvar</i>, <i>type</i>);  void <b>va_copy</b>(va_list <i>dest</i>, va_list <i>src</i>);  void <b>va_end</b>(va_list <i>pvar</i>);</pre>				
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).				
<b>PARAMETERS</b>					
<b>va_start()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>pvar</i></td> <td>Pointer to variable argument list.</td> </tr> <tr> <td><i>name</i></td> <td>Identifier of rightmost parameter in the function definition.</td> </tr> </table>	<i>pvar</i>	Pointer to variable argument list.	<i>name</i>	Identifier of rightmost parameter in the function definition.
<i>pvar</i>	Pointer to variable argument list.				
<i>name</i>	Identifier of rightmost parameter in the function definition.				
<b>va_arg()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>pvar</i></td> <td>Pointer to variable argument list.</td> </tr> <tr> <td><i>type</i></td> <td>Type name of the next argument to be returned.</td> </tr> </table>	<i>pvar</i>	Pointer to variable argument list.	<i>type</i>	Type name of the next argument to be returned.
<i>pvar</i>	Pointer to variable argument list.				
<i>type</i>	Type name of the next argument to be returned.				
<b>va_copy()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>dest</i></td> <td>Destination variable argument list.</td> </tr> <tr> <td><i>src</i></td> <td>Source variable argument list.</td> </tr> </table>	<i>dest</i>	Destination variable argument list.	<i>src</i>	Source variable argument list.
<i>dest</i>	Destination variable argument list.				
<i>src</i>	Source variable argument list.				
<b>va_end()</b>	<table border="0"> <tr> <td style="padding-right: 20px;"><i>pvar</i></td> <td>Pointer to variable argument list.</td> </tr> </table>	<i>pvar</i>	Pointer to variable argument list.		
<i>pvar</i>	Pointer to variable argument list.				
<b>DESCRIPTION</b>	<p>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 <code>varargs()</code> 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.</p> <p><code>va_list</code> is the type defined for the variable used to traverse the list of arguments.</p> <p><code>va_start()</code> is called to initialize <code>pvar</code> to the beginning of the variable argument list. <code>va_start()</code> must be invoked before any access to the unnamed arguments. The parameter <code>name</code> 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 <code>register</code> 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.</p> <p><code>va_arg()</code> expands to an expression that has the type and value of the next argument in the call. The parameter <code>pvar</code> must be initialized by <code>va_start()</code>. Each invocation of <code>va_arg()</code> modifies <code>pvar</code> so that the values of successive arguments are returned in turn. The parameter <code>type</code> 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</p>				

## va\_arg(9F)

has the specified type can be obtained by postfixing a *\** to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The `va_copy()` macro saves the state represented by the `va_list` *src* in the `va_list` *dest*. The `va_list` passed as *dest* should not be initialized by a previous call to `va_start()`. It then must be passed to `va_end()` before being reused as a parameter to `va_start()` or as the *dest* parameter of a subsequent call to `va_copy()`. The behavior is undefined if any of these restrictions are not met.

The `va_end()` macro is used to clean up. It invalidates *pvar* for use (unless `va_start()` is invoked again).

Multiple traversals, each bracketed by a call to `va_start()` and `va_end()`, are possible.

### EXAMPLES **EXAMPLE 1** Creating a Variable Length Command

The following example uses these routines to create a variable length command. This might be useful for a device that provides for a variable-length command set. `ncmdbytes` is the number of bytes in the command. The new command is written to `cmdp`.

```
static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
     * Write variable-length command to destination
     */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}
```

**SEE ALSO** `vcmn_err(9F)`, `vsprintf(9F)`

**NOTES** It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame.

Specifying a second argument of `char` or `short` to `va_arg` makes your code non-portable, because arguments seen by the called function are not `char` or `short`. C converts `char` and `short` arguments to `int` before passing them to a function.

<b>NAME</b>	vsprintf – format characters in memory
<b>SYNOPSIS</b>	<pre>#include &lt;sys/varargs.h&gt; #include &lt;sys/ddi.h&gt; #include &lt;sys/sunddi.h&gt;  char *<b>vsprintf</b>(char *buf, const char *fmt, va_list ap);</pre>
<b>INTERFACE LEVEL</b>	Solaris DDI specific (Solaris DDI).
<b>PARAMETERS</b>	<p><i>buf</i> Pointer to a character string.</p> <p><i>fmt</i> Pointer to a character string.</p> <p><i>ap</i> Pointer to a variable argument list.</p>
<b>DESCRIPTION</b>	<p><code>vsprintf()</code> builds a string in <i>buf</i> under the control of the format <i>fmt</i>. The format is a character string with either plain characters, which are simply copied into <i>buf</i>, or conversion specifications, each of which converts zero or more arguments, again copied into <i>buf</i>. 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 <i>buf</i>.</p> <p><i>ap</i> contains the list of arguments used by the conversion specifications in <i>fmt</i>. <i>ap</i> is a variable argument list and must be initialized by calling <code>va_start(9F)</code>. <code>va_end(9F)</code> is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by <code>va_start(9F)</code> and <code>va_end(9F)</code>, are possible.</p> <p>Each conversion specification is introduced by the <code>%</code> character, after which the following appear in sequence:</p> <p>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.</p> <p>An optional <code>l</code> (<code>ll</code>) specifying that a following <code>d</code>, <code>D</code>, <code>o</code>, <code>O</code>, <code>x</code>, <code>X</code>, or <code>u</code> conversion character applies to a long (<code>long long</code>) integer argument. An <code>l</code> (<code>ll</code>) before any other conversion character is ignored.</p> <p>A character indicating the type of conversion to be applied:</p> <p><code>d,D,o,O,x,X,u</code>  The integer argument is converted to signed decimal (<code>d</code>, <code>D</code>), unsigned octal (<code>o</code>, <code>O</code>), unsigned hexadecimal (<code>x</code>, <code>X</code>) or unsigned decimal (<code>u</code>), respectively, and copied. The letters <code>abcdef</code> are used for <code>x</code> conversion. The letters <code>ABCDEF</code> are used for <code>X</code> conversion.</p> <p><code>c</code>  The character value of the argument is copied.</p> <p><code>b</code>  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</p>

## vsprintf(9F)

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);
    name = ddi_binding_name(dip);

    /* format buf using fmt and arguments contained in ap */
    va_start(ap, fmt);
    vsprintf(buf, fmt, ap);
    va_end(ap);
}
```

**EXAMPLE 1** Using `vsprintf()` (Continued)

```
/* pass formatted string to cmn_err(9F) */
cmn_err(level, "%s%d: %s", name, instance, buf);
}
```

**SEE ALSO** `cmn_err(9F)`, `ddi_binding_name(9F)`, `ddi_get_instance(9F)`, `va_arg(9F)`

*Writing Device Drivers*

## WR(9F)

<b>NAME</b>	WR, wr – get pointer to the write queue for this module or driver
<b>SYNOPSIS</b>	<pre>#include &lt;sys/stream.h&gt; #include &lt;sys/ddi.h&gt;  queue_t *WR(queue_t *q);</pre>
<b>INTERFACE LEVEL</b>	Architecture independent level 1 (DDI/DKI).
<b>PARAMETERS</b>	<i>q</i> Pointer to the <i>read</i> queue whose <i>write</i> queue is to be returned.
<b>DESCRIPTION</b>	<p>The WR() function accepts a <i>read</i> queue pointer as an argument and returns a pointer to the <i>write</i> queue of the same module.</p> <p><b>CAUTION:</b> Make sure the argument to this function is a pointer to a <i>read</i> queue. WR() will not check for queue type, and a system panic could result if the pointer is not to a <i>read</i> queue.</p>
<b>RETURN VALUES</b>	The pointer to the <i>write</i> queue.
<b>CONTEXT</b>	WR() can be called from user or interrupt context.
