Preface

OVERVIEW

A man page is provided for both the naive user, and sophisticated user who is familiar with the SunOS operating system and is in need of on-line information. 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.

The following contains a brief description of each section in the man pages 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 of this volume.
• 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, etc.

• 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 operating systems 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 may include in a device driver.

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

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

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

NAME

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

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Literal characters (commands and options) are in bold font and variables (arguments, parameters and substitution characters) are in italic font. Options and
arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

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

... Ellipses. Several values may 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 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. The protocol specification pathname is always listed in bold font.

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, functions and such, are described under USAGE.

IOCTL

This section appears on pages in Section 7 only. Only the device class which supplies appropriate parameters to the ioctl(2) system call is called ioctl and generates its own heading. ioctl calls for a specific device are listed alphabetically (on the man page for that specific device). ioctl calls are used for a particular class of devices all of which have an io ending, such as mtio(7).
OPTIONS
This 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 as void do not return values, so they are not discussed in RETURN VALUES.

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

This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:

- Commands
- Modifiers
- Variables
- Expressions
- Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as

```
example%
```

or if the user must be super-user,

```
example#
```

Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.

ENVIRONMENT

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

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

**ATTRIBUTES**

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

**SEE ALSO**

This section lists references to other man pages, in-house documentation and outside publications.

**DIAGNOSTICS**

This section lists diagnostic messages with a brief explanation of the condition causing the error. Messages appear in **bold** font with the exception of variables, which are in *italic* font.

**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.
Section 9F describes the kernel functions available for use by device drivers. In this section, the information for each driver function is organized under the following headings:

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

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

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

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

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

- **SEE ALSO** indicates functions that are related by usage and sources, and which can be referred to for further information.
- **EXAMPLES** shows how the function can be used in driver code.

Every driver MUST include `<sys/ddi.h>` and `<sys/sunddi.h>`, in that order, and as the last files the driver includes.
The following table summarizes the STREAMS functions described in this section.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjmsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>allocb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>backq</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bcanput</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bcanputnext</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bufcall</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>canput</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>canputnext</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>clrbuf</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>copyb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>copymsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>datamsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>dupb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>dupmsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>enableok</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>esballoc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>esbbcall</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>flushband</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>flushq</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>freeb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>freemsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>freezestr</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getq</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>insq</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>linkb</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>msgdsize</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>msgpullup</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>mt-streams</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>noenable</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>OTHERQ</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>pullupmsg</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>put</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>putbq</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>putctl</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>putctl1</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>putnext</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>putnextctl</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>qbufcall</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>qenable</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>qprocson</td>
<td>DDI/DKI</td>
</tr>
</tbody>
</table>
The following table summarizes the functions not specific to STREAMS.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>anocancel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>aphysio</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bcmpr</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bcopy</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>biodone</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bioclone</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biofini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioinit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biomodified</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biosize</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioerror</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>bioreset</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>biowait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bp_mapin</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bp_mapout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>btop</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>btopr</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>bzero</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>cmn_err</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>copyin</td>
<td>DDI/DKI</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997

SunOS 5.6

9F-7
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copyout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>cv_broadcast</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_signal</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_timedwait</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_wait</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>cv_wait_sig</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_add_intr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_add_softintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_btop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_btopr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_copyin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_copyout</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_create_minor_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_is_sid</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_nintrs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_nregs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dev_size</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_device_copy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_device_zero</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_devmap_segmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_addr_bind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_addr_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_alloc_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_buf_bind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_buf_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_burstsizes</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_coff</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_curwin</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_devalign</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_free_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_getwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_hloc</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_mem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_mem_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_movable</td>
<td>Solaris SPARC DDI</td>
</tr>
<tr>
<td>ddi_dma_nextcookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_nextseg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_nextwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_numwin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_sectocookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_set_sbus64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_setup</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

9F-8  SunOS 5.6  modified 22 Jan 1997
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_dma_sync</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dma_unbind_handle</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_dmae</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_1stparty</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_alloc</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_disable</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_enable</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_getattr</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_getcnt</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_getlim</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_prog</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_release</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_dmae_stop</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>ddi_enter_critical</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_exit_critical</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_ffs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_fl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_cred</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_driver_private</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_iblock_cookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_instance</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_name</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_parent</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_soft_iblock_cookie</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_get_soft_state</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getlongprop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getlongprop_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getprop</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getproplen</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_intr_hilevel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putchar</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_ioctl</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997
SunOS 5.6
9F-9
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iomin</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iopb_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_iopb_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_map_regs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_intercept</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_nointercept</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mapdev_set_device_acc_attr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get32</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