<b>EXAMPLES</b>	<p><b>EXAMPLE 1</b> Using WR()</p> <p>In a STREAMS close(9E) routine, the driver or module is passed a pointer to the <i>read</i> queue. These usually are set to the address of the module-specific data structure for the minor device.</p> <pre>1 xxxclose(q, flag) 2     queue_t *q; 3     int flag; 4     { 5         q-&gt;q_ptr = NULL; 6         WR(q)-&gt;q_ptr = NULL; 7     }</pre>
<b>SEE ALSO</b>	close(9E), OTHERQ(9F), RD(9F) <i>Writing Device Drivers</i> <i>STREAMS Programming Guide</i>

# Index

---

## Numbers and Symbols

- 32-bit driver ID management routines —
  - id32\_alloc, 455
- 32-bit driver ID management routines —
  - id32\_free, 455
- 32-bit driver ID management routines —
  - id32\_lookup, 455

## A

- activate a new DMA window —
  - ddi\_dma\_getwin, 272
- add a fully initialized kstat to the system —
  - kstat\_install, 471
- add a soft interrupt, — ddi\_add\_softintr, 213
- add an interrupt handler, — ddi\_add\_intr, 209
- address, return mapped virtual address —
  - csx\_GetMappedAddr, 110
- adjmsg — trim bytes from a message, 40
- Device power cycle advisory check —
  - pm\_trans\_check, 529
- allocate and free a scsi\_pkt structure —
  - scsi\_hba\_pkt\_alloc, 595
- allocate and free transport structures —
  - scsi\_hba\_tran\_alloc, 598
- allocate and free a scsi\_pkt structure —
  - scsi\_hba\_pkt\_alloc, scsi\_hba\_pkt\_free, 595
- allocate and free transport structures —
  - scsi\_hba\_tran\_alloc, scsi\_hba\_tran\_free, 598
- allocate a message block — allocb, 41
- allocate and free non-sequentially accessed memory
  - allocate and free non-sequentially accessed memory (Continued)
    - ddi\_iopb\_alloc, 316
    - ddi\_iopb\_free, 316
  - allocate DMA handle —
    - ddi\_dma\_alloc\_handle, 253
  - allocate kernel memory
    - ddi\_umem\_alloc, 400
    - ddi\_umem\_free, 400
    - ddi\_umem\_zalloc, 400
    - kmem\_alloc, 461
    - kmem\_free, 461
    - kmem\_zalloc, 461
  - allocate memory for DMA transfer —
    - ddi\_dma\_mem\_alloc, 276
  - allocate space — rmalloc, 562
  - allocate space from a resource map —
    - rmalloc\_wait, 566
  - allocb — allocate a message block, 41
  - allow 64 bit transfers on SBus —
    - ddi\_dma\_set\_sbus64, 290
  - anocancel — prevent cancellation of asynchronous I/O request, 44
  - aphysio — perform asynchronous physical I/O, 45
  - assert — expression verification, 47
  - associate STREAMS queue with bottom driver in queue — qassociate, 547
  - asynchronous physical I/O — aphysio, 45
  - asynchronous STREAMS perimeter upgrade —
    - qwriter, 560

## B

`bcopy` — copy data between address locations in kernel, 51

binds a system buffer to a DMA handle — `ddi_dma_buf_bind_handle`, 256

binds an address to a DMA handle — `ddi_dma_addr_bind_handle`, 248

`bioclon` — clone another buffer, 53

`bioerror` — indicate error in buffer header, 58

`biofini` — uninitialized a buffer structure, 59

`bioinit` — initialize a buffer structure, 60

`biomodified` — check if a buffer is modified, 61

`bioreset` — reuse a private buffer header after I/O is complete, 62

`biosize` — returns size of a buffer structure, 63

`bufcall` — call a function when a buffer becomes available, 69, 649

`bufcall`, call a function when a buffer becomes available, 69

buffer header

- indicate error — `bioerror`, 58
- reuse a private buffer header after I/O is complete — `bioreset`, 62

busy-wait for specified interval — `drv_usecwait`, 427

byte streams, compare two — `bcmp`, 50

bytes, size

- convert size in pages — `ptob`, 534
- convert to size in memory pages (round down) — `btop`, 67
- convert to size in memory pages (round up) — `btopr`, 68

## C

call a function when a buffer becomes available — `qbufcall`, 549

call a function when a buffer becomes available, `bufcall`, 69

call a STREAMS put procedure — `put`, 537

cancel a pending `qbufcall` request — `qunbufcall`, 556

cancel previous timeout function call — `quntimeout`, 557

cancellation of asynchronous I/O — `anocancel`, 44

character strings

- compare two null terminated strings — `strcmp`, `strncmp`, 631
- convert between an integer and a decimal string — `stoi`, `numtos`, 629
- copy a string from one location to another — `strcpy`, `strncpy`, 632
- determine the number of non-null bytes in a string — `strlen`, 633
- find a character in a string — `strchr`, 630
- format in memory — `sprintf`, 627

check data access and DMA handles, 221

check device state, 303

check for an available buffer — `testb`, 644

check for the existence of a property — `ddi_prop_exists`, 360

check if a buffer is modified — `biomodified`, 61

CIS tuple

- first tuple — `csx_GetFirstTuple`, 107
- next tuple — `csx_GetNextTuple`, 107

clear client event mask — `csx_ReleaseSocketMask`, 196

client, register client — `csx_RegisterClient`, 175

client event mask

- return client event mask — `csx_GetEventMask`, 204
- set client event mask — `csx_SetEventMask`, 204

client return

- `csx_GetFirstClient`, 105
- `csx_GetNextClient`, 105

clone another buffer — `bioclon`, 53

concatenate two message blocks — `linkb`, 475

condition variable routines, driver

- `condvar`, 81
- `cv_broadcast`, 81
- `cv_init`, 81
- `cv_signal`, 81
- `cv_timedwait`, 81
- `cv_timedwait_sig`, 81
- `cv_wait`, 81
- `cv_wait_sig`, 81

configure PC Card and socket — `csx_RequestConfiguration`, 184

control driver notification of user accesses — `ddi_mapdev_intercept`, 331

`ddi_mapdev_nointercept`, 331

control device components' availability for Power Management

- pm\_busy\_component, 519
- pm\_idle\_component, 519

control the validation of memory address translations

- devmap\_load, 419
- devmap\_unload, 419

convert a DMA segment to a DMA address cookie — ddi\_dma\_segtocookie, 288

convert clock ticks to microseconds — drv\_hztousec, 424

convert device sizes — csx\_ConvertSize, 94

convert device speeds — csx\_ConvertSpeed, 95

convert error return codes to text strings — csx\_Error2Text, 101

convert events to text strings — csx\_Event2Text, 102

convert microseconds to clock ticks — drv\_usecsthz, 426

copy data from one device register to another device register — ddi\_device\_copy, 231

create minor nodes for client — csx\_MakeDeviceNode, 117

create a minor node for this device — ddi\_create\_minor\_node, 229

create and initialize a new kstat — kstat\_create, 469

create driver-controlled mapping of device — ddi\_mapdev, 328

csx\_AccessConfigurationRegister — read or write a PC Card Configuration Register, 92

csx\_ConvertSize — convert device sizes, 94

csx\_ConvertSpeed — convert device speeds, 95

csx\_CS\_DDI\_Info — obtain DDI information, 96

csx\_DeregisterClient — remove client from Card Services list, 98

csx\_DupHandle — duplicate access handle, 99

csx\_Error2Text — convert error return codes to text strings, 101

csx\_Event2Text — convert events to text strings, 102

csx\_FreeHandle — free access handle, 103

csx\_Get16 — read from device register, 104

csx\_Get32 — read from device register, 104

csx\_Get64 — read from device register, 104

csx\_Get8 — read from device register, 104

csx\_GetEventMask — return client event mask, 204

csx\_GetFirstClient — return first client, 105

csx\_GetFirstTuple — return first CIS tuple, 107

csx\_GetHandleOffset — return current access handle offset, 109

csx\_GetMappedAddr — return mapped virtual address, 110

csx\_GetNextClient — return next client, 105

csx\_GetNextTuple — return next CIS tuple, 107

csx\_GetStatus — return status of PC Card and socket, 111

csx\_GetTupleData — return data portion of tuple, 115

csx\_MakeDeviceNode — create minor nodes for client, 117

csx\_MapLogSocket — return physical socket number, 119

csx\_MapMemPage — map memory area on PC Card, 120

csx\_ModifyConfiguration — modify PC Card configuration, 121

csx\_ModifyWindow — modify window attributes, 123

csx\_Parse\_CISTPL\_BATTERY — parse Battery Replacement Date tuple, 125

csx\_Parse\_CISTPL\_BYTEORDER — parse Byte Order tuple, 126

csx\_Parse\_CISTPL\_CFTABLE\_ENTRY — parse Card Configuration Table tuple, 128

csx\_Parse\_CISTPL\_CONFIG — parse Configuration tuple, 134

csx\_Parse\_CISTPL\_DATE — parse Card Initialization Date tuple, 137

csx\_Parse\_CISTPL\_DEVICE — parse Device Information tuple for Common Memory, 138

csx\_Parse\_CISTPL\_DEVICE\_A — parse Device Information tuple for Attribute Memory, 138

csx\_Parse\_CISTPL\_DEVICE\_OA — parse Other Condition Device Information tuple for Attribute Memory, 138

csx\_Parse\_CISTPL\_DEVICE\_OC — parse Other Condition Device Information tuple for Common Memory, 138

csx\_Parse\_CISTPL\_DEVICEGEO — parse Device Geo tuple, 141

csx\_Parse\_CISTPL\_DEVICEGEO\_A — parse Device Geo A tuple, 143

csx\_Parse\_CISTPL\_FORMAT — parse Data Recording Format tuple, 145  
 csx\_Parse\_CISTPL\_FUNCE — parse Function Extension tuple, 147  
 csx\_Parse\_CISTPL\_FUNCID — parse Function Identification tuple, 155  
 csx\_Parse\_CISTPL\_GEOMETRY — parse Geometry tuple, 157  
 csx\_Parse\_CISTPL\_JEDEC\_A — parse JEDEC Identifier tuple for Attribute Memory, 158  
 csx\_Parse\_CISTPL\_JEDEC\_C — parse JEDEC Identifier tuple for Common Memory, 158  
 csx\_Parse\_CISTPL\_LINKTARGET — parse Link Target tuple, 160  
 csx\_Parse\_CISTPL\_LONGLINK\_A — parse Long Link A tuple, 161  
 csx\_Parse\_CISTPL\_LONGLINK\_C — parse Long Link C tuple, 161  
 csx\_Parse\_CISTPL\_LONGLINK\_MFC — parse Multi-Function tuple, 163  
 csx\_Parse\_CISTPL\_MANFID — parse Manufacturer Identification tuple, 165  
 csx\_Parse\_CISTPL\_ORG — parse Data Organization tuple, 166  
 csx\_Parse\_CISTPL\_SPCL — parse Special Purpose tuple, 167  