9F-10 SunOS 5.6 modified 22 Jan 1997
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_mmap_get_model</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_model_convert_from</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_node_name</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peek8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peekc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peekl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poke8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokec</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokel</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_pokes</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_create</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_exists</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_get_int</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_byte_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_int_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_string</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_lookup_string_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_modify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_op</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_remove</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997 SunOS 5.6 9F-11
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_remove_all</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_undefine</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_byte_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_int</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_int_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_string</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_prop_update_string_array</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_ptob</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_regs_map_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_regs_map_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_intr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_minor_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_remove_softcintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_get8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put16</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put32</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put64</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_put8</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_report_dev</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_root_node</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_segmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_segmap_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_set_driver_private</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_slaveonly</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>ddi_soft_state_fini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_soft_state_zalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_trigger_softintr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_umem_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_umem_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ddi_unmap_regs</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>delay</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>devmap_default_access</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_devmem_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_do_ctxmgt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_load</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_set_ctx_timeout</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_unmem_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_unload</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>disksort</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>drv_getparm</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_hztosec</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_priv</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_usectohz</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>drv_usecwait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>free_pktiopb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>freerbuf</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>get_pktiopb</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>geterror</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getmajor</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getminor</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>getrbuf</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>hat_getkpfnum</td>
<td>DKI only</td>
</tr>
<tr>
<td>inb</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>inl</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>inw</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>kmem_alloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>kmem_free</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>kmem_zalloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>kstat_create</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_delete</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_install</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_named_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_queue</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_back_to_waitq</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>kstat_runq_exit</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997        SunOS 5.6 9F-13
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
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