 csx\_Parse\_CISTPL\_SWIL — parse Software Interleaving tuple, 169  
 csx\_Parse\_CISTPL\_VERS\_1 — parse Level-1 Version/Product Information tuple, 170  
 csx\_Parse\_CISTPL\_VERS\_2 — parse Level-2 Version and Information tuple, 171  
 csx\_ParseTuple — generic tuple parser, 172  
 csx\_Put16 — write to device register, 174  
 csx\_Put32 — write to device register, 174  
 csx\_Put64 — write to device register, 174  
 csx\_Put8 — write to device register, 174  
 csx\_RegisterClient — register client, 175  
 csx\_ReleaseConfiguration — release configuration on PC Card, 178  
 csx\_ReleaseIO — release I/O resources, 188  
 csx\_ReleaseIRQ — release IRQ resource, 193  
 csx\_ReleaseSocketMask — clear client event mask, 196  
 csx\_ReleaseWindow — release window resources, 198  
 csx\_RepGet16 — read repetitively from device register, 180  
 csx\_RepGet32 — read repetitively from device register, 180  
 csx\_RepGet64 — read repetitively from device register, 180  
 csx\_RepGet8 — read repetitively from device register, 180  
 csx\_RepPut16 — write repetitively to device register, 182  
 csx\_RepPut32 — write repetitively to device register, 182  
 csx\_RepPut64 — write repetitively to device register, 182  
 csx\_RepPut8 — write repetitively to device register, 182  
 csx\_RequestConfiguration — configure PC Card and socket, 184  
 csx\_RequestIO — request I/O resources, 188  
 csx\_RequestIRQ — request IRQ resource, 193  
 csx\_RequestSocketMask — request client event mask, 196  
 csx\_RequestWindow — request window resources, 198  
 csx\_ResetFunction — reset a function on a PC card, 203  
 csx\_SetEventMask — set client event mask, 204  
 csx\_SetHandleOffset — set current access handle offset, 206  
 csx\_ValidateCIS — validate Card Information Structure (CIS), 207  
 current thread, get id of, 307

## D

datamsg — test whether a message is a data message, 208  
 DDI access credential structure, —  
   ddi\_get\_cred, 302  
 ddi\_add\_intr — add an interrupt handler, 209  
 ddi\_add\_softintr — add a soft interrupt, 213  
 DDI announce a device, — ddi\_report\_dev, 387  
 ddi\_binding\_name — return driver binding name, 219  
 ddi\_check\_acc\_handle, 221, 303  
 ddi\_check\_dma\_handle, 221  
 ddi\_create\_minor\_node — create a minor node for this device, 229

`ddi_dev_is_needed` — inform the system that a device's component is required, 238  
`ddi_dev_report_fault`, 244  
 DDI device access, slave access only —  
     `ddi_slaveonly`, 394  
`ddi_device_copy` — copy data from one device register to another device register, 231  
 DDI device critical region of control  
     enter — `ddi_enter_critical`, 298  
     exit — `ddi_exit_critical`, 298  
 DDI device information structure  
     find parent — `ddi_get_parent`, 309  
     get the root of the `dev_info` tree —  
         `ddi_root_node`, 390  
     remove a minor node for this `devinfo` —  
         `ddi_remove_minor_node`, 382  
 DDI device instance number, get —  
     `ddi_get_instance`, 306  
 DDI device mapping  
     `ddi_mapdev` — create driver-controlled mapping of device, 328  
     `ddi_mapdev_intercept` — control driver notification of user accesses, 331  
     `ddi_mapdev_nointercept` — control driver notification of user accesses, 331  
     `devmap_default_access` — device mapping access entry point, 408  
 DDI device registers  
     map — `ddi_map_regs`, 335  
     return the number of register sets —  
         `ddi_dev_nregs`, 242  
     return the size — `ddi_dev_regsizesize`, 243  
     unmap — `ddi_unmap_regs`, 335  
 DDI device's private data area  
     get the address —  
         `ddi_get_driver_private`, 304  
     set the address —  
         `ddi_set_driver_private`, 304  
 DDI device virtual address  
     read 16 bit — `ddi_peek16`, 352  
     read 32 bit — `ddi_peek32`, 352  
     read 64 bit — `ddi_peek64`, 352  
     read 8 bit — `ddi_peek8`, 352  
     read a value — `ddi_peek`, 352  
     write 16 bit — `ddi_poke16`, 354  
     write 32 bit — `ddi_poke32`, 354  
     write 64 bit — `ddi_poke64`, 354  
     write 8 bit — `ddi_poke8`, 354  
 DDI device virtual address (Continued)  
     write a value — `ddi_poke`, 354  
`ddi_device_zero` — zero fill the device register, 233  
`ddi_devid_compare` — Kernel interfaces for device ids, 235  
`ddi_devid_free` — Kernel interfaces for device ids, 235  
`ddi_devid_init` — Kernel interfaces for device ids, 235  
`ddi_devid_register` — Kernel interfaces for device ids, 235  
`ddi_devid_sizeof` — Kernel interfaces for device ids, 235  
`ddi_devid_unregister` — Kernel interfaces for device ids, 235  
`ddi_devid_valid` — Kernel interfaces for device ids, 235  
 DDI `devinfo` node name  
     return — `ddi_binding_name`, 219  
     return — `ddi_get_name`, 219  
     return — `ddi_node_name`, 350  
 DDI direct memory access, convert DMA handle to DMA addressing cookie —  
     `ddi_dma_htoc`, 274  
 DDI direct memory access services  
     allocate consistent memory—  
         `ddi_iopb_alloc`, 337  
     convert a DMA cookie — `ddi_dma_coff`, 262  
     easier DMA setup —  
         `ddi_dma_addr_setup`, 251  
     easier DMA setup —  
         `ddi_dma_buf_setup`, 259  
     find minimum alignment and transfer size for device — `ddi_iomin`, 315  
     find post DMA mapping alignment and minimum effect properties —  
         `ddi_dma_devalign`, 264  
     free consistent memory —  
         `ddi_iopb_free`, 337  
     report current DMA window offset and size —  
         `ddi_dma_curwin`, 263  
     setup DMA mapping —  
         `ddi_dma_setup`, 283, 285, 288  
     setup DMA resources —  
         `ddi_dma_setup`, 291  
     shift current DMA window —  
         `ddi_dma_movwin`, 279

DDI direct memory access services (Continued)

- tear down DMA mapping —  
  - ddi\_dma\_free, 269
- ddi\_dma\_addr\_bind\_handle — binds an address to a DMA handle, 248
- ddi\_dma\_alloc\_handle — allocate DMA handle, 253
- ddi\_dma\_buf\_bind\_handle — binds a system buffer to a DMA handle, 256
- ddi\_dma\_burstsizes — find out the allowed burst sizes for a DMA mapping, 261
- ddi\_dma\_free\_handle — free DMA handle, 270
- ddi\_dma\_get\_attr, 271
- ddi\_dma\_getwin — activate a new DMA window, 272
- ddi\_dma\_mem\_alloc — allocate memory for DMA transfer, 276
- ddi\_dma\_mem\_free — free previously allocated memory, 278
- ddi\_dma\_nextcookie — retrieve subsequent DMA cookie, 281
- ddi\_dma\_nextseg — get next DMA segment, 283
- ddi\_dma\_nextwin — get next DMA window, 285
- ddi\_dma\_numwin — retrieve number of DMA windows, 287
- ddi\_dma\_segtocookie — convert a DMA segment to a DMA address cookie, 288
- ddi\_dma\_set\_sbus64 — allow 64 bit transfers on SBUS, 290
- ddi\_dma\_sync — synchronize CPU and I/O views of memory, 293
- ddi\_dma\_unbind\_handle — unbinds the address in a DMA handle, 295
- ddi\_dmae — system DMA engine functions, 266
- ddi\_dmae\_1stparty — system DMA engine functions, 266
- ddi\_dmae\_alloc — system DMA engine functions, 266
- ddi\_dmae\_disable — system DMA engine functions, 266
- ddi\_dmae\_enable — system DMA engine functions, 266
- ddi\_dmae\_getattr — system DMA engine functions, 266
- ddi\_dmae\_getcnt — system DMA engine functions, 266
- ddi\_dmae\_getlim — system DMA engine functions, 266
- ddi\_dmae\_prog — system DMA engine functions, 266
- ddi\_dmae\_release — system DMA engine functions, 266
- ddi\_dmae\_stop — system DMA engine functions, 266
- ddi\_driver\_major, 296
- ddi\_driver\_name — return normalized driver name, 297
- ddi\_ffs — find first (last) bit set in a long integer, 299
- ddi\_flb — find first (last) bit set in a long integer, 299
- ddi\_get\_iblock\_cookie — get interrupt block cookie, 209
- ddi\_get\_kt\_did, 307
- ddi\_get\_lbolt, returns the value of lbolt, 308
- ddi\_get\_name — return driver binding name, 219
- ddi\_get\_pid, returns the process ID, 310
- ddi\_get\_soft\_iblock\_cookie — get soft interrupt block cookie, 213
- ddi\_get\_time, returns the current time in seconds, 311
- ddi\_get16 — read data from the device, 300
- ddi\_get32 — read data from the device, 300
- ddi\_get64 — read data from the device, 300
- ddi\_get8 — read data from the device, 300
- ddi\_getimajor, display a SCSI request sense message, 305
- ddi\_in\_panic — determine if system is in panic state, 312
- DDI information — csx\_CS\_DDI\_Info, 96
- DDI interrupt handling
  - add an interrupt — ddi\_add\_intr, 209
  - get interrupt block cookie —  
    - ddi\_get\_iblock\_cookie, 209
  - indicate interrupt handler type —  
    - ddi\_intr\_hilevel, 313
  - remove an interrupt — ddi\_remove\_intr, 209
  - return the number of interrupt specifications —  
    - ddi\_dev\_nintrs, 241
- ddi\_io\_get16 — read data from the mapped device register in I/O space, 314

`ddi_io_get32` — read data from the mapped device register in I/O space, 314  
`ddi_io_get8` — read data from the mapped device register in I/O space, 314  
`ddi_io_getb` — read data from the mapped device register in I/O space, 314  
`ddi_io_getl` — read data from the mapped device register in I/O space, 314  
`ddi_io_getw` — read data from the mapped device register in I/O space, 314  
`ddi_io_put16` — write data to the mapped device register in I/O space, 318  
`ddi_io_put32` — write data to the mapped device register in I/O space, 318  
`ddi_io_put8` — write data to the mapped device register in I/O space, 318  
`ddi_io_putb` — write data to the mapped device register in I/O space, 318  
`ddi_io_putl` — write data to the mapped device register in I/O space, 318  
`ddi_io_putw` — write data to the mapped device register in I/O space, 318  