9F-14   SunOS 5.6   modified 22 Jan 1997
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc_ref</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>proc_signal</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>proc_unref</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ptof</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>repinsb</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>repinsd</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>repinsw</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>repoutsb</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>repoutsd</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>repoutsw</td>
<td>Solaris x86 DDI</td>
</tr>
<tr>
<td>rmalloc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmalloc_wait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmalloccmap</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmalloccmap_wait</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmfree</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rmfreemap</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>rw_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_downgrade</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_enter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_exit</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_read_locked</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_tryenter</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>rw_tryupgrade</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_abort</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_alloc_consistent_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_cname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_destroy_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dmafree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dmaget</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_dname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_errmsg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_free_consistent_buf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_attach</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hbaAttach_setup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_detach</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_fini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_lookup_capstr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_pkt_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_pkt_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_probe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_tran_alloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_hba_tran_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_ifgetcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Function</td>
<td>Library</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>scsi_ifsetcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_init_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_log</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_mname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_pktalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_pktfree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_poll</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_probe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_resalloc</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_reset</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_reset_notify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_resfree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_rname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_slave</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_sname</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_sync_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_transport</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_unprobe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>scsi_unslave</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_destroy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_p</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_p_sig</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_tryp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sema_v</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>sprintf</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>stoi</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strchr</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strcmp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strcpy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strlen</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strncmp</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>strncpy</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>swab</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>timeout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>uiomove</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>utimeout</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>ureadc</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>uwritec</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>va_arg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>va_end</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>va_start</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>vcmn_err</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>vsprintf</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>
### Kernel Functions for Drivers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjmsg(9F)</td>
<td>trim bytes from a message</td>
</tr>
<tr>
<td>allocb(9F)</td>
<td>allocate a message block</td>
</tr>
<tr>
<td>anocancel(9F)</td>
<td>prevent cancellation of asynchronous I/O request</td>
</tr>
<tr>
<td>aphysio(9F)</td>
<td>perform asynchronous physical I/O</td>
</tr>
<tr>
<td>ASSERT(9F)</td>
<td>expression verification</td>
</tr>
<tr>
<td>assert(9F)</td>
<td>See ASSERT(9F)</td>
</tr>
<tr>
<td>backq(9F)</td>
<td>get pointer to the queue behind the current queue</td>
</tr>
<tr>
<td>bcanput(9F)</td>
<td>test for flow control in specified priority band</td>
</tr>
<tr>
<td>bcanputnext(9F)</td>
<td>See canputnext(9F)</td>
</tr>
<tr>
<td>bcm(9F)</td>
<td>compare two byte arrays</td>
</tr>
<tr>
<td>bcopy(9F)</td>
<td>copy data between address locations in the kernel</td>
</tr>
<tr>
<td>bioclone(9F)</td>
<td>clone another buffer</td>
</tr>
<tr>
<td>biodone(9F)</td>
<td>release buffer after buffer I/O transfer and notify blocked threads</td>
</tr>
<tr>
<td>bioerror(9F)</td>
<td>indicate error in buffer header</td>
</tr>
<tr>
<td>biofini(9F)</td>
<td>uninitialize a buffer structure</td>
</tr>
<tr>
<td>bioinit(9F)</td>
<td>initialize a buffer structure</td>
</tr>
<tr>
<td>biomodified(9F)</td>
<td>check if a buffer is modified</td>
</tr>
<tr>
<td>bioreset(9F)</td>
<td>reuse a private buffer header after I/O is complete</td>
</tr>
<tr>
<td>biosize(9F)</td>
<td>returns size of a buffer structure</td>
</tr>
<tr>
<td>biowait(9F)</td>
<td>suspend processes pending completion of block I/O</td>
</tr>
<tr>
<td>bp_mapin(9F)</td>
<td>allocate virtual address space</td>
</tr>
<tr>
<td>bp_mapout(9F)</td>
<td>deallocate virtual address space</td>
</tr>
<tr>
<td>btop(9F)</td>
<td>convert size in bytes to size in pages (round down)</td>
</tr>
<tr>
<td>btopr(9F)</td>
<td>convert size in bytes to size in pages (round up)</td>
</tr>
<tr>
<td>bufcall(9F)</td>
<td>call a function when a buffer becomes available</td>
</tr>
<tr>
<td>bzero(9F)</td>
<td>clear memory for a given number of bytes</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997
<table>
<thead>
<tr>
<th>Function (9F)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>canput(9F)</td>
<td>test for room in a message queue</td>
</tr>
<tr>
<td>canputnext(9F)</td>
<td>test for room in next module’s message queue</td>
</tr>
<tr>
<td>clrbuf(9F)</td>
<td>erase the contents of a buffer</td>
</tr>
<tr>
<td>cmn_err(9F)</td>
<td>display an error message or panic the system</td>
</tr>
<tr>
<td>condvar(9F)</td>
<td>condition variable routines</td>
</tr>
<tr>
<td>copyb(9F)</td>
<td>copy a message block</td>
</tr>
<tr>
<td>copyin(9F)</td>
<td>copy data from a user program to a driver buffer</td>
</tr>
<tr>
<td>copymsg(9F)</td>
<td>copy a message</td>
</tr>
<tr>
<td>copyout(9F)</td>
<td>copy data from a driver to a user program</td>
</tr>
<tr>
<td>csx_AccessConfigurationRegister(9F)</td>
<td>read or write a PC Card Configuration Register</td>
</tr>
<tr>
<td>csx_ConvertSize(9F)</td>
<td>convert device sizes</td>
</tr>
<tr>
<td>csx_ConvertSpeed(9F)</td>
<td>convert device speeds</td>
</tr>
<tr>
<td>csx_CS_DDI_Info(9F)</td>
<td>obtain DDI information</td>
</tr>
<tr>
<td>csx_DeregisterClient(9F)</td>
<td>remove client from Card Services list</td>
</tr>
<tr>
<td>csx_DupHandle(9F)</td>
<td>duplicate access handle</td>
</tr>
<tr>
<td>csx_Error2Text(9F)</td>
<td>convert error return codes to text strings</td>
</tr>
<tr>
<td>csx_Event2Text(9F)</td>
<td>convert events to text strings</td>
</tr>
<tr>
<td>csx_FreeHandle(9F)</td>
<td>free access handle</td>
</tr>
<tr>
<td>csx_Get16(9F)</td>
<td>read data from device address</td>
</tr>
<tr>
<td>csx_Get32(9F)</td>
<td>See csx_Get8(9F)</td>
</tr>
<tr>
<td>csx_Get64(9F)</td>
<td>See csx_Get8(9F)</td>
</tr>
<tr>
<td>csx_Get8(9F)</td>
<td>See csx_Get8(9F)</td>
</tr>
<tr>
<td>csx_GetEventMask(9F)</td>
<td>see csx_SetEventMask(9F)</td>
</tr>
<tr>
<td>csx_GetFirstClient(9F)</td>
<td>return first or next client</td>
</tr>
<tr>
<td>csx_GetFirstTuple(9F)</td>
<td>return Card Information Structure tuple</td>
</tr>
<tr>
<td>csx_GetHandleOffset(9F)</td>
<td>return current access handle offset</td>
</tr>
<tr>
<td>csx_GetMappedAddr(9F)</td>
<td>return mapped virtual address</td>
</tr>
<tr>
<td>csx_GetNextClient(9F)</td>
<td>See csx_GetFirstClient(9F)</td>
</tr>
<tr>
<td>csx_GetNextTuple(9F)</td>
<td>See csx_GetFirstTuple(9F)</td>
</tr>
<tr>
<td>csx_GetStatus(9F)</td>
<td>return the current status of a PC Card and its socket</td>
</tr>
<tr>
<td>csx_GetTupleData(9F)</td>
<td>return the data portion of a tuple</td>
</tr>
</tbody>
</table>
csx_MakeDeviceNode(9F)
csx_MapLogSocket(9F)
csx_MapMemPage(9F)
csx_ModifyConfiguration(9F)
csx_ModifyWindow(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_DEVICE_A(9F)
csx_Parse_CISTPL_DEVICEGEO(9F)
csx_Parse_CISTPL_DEVICEGEO_A(9F)
csx_Parse_CISTPL_DEVICE_OA(9F)
csx_Parse_CISTPL_DEVICE_OC(9F)
csx_Parse_CISTPL_FORMAT(9F)
csx_Parse_CISTPL_FUNCE(9F)
csx_Parse_CISTPL_FUNCID(9F)
csx_Parse_CISTPL_GEOMETRY(9F)
csx_Parse_CISTPL_JEDEC_A(9F)
csx_Parse_CISTPL_JEDEC_C(9F)
csx_Parse_CISTPL_LINKTARGET(9F)
csx_Parse_CISTPL_LONGLINK_A(9F)
csx_Parse_CISTPL_LONGLINK_C(9F)
csx_Parse_CISTPL_LONGLINK_MFC(9F)
csx_Parse_CISTPL_MANFID(9F)
csx_Parse_CISTPL_ORG(9F)
csx_Parse_CISTPL_SPCL(9F)
csx_Parse_CISTPL_SWIL(9F)
csx_Parse_CISTPL_VERS_1(9F)

create and remove minor nodes on behalf of the client
return the physical socket number associated with the client
handle
map the memory area on a PC Card
modify socket and PC Card Configuration Register
modify window attributes
parse the Battery Replacement Date tuple
parse the Byte Order tuple
parse 16-bit Card Configuration Table Entry
tuple
parse Configuration tuple
parse the Card Initialization Date tuple
parse Device Information tuples
See csx_Parse_CISTPL_DEVICE(9F)
parse the Device Geo tuple
parse the Device Geo A tuple
See csx_Parse_CISTPL_DEVICE(9F)
parse the Data Recording Format tuple
parse Function Extension tuple
parse Function Identification tuple
parse the Geometry tuple
See csx_Parse_CISTPL_JEDEC_C(9F)
parse JEDEC Identifier tuples
parse the Link Target tuple
parse the Long Link A and C tuples
See csx_Parse_CISTPL_LONGLINK_A(9F)
parse the Multi-Function tuple
parse Manufacturer Identification tuple
parse the Data Organization tuple
parse the Special Purpose tuple
parse the Software Interleaving tuple
parse Level-1 Version/Product Information
tuple

modified 22 Jan 1997  SunOS 5.6  9F-19
<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>csx_Parse_CISTPL_VERS_2(9F)</code></td>
<td>parse Level-2 Version and Information tuple</td>
</tr>
<tr>
<td><code>csx_ParseTuple(9F)</code></td>
<td>generic tuple parser</td>
</tr>
<tr>
<td><code>csx_Put16(9F)</code></td>
<td>See <code>csx_Put8(9F)</code></td>
</tr>
<tr>
<td><code>csx_Put32(9F)</code></td>
<td>See <code>csx_Put8(9F)</code></td>
</tr>
<tr>
<td><code>csx_Put64(9F)</code></td>
<td>See <code>csx_Put8(9F)</code></td>
</tr>
<tr>
<td><code>csx_Put8(9F)</code></td>
<td>write to device register</td>
</tr>
<tr>
<td><code>csx_RegisterClient(9F)</code></td>
<td>register a client</td>
</tr>
<tr>
<td><code>csx_ReleaseConfiguration(9F)</code></td>
<td>release PC Card and socket configuration</td>
</tr>
<tr>
<td><code>csx_ReleaseIO(9F)</code></td>
<td>See <code>csx_RequestIO(9F)</code></td>
</tr>
<tr>
<td><code>csx_ReleaseIRQ(9F)</code></td>
<td>See <code>csx_RequestIRQ(9F)</code></td>
</tr>
<tr>
<td><code>csx_ReleaseSocketMask(9F)</code></td>
<td>See <code>csx_RequestSocketMask(9F)</code></td>
</tr>
<tr>
<td><code>csx_ReleaseWindow(9F)</code></td>
<td>See <code>csx_RequestWindow(9F)</code></td>
</tr>
<tr>
<td><code>csx_RemoveDeviceNode(9F)</code></td>
<td>See <code>csx_MakeDeviceNode(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepGet16(9F)</code></td>
<td>See <code>csx_RepGet8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepGet32(9F)</code></td>
<td>See <code>csx_RepGet8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepGet64(9F)</code></td>
<td>See <code>csx_RepGet8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepGet8(9F)</code></td>
<td>read repetitively from the device register</td>
</tr>
<tr>
<td><code>csx_RepPut16(9F)</code></td>
<td>See <code>csx_RepPut8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepPut32(9F)</code></td>
<td>See <code>csx_RepPut8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepPut64(9F)</code></td>
<td>See <code>csx_RepPut8(9F)</code></td>
</tr>
<tr>
<td><code>csx_RepPut8(9F)</code></td>
<td>write repetitively to the device register</td>
</tr>
<tr>
<td><code>csx_RequestConfiguration(9F)</code></td>
<td>configure the PC Card and socket</td>
</tr>
<tr>
<td><code>csx_RequestIO(9F)</code></td>
<td>request or release I/O resources for the client</td>
</tr>
<tr>
<td><code>csx_RequestIRQ(9F)</code></td>
<td>request or release IRQ resource</td>
</tr>
<tr>
<td><code>csx_RequestSocketMask(9F)</code></td>
<td>set or clear the client’s client event mask</td>
</tr>
<tr>
<td><code>csx_RequestWindow(9F)</code></td>
<td>request or release window resources</td>
</tr>
<tr>
<td><code>csx_ResetFunction(9F)</code></td>
<td>reset a function on a PC card</td>
</tr>
<tr>
<td><code>csx_SetEventMask(9F)</code></td>
<td>set or return the client event mask for the client</td>
</tr>
<tr>
<td><code>csx_SetHandleOffset(9F)</code></td>
<td>set current access handle offset</td>
</tr>
<tr>
<td><code>csx_ValidateCIS(9F)</code></td>
<td>validate the Card Information Structure (CIS)</td>
</tr>
<tr>
<td><code>cv_broadcast(9F)</code></td>
<td>See <code>condvar(9F)</code></td>
</tr>
<tr>
<td><code>cv_destroy(9F)</code></td>
<td>See <code>condvar(9F)</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>cv_init 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>cv_signal 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>cv_timedwait 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>cv_timedwait_sig 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>cv_wait 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>cv_wait_sig 9F</td>
<td>See condvar(9F)</td>
</tr>
<tr>
<td>datamsg 9F</td>
<td>test whether a message is a data message</td>
</tr>
<tr>
<td>ddi_add_intr 9F</td>
<td>hardware interrupt handling routines</td>
</tr>
<tr>
<td>ddi_add_softintr 9F</td>
<td>software interrupt handling routines</td>
</tr>
<tr>
<td>ddi_binding_name 9F</td>
<td>return driver binding name</td>
</tr>
<tr>
<td>ddi_btop 9F</td>
<td>page size conversions</td>
</tr>
<tr>
<td>ddi_btopr 9F</td>
<td>See ddi_btop 9F</td>
</tr>
<tr>
<td>ddi_copyin 9F</td>
<td>copy data to a driver buffer</td>
</tr>
<tr>
<td>ddi_copyout 9F</td>
<td>copy data from a driver</td>
</tr>
<tr>
<td>ddi_create_minor_node 9F</td>
<td>create a minor node for this device</td>
</tr>
<tr>
<td>ddi_device_copy 9F</td>
<td>copy data from one device register to another device register</td>
</tr>
<tr>
<td>ddi_device_zero 9F</td>
<td>zero fill the device</td>
</tr>
<tr>
<td>ddi_devid_compare 9F</td>
<td>Kernel interfaces for device ids</td>
</tr>
<tr>
<td>ddi_devid_free 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_devid_init 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_devid_register 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_devid_sizeof 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_devid_unregister 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_devid_valid 9F</td>
<td>See ddi_devid_compare 9F</td>
</tr>
<tr>
<td>ddi_dev_is_needed 9F</td>
<td>inform the system that a device’s component is required</td>
</tr>
<tr>
<td>ddi_dev_is_sid 9F</td>
<td>tell whether a device is self-identifying</td>
</tr>
<tr>
<td>ddi_devmap_segmap 9F</td>
<td>See devmap_setup(9F)</td>
</tr>
<tr>
<td>ddi_dev_nintrs 9F</td>
<td>return the number of interrupt specifications a device has</td>
</tr>
<tr>
<td>ddi_dev_nregs 9F</td>
<td>return the number of register sets a device has</td>
</tr>
<tr>
<td>ddi_dma_addr_bind_handle 9F</td>
<td>return the size of a device’s register</td>
</tr>
<tr>
<td>ddi_dma_addr_setup 9F</td>
<td>binds an address to a DMA handle</td>
</tr>
<tr>
<td>ddi_dma_addr_setup 9F</td>
<td>easier DMA setup for use with virtual</td>
</tr>
</tbody>
</table>