`ddi_io_rep_get16` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_get32` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_get8` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_getb` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_getl` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_getw` — read multiple data from the mapped device register in I/O space, 320  
`ddi_io_rep_put16` — write multiple data to the mapped device register in I/O space, 322  
`ddi_io_rep_put32` — write multiple data to the mapped device register in I/O space, 322  
`ddi_io_rep_put8` — write multiple data to the mapped device register in I/O space, 322  
`ddi_io_rep_putb` — write multiple data to the mapped device register in I/O space, 322  
`ddi_io_rep_putl` — write multiple data to the mapped device register in I/O space, 322  
`ddi_io_rep_putw` — write multiple data to the mapped device register in I/O space, 322  
`ddi_iopb_alloc` — allocate and free non-sequentially accessed memory, 316  
`ddi_iopb_free` — allocate and free non-sequentially accessed memory, 316  
`ddi_log_sysevent`, 324  
`ddi_mapdev` — create driver-controlled mapping of device, 328  
`ddi_mapdev_intercept` — control driver notification of user accesses, 331  
`ddi_mapdev_set_device_acc_attr` — Set the device attributes for the mapping, 333  
`ddi_mem_get16` — read data from mapped device in the memory space or allocated DMA memory, 339  
`ddi_mem_get32` — read data from mapped device in the memory space or allocated DMA memory, 339  
`ddi_mem_get64` — read data from mapped device in the memory space or allocated DMA memory, 339  
`ddi_mem_put16` — write data to mapped device in the memory space or allocated DMA memory, 340  
`ddi_mem_put32` — write data to mapped device in the memory space or allocated DMA memory, 340  
`ddi_mem_put64` — write data to mapped device in the memory space or allocated DMA memory, 340  
`ddi_mem_rep_get16` — read data from mapped device in the memory space or allocated DMA memory, 342  
`ddi_mem_rep_get32` — read data from mapped device in the memory space or allocated DMA memory, 342  
`ddi_mem_rep_get64` — read data from mapped device in the memory space or allocated DMA memory, 342  
`ddi_mem_rep_get8` — read data from mapped device in the memory space or allocated DMA memory, 342  
`ddi_mem_rep_put16` — write data to mapped device in the memory space or allocated DMA memory, 344  
`ddi_mem_rep_put32` — write data to mapped device in the memory space or allocated DMA memory, 344  
`ddi_mem_rep_put64` — write data to mapped device in the memory space or allocated DMA memory, 344

`ddi_mem_rep_put8` — write data to mapped device in the memory space or allocated DMA memory, 344  
 DDI memory mapping  
   map a segment — `ddi_segmap`, 392  
   map a segment — `devmap_setup`, 418  
`ddi_mmap_get_model` — return data model type of current thread, 346  
`ddi_model_convert_from` — determine data model type mismatch, 348  
`ddi_model_convert_from` — Determine if there is a need to translate shared data structure contents, 348  
`ddi_no_info` — returns `DDI_FAILURE`, a convenience for drivers implementing DLPI style 2 services, 351  
`ddi_node_name` — return the devinfo node name, 350  
 DDI page size conversions  
   — `ddi_btop`, 220  
   — `ddi_btopr`, 220  
   — `ddi_ptob`, 220  
`ddi_prop_exists` — check for the existence of a property, 360  
`ddi_prop_get_int` — look up integer property, 362  
`ddi_prop_lookup` — lookup property information, 365  
`ddi_prop_lookup_byte_array` — lookup property information, 365  
`ddi_prop_lookup_int_array` — lookup property information, 365  
`ddi_prop_lookup_string` — lookup property information, 365  
`ddi_prop_lookup_string_array` — lookup property information, 365  
`ddi_prop_update` — update property information., 374  
`ddi_prop_update_byte_array` — update property information., 374  
`ddi_prop_update_int` — update property information., 374  
`ddi_prop_update_int_array` — update property information., 374  
`ddi_prop_update_string` — update property information., 374  
`ddi_prop_update_string_array` — update property information., 374  
 DDI property management  
   create properties for leaf device drivers —  
     `ddi_prop_create`, 357  
   — `ddi_getlongprop`, 370  
   — `ddi_getlongprop_buf`, 370  
   — `ddi_getprop`, 370  
   — `ddi_getpropflen`, 370  
   — `ddi_prop_op`, 370  
   modify properties for leaf device drivers —  
     `ddi_prop_modify`, 357  
   remove all properties for leaf device drivers —  
     `ddi_prop_remove_all`, 357  
   remove properties for leaf device drivers —  
     `ddi_prop_remove`, 357  
   remove properties for leaf device drivers —  
     `ddi_prop_undefine`, 357  
`ddi_put16` — write data to the device, 377  
`ddi_put32` — write data to the device, 377  
`ddi_put64` — write data to the device, 377  
`ddi_put8` — write data to the device, 377  
`ddi_regs_map_free` — free mapped register address space, 379  
`ddi_regs_map_setup` — set up a mapping for a register address space, 380  
`ddi_remove_intr` — remove an interrupt handler, 209  
`ddi_remove_softintr` — remove a soft interrupt, 213  
`ddi_removing_power`, 383  
`ddi_rep_get16` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_get32` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_get64` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_get8` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_getb` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_getl` — read data from the mapped memory address, device register or allocated DMA memory address, 385

`ddi_rep_getll` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_getw` — read data from the mapped memory address, device register or allocated DMA memory address, 385  
`ddi_rep_put16` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_put32` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_put64` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_put8` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_putb` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_putl` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_putll` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
`ddi_rep_putw` — write data to the mapped memory address, device register or allocated DMA memory address, 388  
DDI self identifying devices, tell whether a device is self-identifying — `ddi_dev_is_sid`, 240  
DDI soft interrupt handling  
  add a soft interrupt — `ddi_add_softintr`, 213  
  get soft interrupt block cookie — `ddi_get_soft_iblock_cookie`, 213  
  remove a soft interrupt — `ddi_remove_softintr`, 213  
DDI soft state utility routines  
  allocate state structure — `ddi_soft_state_zalloc`, 395  
  free soft state entry — `ddi_soft_state_free`, 395  
  get pointer to soft state — `ddi_get_soft_state`, 395  
  initialize state — `ddi_soft_state_init`, 395  
DDI soft state utility routines (Continued)  
  remove all state info — `ddi_soft_state_fini`, 395  
DDI\_SUSPEND, 383  
`ddi_trigger_softintr` — trigger a soft interrupt, 213  
`ddi_umem_alloc` — allocate kernel memory, 400  
`ddi_umem_free` — allocate kernel memory, 400  
`ddi_umem_lock` — Locks and unlocks memory pages, 404  
`ddi_umem_zalloc` — allocate kernel memory, 400  
default SCSI HBA probe function — `scsi_hba_probe`, 597  
delay — delay process execution for a specified number of clock ticks, 406  
deregister client from Card Services list — `csx_DeregisterClient`, 98  
determine data model type mismatch — `ddi_model_convert_from`, 348  
Device Driver Interface, *See* DDI  
device mapping access entry point — `devmap_default_access`, 408  
device switch tables, return function for insignificant entries — `nulldev`, 495  
devices  
  get major device number — `getmajor`, 445  
  get minor device number — `getminor`, 446  
  make device number from major and minor numbers — `makedevice`, 478  
devices, non-pollable, error return function — `nochpoll`, 492  
`devmap_default_access` — device mapping access entry point, 408  
`devmap_devmem_setup` — Set driver memory mapping parameters, 411  
  `devmap_devmem_setup()`, 410  
  `devmap_umem_setup()`, 411  
`devmap_do_ctxmgt` — perform device context switching on a mapping, 413  
`devmap_load` — control the validation of memory address translations, 419  
`devmap_set_ctx_timeout` — set context management timeout value, 416  
`devmap_umem_setup` — Set driver memory mapping parameters, 411

devmap\_unload — control the validation of memory address translations, 419

disksort — single direction elevator seek sort for buffers, 421

display a SCSI request sense message, scsi\_vu\_errmsg, 622

DMA attribute structure, 271

DMA mapping, the allowed burst sizes for — ddi\_dma\_burstsizes, 261

driver buffers

- copy data— ddi\_copyin, 223
- copy data from driver — ddi\_copyout, 226
- copy data from driver to user program — copyout, 90
- copy data from user program — copyin, 86

driver error messages, display an error message or panic the system — cmn\_err, 76

driver privilege — drv\_priv, 425

drv\_getparm — retrieve kernel state information, 422

drv\_hztousec — convert clock ticks to microseconds, 424

drv\_priv — determine driver privilege, 425

drv\_usecstohz — convert microseconds to clock ticks, 426

drv\_usecwait — busy-wait for specified interval, 427

dupb — duplicate a message block descriptor, 428

duplicate a message — dupmsg, 431

duplicate a message block descriptor — dupb, 428

duplicate access handle — csx\_DupHandle, 99

dupmsg — duplicate a message, 431

## E

enable/disable accesses to the PCI Local Bus Configuration space.

- pci\_config\_setup, 512
- pci\_config\_teardown, 512

error return codes converted to text strings — csx\_Error2Text, 101

error return function for illegal entries — nodev, 493

event mask

- return client event mask — csx\_GetEventMask, 204
- set client event mask — csx\_SetEventMask, 204

events converted to text strings — csx\_Event2Text, 102

expression verification, — assert, 47

## F

find first (last) bit set in a long integer —

- ddi\_ffs, 299
- ddi\_fls, 299

first CIS tuple — csx\_GetFirstTuple, 107

flushband — flush messages for specified priority band, 436

free access handle — csx\_FreeHandle, 103

free DMA handle, —

- ddi\_dma\_free\_handle, 270

free mapped register address space —

- ddi\_regs\_map\_free, 379

free previously allocated memory —

- ddi\_dma\_mem\_free, 278

free space — rmfree, 567

freerbuf — free a raw buffer header, 441

freeze, thaw the state of a stream —

- freezestr, 442
- unfreezestr, 442

freezestr — freeze, thaw the state of a stream, 442

## G

generic tuple parser — csx\_ParseTuple, 172

get interrupt block cookie, —

- ddi\_get\_iblock\_cookie, 209

get kernel internal minor number from an external dev\_t, scsi\_vu\_errmsg, 305

get next DMA segment —

- ddi\_dma\_nextseg, 283

get next DMA window —

- ddi\_dma\_nextwin, 285

get soft interrupt block cookie, —

- ddi\_get\_soft\_iblock\_cookie, 213

gethrtime, 444

getmajor — get major device number, 445  
 getminor — get minor device number, 446  
 getrbuf — get a raw buffer header, 450  
 gld\_mac\_alloc — allocate a GLD mac\_info structure, 451  
 gld\_mac\_free — free a GLD mac\_info structure, 451  
 gld\_rcv — pass the inbound packet upstream, 452  
 gld\_register — link the driver with the GLD framework, 451  
 gld\_sched — reschedule stalled outbound packets, 452  
 gld\_unregister — unlink the driver from the GLD framework, 452

## H

handle variable argument list  
   — va\_arg, 655  
   — va\_copy, 655  
   — va\_end, 655  
   — va\_start, 655  
 high resolution time, 444

## I

I/O, block, suspend processes pending completion — biowait, 64  
 I/O, buffer, release buffer and notify processes — biodone, 56  
 I/O, paged request  
   allocate virtual address space — bp\_mapin, 65  
   deallocate virtual address space — bp\_mapout, 66  
 I/O, physical  
   — minphys, 517  
   — physio, 517  
 I/O error, return — geterror, 443  
 I/O resources  
   release I/O resources — csx\_ReleaseIO, 188  
   request I/O resources — csx\_RequestIO, 188  
 id32\_alloc — 32-bit driver ID management routines, 455

id32\_free — 32-bit driver ID management routines, 455  
 id32\_lookup — 32-bit driver ID management routines, 455  
 inb — read from an I/O port, 456  
 inform the system that a device's component is required. — ddi\_dev\_is\_needed, 238  
 initialize a named kstat — kstat\_named\_init, 472  
 initialize a buffer structure — bioinit, 60  
 inl — read from an I/O port, 456  
 interrupt handling  
   add an interrupt — ddi\_add\_intr, 209  
   get interrupt block cookie — ddi\_get\_iblock\_cookie, 209  
   remove an interrupt — ddi\_remove\_intr, 209  
 inw — read from an I/O port, 456  
 IOC\_CONVERT\_FROM — Determine if there is a need to translate M\_IOCTL contents, 460  
 IRQ resource  
   release IRQ resource — csx\_ReleaseIRQ, 193  
   request IRQ resource — csx\_RequestIRQ, 193

## K

kernel memory cache allocator operations — kmem\_cache\_alloc, 463  
 kernel memory cache allocator operations — kmem\_cache\_create, 463  
 kernel memory cache allocator operations — kmem\_cache\_destroy, 463  
 kernel memory cache allocator operations — kmem\_cache\_free, 463  
 kernel address locations, between locations — bcopy, 51  
 kernel addresses, get page frame number — hat\_getkpfnum, 454  
 Kernel interfaces for device ids  
   — ddi\_devid\_compare, 235  
   — ddi\_devid\_free, 235  
   — ddi\_devid\_init, 235  
   — ddi\_devid\_register, 235  
   — ddi\_devid\_sizeof, 235  
   — ddi\_devid\_unregister, 235  
   — ddi\_devid\_valid, 235

kernel modules, dynamic loading  
 add loadable module — `mod_install`, 484  
 query loadable module — `mod_info`, 484  
 remove loadable module —  
   `mod_remove`, 484  
 kernel state information — `drv_getparm`, 422  
`kmem_alloc` — allocate kernel memory, 461  
`kmem_cache_alloc` — kernel memory cache  
 allocator operations, 463  
`kmem_cache_create` — kernel memory cache  
 allocator operations, 463  
`kmem_cache_destroy` — kernel memory cache  
 allocator operations, 463  
`kmem_cache_free` — kernel memory cache  
 allocator operations, 463  
`kmem_free` — allocate kernel memory, 461  
`kmem_zalloc` — allocate kernel memory, 461  
`kstat_create` — create and initialize a new  
 kstat, 469  
`kstat_delete` — remove a kstat from the  
 system, 470  
`kstat_install` — add a fully initialized kstat to  
 the system, 471  
`kstat_named_init` — initialize a named  
 kstat, 472  
`kstat_queue` — update I/O kstat statistics, 473  
`kstat_runq_back_to_waitq` — update I/O kstat  
 statistics, 473  
`kstat_runq_enter` — update I/O kstat  
 statistics, 473  
`kstat_runq_exit` — update I/O kstat  
 statistics, 473  
`kstat_waitq_enter` — update I/O kstat  
 statistics, 473  
`kstat_waitq_exit` — update I/O kstat  
 statistics, 473  
`kstat_waitq_to_runq` — update I/O kstat  
 statistics, 473

## L

`linkb` — concatenate two message blocks, 475  
 Locks and unlocks memory pages —  
   `ddi_umem_lock`, 404  
 look up integer property —  
   `ddi_prop_get_int`, 362

look up per-device-type scsi-options property  
   — `scsi_get_device_type_scsi_options`, 587  
 lookup property information  
   — `ddi_prop_lookup`, 365  
   — `ddi_prop_lookup_byte_array`, 365  
   — `ddi_prop_lookup_int_array`, 365  
   — `ddi_prop_lookup_string`, 365  
   — `ddi_prop_lookup_string_array`, 365

## M

major device number, 296  
`makedevice` — make device number from major  
 and minor numbers, 478  
 map memory area on PC Card —  
   `csx_MapMemPage`, 120  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_boolean`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_byte`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_byte_array`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int16`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int16_array`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int32`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int32_array`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int64`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_int64_array`, 500  
 match name and type indicated by the interface  
 name and retrieve data value —  
   `nvlist_lookup_string`, 500

match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_string\_array, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint16, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint16\_array, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint32, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint32\_array, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint64, 500  
 match name and type indicated by the interface name and retrieve data value —  
     nvlist\_lookup\_uint64\_array, 500  
 max — return the larger of two integers, 479  
 memory, clear for a given number of bytes —  
     bzero, 72  
 min — return the lesser of two integers, 480  
 minor node for device, create —  
     ddi\_create\_minor\_node, 229  
 modify PC Card configuration —  
     csx\_ModifyConfiguration, 121  
 modify window attributes —  
     csx\_ModifyWindow, 123  
 mt-streams — STREAMS multithreading, 487  
 mutex routines  
     — mutex, 489  
     — mutex\_destroy, 489  
     — mutex\_enter, 489  
     — mutex\_exit, 489  
     — mutex\_init, 489  
     — mutex\_owned, 489  
     — mutex\_tryenter, 489  
 mutual exclusion lock, *See* mutex

**N**  
 next CIS tuple — csx\_GetNextTuple, 107  
 nodes, create minor nodes for client —  
     csx\_MakeDeviceNode, 117  
 Notify pm framework of autonomous power level change — pm\_power\_has\_changed, 524  
 notify target driver of bus resets —  
     scsi\_reset\_notify, 615  