modified 22 Jan 1997 SunOS 5.6 9F-21
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_dma_alloc_handle(9F)</td>
<td>allocate DMA handle</td>
</tr>
<tr>
<td>ddi_dma_buf_bind_handle(9F)</td>
<td>binds a system buffer to a DMA handle</td>
</tr>
<tr>
<td>ddi_dma_buf_setup(9F)</td>
<td>easier DMA setup for use with buffer structures</td>
</tr>
<tr>
<td>ddi_dma_burstsizes(9F)</td>
<td>find out the allowed burst sizes for a DMA mapping</td>
</tr>
<tr>
<td>ddi_dma_coff(9F)</td>
<td>convert a DMA cookie to an offset within a DMA handle</td>
</tr>
<tr>
<td>ddi_dma_curwin(9F)</td>
<td>report current DMA window offset and size</td>
</tr>
<tr>
<td>ddi_dma_devalign(9F)</td>
<td>find DMA mapping alignment and minimum transfer size</td>
</tr>
<tr>
<td>ddi_dmae(9F)</td>
<td>system DMA engine functions</td>
</tr>
<tr>
<td>ddi_dmae_1stparty(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_alloc(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_disable(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_enable(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_getattr(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_getcnt(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_getlim(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_prog(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_release(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dmae_stop(9F)</td>
<td>See ddi_dmae(9F)</td>
</tr>
<tr>
<td>ddi_dma_free(9F)</td>
<td>release system DMA resources</td>
</tr>
<tr>
<td>ddi_dma_free_handle(9F)</td>
<td>free DMA handle</td>
</tr>
<tr>
<td>ddi_dma_getwin(9F)</td>
<td>activate a new DMA window</td>
</tr>
<tr>
<td>ddi_dma_htoc(9F)</td>
<td>convert a DMA handle to a DMA address cookie</td>
</tr>
<tr>
<td>ddi_dma_mem_alloc(9F)</td>
<td>allocate memory for DMA transfer</td>
</tr>
<tr>
<td>ddi_dma_mem_free(9F)</td>
<td>free previously allocated memory</td>
</tr>
<tr>
<td>ddi_dma_movwin(9F)</td>
<td>shift current DMA window</td>
</tr>
<tr>
<td>ddi_dma_nextcookie(9F)</td>
<td>retrieve subsequent DMA cookie</td>
</tr>
<tr>
<td>ddi_dma_nextseg(9F)</td>
<td>get next DMA segment</td>
</tr>
<tr>
<td>ddi_dma_nextwin(9F)</td>
<td>get next DMA window</td>
</tr>
<tr>
<td>ddi_dma_numwin(9F)</td>
<td>retrieve number of DMA windows</td>
</tr>
<tr>
<td>ddi_dma_segtocookie(9F)</td>
<td>convert a DMA segment to a DMA address cookie</td>
</tr>
</tbody>
</table>
Cookie Functions for Drivers