 nvlist\_lookup\_boolean — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_byte — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_byte\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int16 — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int16\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int32 — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int32\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int64 — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_int64\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_string — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_string\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_uint16 — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_uint16\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_uint32 — match name and type indicated by the interface name and retrieve data value, 500

nvlist\_lookup\_uint32\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_uint64 — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_lookup\_uint64\_array — match name and type indicated by the interface name and retrieve data value, 500  
 nvlist\_remove — remove name-value pairs, 504  
 nvlist\_remove\_all — remove name-value pairs, 504

## O

obtain DDI information —  
     csx\_CS\_DDI\_Info, 96  
 OTHERQ — get pointer to queue's partner queue, 507  
 outb — write to an I/O port, 508  
 outl — write to an I/O port, 508  
 outw — write to an I/O port, 508

## P

panic state — ddi\_in\_panic, 312  
 parse Battery Replacement Date tuple —  
     csx\_Parse\_CISTPL\_BATTERY, 125  
 parse Byte Order tuple —  
     csx\_Parse\_CISTPL\_BYTEORDER, 126  
 parse Card Configuration Table tuple —  
     csx\_Parse\_CISTPL\_CFTABLE\_ENTRY, 128  
 parse Card Initialization Date tuple —  
     csx\_Parse\_CISTPL\_DATE, 137  
 parse Configuration tuple —  
     csx\_Parse\_CISTPL\_CONFIG, 134  
 parse Data Organization tuple —  
     csx\_Parse\_CISTPL\_ORG, 166  
 parse Data Recording Format tuple —  
     csx\_Parse\_CISTPL\_FORMAT, 145  
 parse Device Geo A tuple —  
     csx\_Parse\_CISTPL\_DEVICEGEO\_A, 143  
 parse Device Geo tuple —  
     csx\_Parse\_CISTPL\_DEVICEGEO, 141

parse Device Information tuple  
     for Attribute Memory —  
         csx\_Parse\_CISTPL\_DEVICE\_A, 138  
     for Common Memory —  
         csx\_Parse\_CISTPL\_DEVICE, 138  
 parse Function Extension tuple —  
     csx\_Parse\_CISTPL\_FUNCE, 147  
 parse Function Identification tuple —  
     csx\_Parse\_CISTPL\_FUNCID, 155  
 parse Geometry tuple —  
     csx\_Parse\_CISTPL\_GEOMETRY, 157  
 parse JEDEC Identifier tuple  
     for Attribute Memory —  
         csx\_Parse\_CISTPL\_JEDEC\_A, 158  
     for Common Memory —  
         csx\_Parse\_CISTPL\_JEDEC\_C, 158  
 parse Level-1 Version/Product Information tuple —  
     csx\_Parse\_CISTPL\_VERS\_1, 170  
 parse Level-2 Version and Information tuple —  
     csx\_Parse\_CISTPL\_VERS\_2, 171  
 parse Link Target tuple —  
     csx\_Parse\_CISTPL\_LINKTARGET, 160  
 parse Long Link A tuple, —  
     csx\_Parse\_CISTPL\_LONGLINK\_A, 161  
 parse Long Link C tuple, —  
     csx\_Parse\_CISTPL\_LONGLINK\_C, 161  
 parse Manufacturer Identification tuple —  
     csx\_Parse\_CISTPL\_MANFID, 165  
 parse Multi-Function tuple —  
     csx\_Parse\_CISTPL\_LONGLINK\_MFC, 163  
 parse Other Condition Device Information tuple  
     for Attribute Memory —  
         csx\_Parse\_CISTPL\_DEVICE\_OA, 138  
     for Common Memory —  
         csx\_Parse\_CISTPL\_DEVICE\_OC, 138  
 parse Software Interleaving tuple —  
     csx\_Parse\_CISTPL\_SWIL, 169  
 parse Special Purpose tuple —  
     csx\_Parse\_CISTPL\_SPCL, 167  
 parser, for tuples (generic) —  
     csx\_ParseTuple, 172  
 pci\_config\_get16 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510  
 pci\_config\_get32 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_get64 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_get8 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_getb — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_getl — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_getll — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_getw — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_put16 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_put32 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_put64 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_put8 — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_putb — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_putl — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_putll — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_putw — read or write single datum of various sizes to the PCI Local Bus Configuration space, 510

pci\_config\_setup — enable/disable accesses to the PCI Local Bus Configuration space., 512

pci\_config\_teardown — enable/disable accesses to the PCI Local Bus Configuration space., 512

pci\_report\_pmcap—Report power management capability of a PCI device, 513

perform device context switching on a mapping — devmap\_do\_ctxmgt, 413

pm\_busy\_component — control device components' availability for Power Management, 519

pm\_idle\_component — control device components' availability for Power Management, 519

pm\_power\_has\_changed – Notify pm framework of autonomous power level change, 524

pm\_raise\_power – Raise or lower power of components, 526

pm\_trans\_check – advisory check for device power cycles, 529

pollwakeup — inform a process that an event has occurred, 531

proc\_ref — send a signal to a process, 532

proc\_signal — send a signal to a process, 532

proc\_unref — send a signal to a process, 532

put — call a STREAMS put procedure, 537

## Q

qassociate — associate STREAMS queue with bottom driver in queue, 547

qbufcall — call a function when a buffer becomes available, 549

qtimeout — execute a function after a specified length of time, 555

qunbufcall — cancel a pending qbufcall request, 556

quntimeout — cancel previous timeout function call, 557

qwait — STREAMS wait routines, 558

qwait\_sig — STREAMS wait routines, 558

qwriter — asynchronous STREAMS perimeter upgrade, 560

## R

Raise or lower power of components – pm\_raise\_power, 526

- raw buffer
  - free a raw buffer header — `freerbuf`, 441
  - get a raw buffer header — `getrbuf`, 450
- RD — get pointer to the read queue, 561
- read from an I/O port — `inb`, 456
- read from an I/O port — `inl`, `inl`, 456
- read from an I/O port — `inw`, `inw`, 456
- read from an I/O port — `repinsb`, `repinsb`, 456
- read from an I/O port — `repinsd`, `repinsd`, 456
- read from an I/O port — `repinsw`, `repinsw`, 456
- read data from mapped device in the memory space or allocated DMA memory
  - `ddi_mem_get16`, 339
  - `ddi_mem_get32`, 339
  - `ddi_mem_get64`, 339
  - `ddi_mem_get8`, 339
  - `ddi_mem_rep_get16`, 342
  - `ddi_mem_rep_get32`, 342
  - `ddi_mem_rep_get64`, 342
  - `ddi_mem_rep_get8`, 342
- read data from the device
  - `ddi_get16`, 300
  - `ddi_get32`, 300
  - `ddi_get64`, 300
  - `ddi_get8`, 300
- read data from the mapped device register in I/O space
  - `ddi_io_get16`, 314
  - `ddi_io_get32`, 314
  - `ddi_io_get8`, 314
  - `ddi_io_getb`, 314
  - `ddi_io_getl`, 314
  - `ddi_io_getw`, 314
- read data from the mapped memory address, device register or allocated DMA memory address
  - `ddi_rep_get16`, 385
  - `ddi_rep_get32`, 385
  - `ddi_rep_get64`, 385
  - `ddi_rep_get8`, 385
  - `ddi_rep_getb`, 385
  - `ddi_rep_getl`, 385
  - `ddi_rep_getll`, 385
  - `ddi_rep_getw`, 385
- read from device register
  - `csx_Get16`, 104
  - `csx_Get32`, 104
  - `csx_Get64`, 104
- read from device register (Continued)
  - `csx_Get8`, 104
- read multiple data from the mapped device register in I/O space
  - `ddi_io_rep_get16`, 320
  - `ddi_io_rep_get32`, 320
  - `ddi_io_rep_get8`, 320
  - `ddi_io_rep_getb`, 320
  - `ddi_io_rep_getl`, 320
  - `ddi_io_rep_getw`, 320
- read or write a PC Card Configuration Register
  - `csx_AccessConfigurationRegister`, 92
- read or write single datum of various sizes to the PCI Local Bus Configuration space
  - `pci_config_get16`, 510
  - `pci_config_get32`, 510
  - `pci_config_get64`, 510
  - `pci_config_get8`, 510
  - `pci_config_getb`, 510
  - `pci_config_getl`, 510
  - `pci_config_getll`, 510
  - `pci_config_getw`, 510
  - `pci_config_put16`, 510
  - `pci_config_put32`, 510
  - `pci_config_put64`, 510
  - `pci_config_put8`, 510
  - `pci_config_putb`, 510
  - `pci_config_putl`, 510
  - `pci_config_putll`, 510
  - `pci_config_putw`, 510
- read repetitively from device register
  - `csx_RepGet16`, 180
  - `csx_RepGet32`, 180
  - `csx_RepGet64`, 180
  - `csx_RepGet8`, 180
- readers/writer lock functions
  - `rw_destroy`, 571
  - `rw_downgrade`, 571
  - `rw_enter`, 571
  - `rw_exit`, 571
  - `rw_init`, 571
  - `rw_read_locked`, 571
  - `rw_tryenter`, 571
  - `rw_tryupgrade`, 571
  - `rwlock`, 571
- register client — `csx_RegisterClient`, 175