- `ddi_dma_set_sbus64(9F)` allow 64 bit transfers on SBus
- `ddi_dma_setup(9F)` setup DMA resources
- `ddi_dma_sync(9F)` synchronize CPU and I/O views of memory
- `ddi_dma_unbind(9F)` unbinds the address in a DMA handle
- `ddi_enter_critical(9F)` enter and exit a critical region of control
- `ddi_exit_critical(9F)` See `ddi_enter_critical(9F)`
- `ddi_ffs(9F)` find first (last) bit set in a long integer
- `ddi_fs(9F)` See `ddi_ffs(9F)`
- `ddi_get16(9F)` See `ddi_get8(9F)`
- `ddi_get32(9F)` See `ddi_get8(9F)`
- `ddi_get64(9F)` See `ddi_get8(9F)`
- `ddi_get8(9F)` read data from the mapped memory
- `ddi_getb(9F)` address, device register or allocated DMA
- `ddi_get_cred(9F)` memory address
- `ddi_get_driver_private(9F)` returns a pointer to the credential structure
- `ddi_get_iblock_cookie(9F)` of the caller
- `ddi_get_instance(9F)` get or set the address of the device’s private
- `ddi_get_l(9F)` data area
- `ddi_getll(9F)` See `ddi_add_intr(9F)`
- `ddi_getlongprop(9F)` get device instance number
- `ddi_getlongprop_buf(9F)` See `ddi_get8(9F)`
- `ddi_get_name(9F)` See `ddi_prop_op(9F)`
- `ddi_get_parent(9F)` find the parent of a device information
- `ddi_getprop(9F)` structure
- `ddi_getproplen(9F)` See `ddi_prop_op(9F)`
- `ddi_getsoft_iblock_cookie(9F)` See `ddi_add_softintr(9F)`
- `ddi_getsoft_state(9F)` See `ddi_soft_state(9F)`
- `ddi_getw(9F)` indicate interrupt handler type
- `ddi_intr_hilevel(9F)` See `ddi_io_get16(9F)`
- `ddi_io_get16(9F)`
### Intro (9F)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_get32(9F)</td>
<td>See ddi_io_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_get8(9F)</td>
<td>read data from the mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_getb(9F)</td>
<td>See ddi_io_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_getl(9F)</td>
<td>See ddi_io_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_getw(9F)</td>
<td>See ddi_io_get8(9F)</td>
</tr>
<tr>
<td>ddi_iomin(9F)</td>
<td>find minimum alignment and transfer size for DMA</td>
</tr>
<tr>
<td>ddi_iopb_alloc(9F)</td>
<td>allocate and free non-sequentially accessed memory</td>
</tr>
<tr>
<td>ddi_iopb_free(9F)</td>
<td>See ddi_iopb_alloc(9F)</td>
</tr>
<tr>
<td>ddi_io_put16(9F)</td>
<td>See ddi_io_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_put32(9F)</td>
<td>See ddi_io_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_put8(9F)</td>
<td>write data to the mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_putb(9F)</td>
<td>See ddi_io_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_putl(9F)</td>
<td>See ddi_io_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_putw(9F)</td>
<td>See ddi_io_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_get16(9F)</td>
<td>See ddi_io_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_get32(9F)</td>
<td>See ddi_io_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_get8(9F)</td>
<td>read multiple data from the mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_getb(9F)</td>
<td>See ddi_io_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_getl(9F)</td>
<td>See ddi_io_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_getw(9F)</td>
<td>See ddi_io_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_put16(9F)</td>
<td>See ddi_io_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_put32(9F)</td>
<td>See ddi_io_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_put8(9F)</td>
<td>write multiple data to the mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_putb(9F)</td>
<td>See ddi_io_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_putl(9F)</td>
<td>See ddi_io_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_io_rep_putw(9F)</td>
<td>See ddi_io_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_mapdev(9F)</td>
<td>create driver-controlled mapping of device</td>
</tr>
<tr>
<td>ddi_mapdev_intercept(9F)</td>
<td>control driver notification of user accesses</td>
</tr>
<tr>
<td>ddi_mapdev_nointercept(9F)</td>
<td>See ddi_mapdev_intercept(9F)</td>
</tr>
<tr>
<td>ddi_mapdev_set_device_acc_attr(9F)</td>
<td>set the device attributes for the mapping</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_map_regs(9F)</td>
<td>map or unmap registers</td>
</tr>
<tr>
<td>ddi_mem_alloc(9F)</td>
<td>allocate and free sequentially accessed memory</td>
</tr>
<tr>
<td>ddi_mem_free(9F)</td>
<td>See ddi_mem_alloc(9F)</td>
</tr>
<tr>
<td>ddi_mem_get16(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_get32(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_get64(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_get8(9F)</td>
<td>read data from mapped device in the memory space or allocated DMA memory</td>
</tr>
<tr>
<td>ddi_mem_getb(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_getl(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_getll(9F)</td>
<td>See ddi_mem_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_put16(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_put32(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_put64(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_put8(9F)</td>
<td>write data to mapped device in the memory space or allocated DMA memory</td>
</tr>
<tr>
<td>ddi_mem_putb(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_putl(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_putll(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_putw(9F)</td>
<td>See ddi_mem_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_get16(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_get32(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_get64(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_get8(9F)</td>
<td>read multiple data from mapped device in the memory space or allocated DMA memory</td>
</tr>
<tr>
<td>ddi_mem_rep_getb(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_getl(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_getll(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_getw(9F)</td>
<td>See ddi_mem_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_put16(9F)</td>
<td>See ddi_mem_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_put32(9F)</td>
<td>See ddi_mem_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_put64(9F)</td>
<td>See ddi_mem_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_mem_rep_put8(9F)</td>
<td>write multiple data to mapped device in the memory space or allocated DMA memory</td>
</tr>
</tbody>
</table>
ddi_mem_rep_putb(9F)  
See ddi_mem_rep_put8(9F)