- release client event mask — `csx_ReleaseSocketMask`, 196

- release I/O resources — `csx_ReleaseIO`, 188
- release IRQ resource — `csx_ReleaseIRQ`, 193
- release window resources —
  - `csx_ReleaseWindow`, 198
- release configuration on PC Card —
  - `csx_ReleaseConfiguration`, 178
- remove name-value pairs —
  - `nvlist_remove`, 504
- remove name-value pairs —
  - `nvlist_remove_all`, 504
- remove a kstat from the system —
  - `kstat_delete`, 470
- remove a soft interrupt, —
  - `ddi_remove_softintr`, 213
- remove an interrupt handler, —
  - `ddi_remove_intr`, 209
- remove client from Card Services list —
  - `csx_DeregisterClient`, 98
- `repinsb` — read from an I/O port, 456
- `repinsd` — read from an I/O port, 456
- `repinsw` — read from an I/O port, 456
- Report a hardware failure, 244
- Report power management capability of a PCI
  - `device-pci_report_pmcap`, 513
- `repoutsb` — write to an I/O port, 508
- `repoutsd` — write to an I/O port, 508
- `repoutsw` — write to an I/O port, 508
- request client event mask —
  - `csx_RequestSocketMask`, 196
- request I/O resources — `csx_RequestIO`, 188
- request IRQ resource — `csx_RequestIRQ`, 193
- request window resources —
  - `csx_RequestWindow`, 198
- reset a function on a PC card —
  - `csx_ResetFunction`, 203
- resource map
  - allocate resource maps — `rmallocmap`, 565
  - free resource maps — `rmallocmap`, 565
- retrieve number of DMA windows —
  - `ddi_dma_numwin`, 287
- retrieve subsequent DMA cookie —
  - `ddi_dma_nextcookie`, 281
- return client event mask —
  - `csx_GetEventMask`, 204
- return client
  - `csx_GetFirstClient`, 105
  - `csx_GetNextClient`, 105

- return current access handle offset —
  - `csx_GetHandleOffset`, 109
- return data model type of current thread —
  - `ddi_mmap_get_model`, 346
- return data portion of tuple —
  - `csx_GetTupleData`, 115
- return driver binding name
  - `ddi_binding_name`, 219
  - `ddi_get_name`, 219
- return index matching capability string —
  - `scsi_hba_lookup_capstr`, 593
- return normalized driver name —
  - `ddi_driver_name`, 297
- return physical socket number —
  - `csx_MapLogSocket`, 119
- return status of PC Card and socket —
  - `csx_GetStatus`, 111
- return the devinfo node name —
  - `ddi_node_name`, 350
- return the larger of two integers — `max`, 479
- return the lesser of two integers — `min`, 480
- return tuple
  - first CIS tuple — `csx_GetFirstTuple`, 107
  - next CIS tuple — `csx_GetNextTuple`, 107
- returns `DDI_FAILURE`, a convenience for
  - drivers implementing DLPI style 2 services
  - `ddi_no_info`, 351
- returns size of a buffer structure — `biosize`, 63
- returns the current time in seconds,
  - `ddi_get_time`, 311
- returns the process ID, `ddi_get_pid`, 310
- returns the value of `lbolt`, returns the value of
  - `lbolt`, 308
- `rmalloc` — allocate space from a resource
  - map, 562
- `rmalloc_wait` — allocate space from a resource
  - map, 566
- `rmfree` — free space back into a resource
  - map, 567

## S

- `SAMESTR` — test if next queue is in the same
  - stream, 574
- SCSI Host Bus Adapter system initialization
  - and completion routines
  - `scsi_hba_init`, 592

scsi\_abort — abort a SCSI command, 575  
 scsi\_alloc\_consistent\_buf — scsi dma utility for allocating an I/O buffer for SCSI DMA, 576  
 scsi\_cname — decode SCSI commands, 578  
 SCSI commands, make packet  
   — makecom, 476  
   — makecom\_g0, 476  
   — makecom\_g0\_s, 476  
   — makecom\_g1, 476  
   — makecom\_g5, 476  
 scsi\_destroy\_pkt — free an allocated SCSI packet and its DMA resource, 580  
 SCSI dma utility routines  
   — scsi\_dmafree, 581  
   — scsi\_dmaget, 581  
 scsi\_dname — decode SCSI peripheral device type, 578  
 scsi\_errmsg — display a SCSI request sense message, 583  
 scsi\_free\_consistent\_buf — free a previously allocated SCSI DMA I/O buffer, 586  
 scsi\_get\_device\_type\_scsi\_options — look up per-device-type scsi-options property, 587  
 scsi\_hba\_attach — SCSI HBA attach and detach routines, 589  
 SCSI HBA attach and detach routines  
   — scsi\_hba\_attach, 589  
   — scsi\_hba\_attach\_setup, 589  
   — scsi\_hba\_detach, 589  
 scsi\_hba\_attach\_setup — SCSI HBA attach and detach routines, 589  
 scsi\_hba\_detach — SCSI HBA attach and detach routines, 589  
 scsi\_hba\_fini — SCSI Host Bus Adapter system completion routines, 592  
 scsi\_hba\_init — SCSI Host Bus Adapter system initialization routines, 592  
 scsi\_hba\_lookup\_capstr — return index matching capability string, 593  
 scsi\_hba\_pkt\_alloc — allocate and free a scsi\_pkt structure, 595  
 scsi\_hba\_pkt\_free — allocate and free a scsi\_pkt structure, 595  
 scsi\_hba\_probe — default SCSI HBA probe function, 597  
 scsi\_hba\_tran\_alloc — allocate and free transport structures, 598  
 scsi\_hba\_tran\_free — allocate and free transport structures, 598  
 scsi\_ifgetcap — get SCSI transport capability, 599  
 scsi\_ifsetcap — set SCSI transport capability, 599  
 scsi\_init\_pkt — prepare a complete SCSI packet, 603  
 scsi\_log — display a SCSI-device-related message, 607  
 scsi\_mname — decode SCSI messages, 578  
 SCSI packet  
   allocate a SCSI packet in iopb map —  
     get\_pktiopb, 447  
   free a packet in iopb map —  
     free\_pktiopb, 447  
   free an allocated SCSI packet and its DMA resource — scsi\_destroy\_pkt, 580  
 SCSI packet utility routines  
   — scsi\_pktalloc, 609  
   — scsi\_pktfree, 609  
   — scsi\_realloc, 609  
   — scsi\_resfree, 609  
 scsi\_poll — run a polled SCSI command on behalf of a target driver, 611  
 scsi\_probe — utility for probing a scsi device, 612  
 scsi\_reset — reset a SCSI bus or target, 614  
 scsi\_reset\_notify — notify target driver of bus resets, 615  
 scsi\_rname — decode SCSI packet completion reasons, 578  
 scsi\_setup\_cdb — setup SCSI command descriptor block (CDB), 616  
 scsi\_slave — utility for SCSI target drivers to establish the presence of a target, 617  
 scsi\_sname — decode SCSI sense keys, 578  
 scsi\_sync\_pkt — synchronize CPU and I/O views of memory, 619  
 scsi\_transport — request by a target driver to start a SCSI command, 620  
 scsi\_unprobe — free resources allocated during initial probing, 621  
 scsi\_unslave — free resources allocated during initial probing, 621  
 scsi\_vu\_errmsg, display a SCSI request sense message, 622

- semaphore functions
  - sema\_destroy, 625
  - sema\_init, 625
  - sema\_p, 625
  - sema\_p\_sig, 625
  - sema\_try, 625
  - sema\_v, 625
  - semaphore, 625
- send a signal to a process
  - proc\_ref, 532
  - proc\_signal, 532
  - proc\_unref, 532
- set client event mask —
  - csx\_RequestSocketMask, 196
- set client event mask — csx\_SetEventMask, 204
- set current access handle offset —
  - csx\_SetHandleOffset, 206
- Set driver memory mapping parameters
  - devmap\_devmem\_setup, 411
  - devmap\_umem\_setup, 411
- Set the device attributes for the mapping —
  - ddi\_mapdev\_set\_device\_acc\_attr, 333
- set up a mapping for a register address space —
  - ddi\_regs\_map\_setup, 380
- setup SCSI command descriptor block (CDB) —
  - scsi\_setup\_cdb, 616
- single direction elevator seek sort for buffers —
  - disksort, 421
- size in bytes
  - convert size in pages — ptob, 534
  - convert to size in memory pages (round down) — btop, 67
  - convert to size in memory pages (round up) — btopr, 68
- socket number, return physical socket number
  - csx\_MapLogSocket, 119
- soft interrupt handling
  - add a soft interrupt — ddi\_add\_softintr, 213
  - get soft interrupt block cookie —
    - ddi\_get\_soft\_iblock\_cookie, 213
  - remove a soft interrupt —
    - ddi\_remove\_softintr, 213
  - trigger a soft interrupt —
    - ddi\_trigger\_softintr, 213
- sprintf — format characters in memory, 627
- status of PC Card and socket —
  - csx\_GetStatus, 111
- STREAMS wait routines — qwait,
  - qwait\_sig, 558
- STREAMS ioctl blocks, allocate — mkiocb, 481
- STREAMS message blocks
  - attach a user-supplied data buffer in place —
    - esballoc, 433
  - call a function when a buffer becomes available — bufcall, 69, 649
  - call a function when a buffer becomes available — qbufcall, 549, 556
  - call function when buffer is available —
    - esbcall, 435
  - concatenate bytes in a message —
    - msgpullup, 486
  - concatenate bytes in a message —
    - pullupmsg, 535
  - concatenate two — linkb, 475
  - copy — copyb, 84
  - erase the contents of a buffer — clrbuf, 75
  - free all message blocks in a message —
    - freemsg, 440
  - free one — freeb, 439
  - remove from head of message —