ddi_mem_rep_putl(9F)  
See ddi_mem_rep_put8(9F)

ddi_mem_rep_putll(9F)  
See ddi_mem_rep_put8(9F)

ddi_mmap_get_model(9F)  
return data model type of current thread

ddi_model_convert_from(9F)  
determine data model type mismatch

ddi_node_name(9F)  
return the devinfo node name

ddi_peek(9F)  
read a value from a location

ddi_peek16(9F)  
See ddi_peek(9F)

ddi_peek32(9F)  
See ddi_peek(9F)

ddi_peek64(9F)  
See ddi_peek(9F)

ddi_peek8(9F)  
See ddi_peek(9F)

ddipeekc(9F)  
See ddi_peek(9F)

ddipeekd(9F)  
See ddi_peek(9F)

ddipeekl(9F)  
See ddi_peek(9F)

ddipeeks(9F)  
See ddi_peek(9F)

ddi_poke(9F)  
write a value to a location

ddi_poke16(9F)  
See ddi_poke(9F)

ddi_poke32(9F)  
See ddi_poke(9F)

ddi_poke64(9F)  
See ddi_poke(9F)

ddi_poke8(9F)  
See ddi_poke(9F)

ddi_pokec(9F)  
See ddi_poke(9F)

ddi_poked(9F)  
See ddi_poke(9F)

ddi_poke16(9F)  
See ddi_poke(9F)

ddi_pokes(9F)  
See ddi_poke(9F)

ddi_prop_create(9F)  
create, remove, or modify properties for leaf device drivers

ddi_prop_exists(9F)  
check for the existence of a property

ddi_prop_free(9F)  
See ddi_prop_lookup(9F)

ddi_prop_get_int(9F)  
lookup integer property

ddi_prop_lookup(9F)  
look up property information

ddi_prop_lookup_byte_array(9F)  
See ddi_prop_lookup(9F)

ddi_prop_lookup_int_array(9F)  
See ddi_prop_lookup(9F)