    - unlinkb, 650
  - remove one from a message — rmvb, 568
- STREAMS message queue, insert a message into a queue — insq, 458
- STREAMS message queues, 48
- STREAMS Message queues, get next message —
  - getq, 449
- STREAMS message queues
  - reschedule a queue for service —
    - enableok, 432
  - test for room — canputnext, 74
  - test for room — canput, 73
- STREAMS messages
  - copy a message — copymsg, 88
  - flush for specified priority band —
    - flushband, 436
  - remove from queue — flushq, 437
  - remove from queue — rmvq, 569
  - return the number of bytes in a message —
    - msgdsz, 485
  - submit messages to the log driver —
    - strlog, 634
  - test whether a message is a data message —
    - datamsg, 208
  - trim bytes — adjmsg, 70

STREAMS multithreading

- mt-streams, 487
- qbufcall — call a function when a buffer becomes available, 549
- qtimeout — execute a function after a specified length of time, 555
- qunbufcall — cancel a pending qbufcall request, 556
- qtimeout — cancel previous timeout function call, 557
- qwait, qwait\_sig — STREAMS wait routines, 558
- qwriter — asynchronous STREAMS perimeter upgrade, 560

STREAMS put and service procedures

- disable — qprocsoff, 551
- enable — qprocon, 551

STREAMS queues

- change information about a queue or band of the queue — strqset, 637
- enable a queue — qenable, 550
- get pointer to queue's partner queue — OTHERQ, 507
- get pointer to the read queue — RD, 561
- get information about a queue or band of the queue — strqget, 636
- number of messages on a queue — qsize, 554
- place a message at the head of a queue — putbq, 538
- prevent a queue from being scheduled — noenable, 494
- put a message on a queue — putq, 546
- send a control message to a queue — putctl, 540
- send a control message to a queue — putnextctl, 544
- send a control message with a one-byte parameter to a queue — putctl1, 539
- send a control message with a one-byte parameter to a queue — putnextctl1, 543
- send a message on a stream in the reverse direction — qreply, 552
- send a message to the next queue — putnext, 542
- test if next queue is in the same stream — SAMESTR, 574

STREAMS queues (Continued)

- test for flow control in specified priority band — bcanput, 49
- STREAMS write queues, get pointer for this module or driver — WR, 660
- STRUCT\_DECL, 32-bit application data access macros, 638
- swab — swap bytes in 16-bit halfwords, 643
- synchronize CPU and I/O views of memory — ddi\_dma\_sync, 293
- synchronize CPU and I/O views of memory — scsi\_sync\_pkt, 619
- system DMA engine functions
  - ddi\_dmae, 266
  - ddi\_dmae\_1stparty, 266
  - ddi\_dmae\_alloc, 266
  - ddi\_dmae\_disable, 266
  - ddi\_dmae\_enable, 266
  - ddi\_dmae\_getattr, 266
  - ddi\_dmae\_getcnt, 266
  - ddi\_dmae\_getlim, 266
  - ddi\_dmae\_prog, 266
  - ddi\_dmae\_release, 266
  - ddi\_dmae\_stop, 266
- system event, logging of, 324

## T

- testb — check for an available buffer, 644
- timeout — execute a function after a specified length of time, 646
- timeout, cancel previous timeout function call — untimeout, 651
- trigger a soft interrupt, — ddi\_trigger\_softintr, 213
- tuple
  - first CIS tuple — csx\_GetFirstTuple, 107
  - next CIS tuple — csx\_GetNextTuple, 107
  - return data portion of tuple — csx\_GetTupleData, 115
- tuple entry
  - generic tuple parser — csx\_ParseTuple, 172
  - parse Device Information tuple for Attribute Memory — csx\_Parse\_CISTPL\_DEVICE\_A, 138

tuple entry (Continued)

- parse Device Information tuple for Common Memory — `csx_Parse_CISTPL_DEVICE`, 138
- parse JEDEC Identifier tuple for Attribute Memory —
  - `csx_Parse_CISTPL_JEDEC_A`, 158
- parse JEDEC Identifier tuple for Common Memory —
  - `csx_Parse_CISTPL_JEDEC_C`, 158
- parse Long Link A tuple —
  - `csx_Parse_CISTPL_LONGLINK_A`, 161
- parse Long Link C tuple —
  - `csx_Parse_CISTPL_LONGLINK_C`, 161
- parse Other Condition Device Information tuple for Attribute Memory —
  - `csx_Parse_CISTPL_DEVICE_OA`, 138
- parse Other Condition Device Information tuple for Common Memory —
  - `csx_Parse_CISTPL_DEVICE_OC`, 138
- parse Battery Replacement Date tuple —
  - `csx_Parse_CISTPL_BATTERY`, 125
- parse Byte Order tuple —
  - `csx_Parse_CISTPL_BYTEORDER`, 126
- parse Card Configuration Table tuple —
  - `csx_Parse_CISTPL_CFTABLE_ENTRY`, 128
- parse Card Initialization Date tuple —
  - `csx_Parse_CISTPL_DATE`, 137
- parse Configuration tuple —
  - `csx_Parse_CISTPL_CONFIG`, 134
- parse Data Organization tuple —
  - `csx_Parse_CISTPL_ORG`, 166
- parse Data Recording Format tuple —
  - `csx_Parse_CISTPL_FORMAT`, 145
- parse Device Geo A tuple —
  - `csx_Parse_CISTPL_DEVICE_A`, 143
- parse Device Geo tuple —
  - `csx_Parse_CISTPL_DEVICEGEO`, 141
- parse Function Extension tuple —
  - `csx_Parse_CISTPL_FUNCE`, 147
- parse Function Identification tuple —
  - `csx_Parse_CISTPL_FUNCID`, 155
- parse Geometry tuple —
  - `csx_Parse_CISTPL_GEOMETRY`, 157
- parse Level-1 Version/Product Information tuple — `csx_Parse_CISTPL_VERS_1`, 170
- parse Level-2 Version and Information tuple — `csx_Parse_CISTPL_VERS_2`, 171

tuple entry (Continued)

- parse Link Target tuple —
  - `csx_Parse_CISTPL_LINKTARGET`, 160
- parse Manufacturer Identification tuple —
  - `csx_Parse_CISTPL_MANFID`, 165
- parse Multi-Function tuple —
  - `csx_Parse_CISTPL_LONGLINK_MFC`, 163
- parse Software Interleaving tuple —
  - `csx_Parse_CISTPL_SWIL`, 169
- parse Special Purpose tuple —
  - `csx_Parse_CISTPL_SPCL`, 167

## U

`uio` structure

- add character — `ureadc`, 653
- remove a character — `uwritec`, 654

`uiomove` — copy kernel data using `uio` structure, 648

unbinds the address in a DMA handle — `ddi_dma_unbind_handle`, 295

`unfreezestr` — freeze, thaw the state of a stream, 442

uninitialize a buffer structure — `biofini`, 59

update I/O `kstat` statistics

- `kstat_queue`, 473
- `kstat_runq_back_to_waitq`, 473
- `kstat_runq_enter`, 473
- `kstat_runq_exit`, 473
- `kstat_waitq_enter`, 473
- `kstat_waitq_exit`, 473
- `kstat_waitq_to_runq`, 473

update property information.

- `ddi_prop_update`, 374
- `ddi_prop_update_byte_array`, 374
- `ddi_prop_update_int`, 374
- `ddi_prop_update_int_array`, 374
- `ddi_prop_update_string`, 374
- `ddi_prop_update_string_array`, 374

## V

`va_arg` — handle variable argument list, 655

`va_copy` — handle variable argument list, 655

`va_end` — handle variable argument list, 655

`va_start` — handle variable argument list, 655

validate Card Information Structure (CIS) —  
  csx\_ValidateCIS, 207  
virtual address, return mapped virtual address  
  — csx\_GetMappedAddr, 110  
vsprintf — format characters in memory, 657

## W

window resources  
  release window resources —  
    csx\_ReleaseWindow, 198  
  request window resources —  
    csx\_RequestWindow, 198  
write data to mapped device in the memory  
  space or allocated DMA memory  
  — ddi\_mem\_put16, 340  
  — ddi\_mem\_put32, 340  
  — ddi\_mem\_put64, 340  
  — ddi\_mem\_put8, 340  
  — ddi\_mem\_rep\_put16, 344  
  — ddi\_mem\_rep\_put32, 344  
  — ddi\_mem\_rep\_put64, 344  
  — ddi\_mem\_rep\_put8, 344  
write data to the device  
  — ddi\_put16, 377  
  — ddi\_put32, 377  
  — ddi\_put64, 377  
  — ddi\_put8, 377  
write data to the mapped device register in I/O  
  space  
  — ddi\_io\_put16, 318  
  — ddi\_io\_put32, 318  
  — ddi\_io\_put8, 318  
  — ddi\_io\_putb, 318  
  — ddi\_io\_putl, 318  
  — ddi\_io\_putw, 318  
write data to the mapped memory address,  
  device register or allocated DMA memory  
  address  
  — ddi\_rep\_put16, 388  
  — ddi\_rep\_put32, 388  
  — ddi\_rep\_put64, 388  
  — ddi\_rep\_put8, 388  
  — ddi\_rep\_putb, 388  
  — ddi\_rep\_putl, 388  
  — ddi\_rep\_putll, 388  
  — ddi\_rep\_putw, 388

write multiple data to the mapped device  
  register in I/O space  
  — ddi\_io\_rep\_put16, 322  
  — ddi\_io\_rep\_put32, 322  
  — ddi\_io\_rep\_put8, 322  
  — ddi\_io\_rep\_putb, 322  
  — ddi\_io\_rep\_putl, 322  
  — ddi\_io\_rep\_putw, 322  
write or read a PC Card Configuration Register  
  — csx\_AccessConfigurationRegister, 92  
write repetitively to device register  
  — csx\_RepPut16, 182  
  — csx\_RepPut32, 182  
  — csx\_RepPut64, 182  
  — csx\_RepPut8, 182  
write to an I/O port  
  — outb, 508  
  — outl, 508  
  — outw, 508  
  — repoutsb, 508  
  — repoutsd, 508  
  — repoutsw, 508  
write to device register  
  — csx\_Put16, 174  
  — csx\_Put32, 174  
  — csx\_Put64, 174  
  — csx\_Put8, 174

## Z

zero fill the device register —  
  ddi\_device\_zero, 233