ddi_prop_lookup_string(9F)  
See ddi_prop_lookup(9F)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_lookup_string_array(9F)</td>
<td>See ddi_prop_lookup(9F)</td>
</tr>
<tr>
<td>ddi_prop_modify(9F)</td>
<td>See ddi_prop_create(9F)</td>
</tr>
<tr>
<td>ddi_prop_op(9F)</td>
<td>get property information for leaf device drivers</td>
</tr>
<tr>
<td>ddi_prop_remove(9F)</td>
<td>See ddi_prop_create(9F)</td>
</tr>
<tr>
<td>ddi_prop_remove_all(9F)</td>
<td>See ddi_prop_create(9F)</td>
</tr>
<tr>
<td>ddi_prop_undefined(9F)</td>
<td>See ddi_prop_create(9F)</td>
</tr>
<tr>
<td>ddi_prop_update(9F)</td>
<td>update properties</td>
</tr>
<tr>
<td>ddi_prop_update_byte_array(9F)</td>
<td>See ddi_prop_update(9F)</td>
</tr>
<tr>
<td>ddi_prop_update_int(9F)</td>
<td>See ddi_prop_update(9F)</td>
</tr>
<tr>
<td>ddi_prop_update_int_array(9F)</td>
<td>See ddi_prop_update(9F)</td>
</tr>
<tr>
<td>ddi_prop_update_string(9F)</td>
<td>See ddi_prop_update(9F)</td>
</tr>
<tr>
<td>ddi_prop_update_string_array(9F)</td>
<td>See ddi_prop_update(9F)</td>
</tr>
<tr>
<td>ddi_ptob(9F)</td>
<td>See ddi_btop(9F)</td>
</tr>
<tr>
<td>ddi_put16(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_put32(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_put64(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_put8(9F)</td>
<td>write data to the mapped memory address, device register or allocated DMA memory address</td>
</tr>
<tr>
<td>ddi_putb(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_putl(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_putll(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_putw(9F)</td>
<td>See ddi_put8(9F)</td>
</tr>
<tr>
<td>ddi_regs_map_free(9F)</td>
<td>free a previously mapped register address space</td>
</tr>
<tr>
<td>ddi_regs_map_setup(9F)</td>
<td>set up a mapping for a register address space</td>
</tr>
<tr>
<td>ddi_remove_intr(9F)</td>
<td>See ddi_add_intr(9F)</td>
</tr>
<tr>
<td>ddi_remove_minor_node(9F)</td>
<td>remove a minor node for this dev_info</td>
</tr>
<tr>
<td>ddi_remove_softintr(9F)</td>
<td>See ddi_add_softintr(9F)</td>
</tr>
<tr>
<td>ddi_rep_get16(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get32(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get64(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get8(9F)</td>
<td>read data from the mapped memory address, device register or allocated DMA memory address</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ddi_rep_getb(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_getl(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_getll(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_getw(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_report_dev(9F)</td>
<td>announce a device</td>
</tr>
<tr>
<td>ddi_rep_get16(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get32(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get64(9F)</td>
<td>See ddi_rep_get8(9F)</td>
</tr>
<tr>
<td>ddi_rep_get8(9F)</td>
<td>write data to the mapped memory address, device register or allocated DMA memory address</td>
</tr>
<tr>
<td>ddi_rep_putb(9F)</td>
<td>See ddi_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_rep_putl(9F)</td>
<td>See ddi_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_rep_putll(9F)</td>
<td>See ddi_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_rep_putw(9F)</td>
<td>See ddi_rep_put8(9F)</td>
</tr>
<tr>
<td>ddi_root_node(9F)</td>
<td>get the root of the dev_info tree</td>
</tr>
<tr>
<td>ddi_segmap(9F)</td>
<td>set up a user mapping using seg_dev</td>
</tr>
<tr>
<td>ddi_segmap_setup(9F)</td>
<td>See ddi_segmap(9F)</td>
</tr>
<tr>
<td>ddi_set_driver_private(9F)</td>
<td>See ddi_get_driver_private(9F)</td>
</tr>
<tr>
<td>ddi_slaveonly(9F)</td>
<td>tell if a device is installed in a slave access only location</td>
</tr>
<tr>
<td>ddi_soft_state(9F)</td>
<td>driver soft state utility routines</td>
</tr>
<tr>
<td>ddi_soft_state_finiset(9F)</td>
<td>See ddi_soft_state(9F)</td>
</tr>
<tr>
<td>ddi_soft_state_free(9F)</td>
<td>See ddi_soft_state(9F)</td>
</tr>
<tr>
<td>ddi_soft_state_init(9F)</td>
<td>See ddi_soft_state(9F)</td>
</tr>
<tr>
<td>ddi_soft_state_zalloc(9F)</td>
<td>See ddi_soft_state(9F)</td>
</tr>
<tr>
<td>ddi_trigger_softintr(9F)</td>
<td>See ddi_add_softintr(9F)</td>
</tr>
<tr>
<td>ddi_umem_alloc(9F)</td>
<td>allocate and free page-aligned kernel memory</td>
</tr>
<tr>
<td>ddi_umem_free(9F)</td>
<td>See ddi_umem_alloc(9F)</td>
</tr>
<tr>
<td>ddi_unmap_regs(9F)</td>
<td>See ddi_map_regs(9F)</td>
</tr>
<tr>
<td>delay(9F)</td>
<td>delay execution for a specified number of clock ticks</td>
</tr>
<tr>
<td>devmap_default_access(9F)</td>
<td>default driver memory access function</td>
</tr>
<tr>
<td>devmap_devmem_setup(9F)</td>
<td>set driver memory mapping parameters</td>
</tr>
<tr>
<td>devmap_do_ctxmgt(9F)</td>
<td>perform device context switching on a</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

**devmap_load**(9F)
set the timeout value for the context management callback

**devmap_set_ctx_timeout**(9F)
set up a user mapping to device memory using the devmap framework

**devmap_setup**(9F)
See **devmap_devmem_setup**(9F)

**devmap_umem_setup**(9F)
control validation of memory address translations

**devmap_unload**(9F)
single direction elevator seek sort for buffers

**disksort**(9F)
retrieve kernel state information

**drv_getparm**(9F)
convert clock ticks to microseconds

**drv_hztousec**(9F)
determine driver privilege

**drv_usectohz**(9F)
convert microseconds to clock ticks

**drv_usecwait**(9F)
busy-wait for specified interval

**dupb**(9F)
duplicate a message block descriptor

**dupmsg**(9F)
duplicate a message

**enableok**(9F)
reschedule a queue for service

**esballoc**(9F)
allocate a message block using a caller-supplied buffer

**esbbcall**(9F)
call function when buffer is available

**flushband**(9F)
flush messages for a specified priority band

**flushq**(9F)
remove messages from a queue

**freeb**(9F)
free a message block

**freemsg**(9F)
free all message blocks in a message

**free_pktiopb**(9F)
See **get_pktiopb**(9F)

**freerbuf**(9F)
free a raw buffer header

**freezestr**(9F)
freeze, thaw the state of a stream

**geterror**(9F)
return I/O error

**getmajor**(9F)
get major device number

**getminor**(9F)
get minor device number

**get_pktiopb**(9F)
allocate/free a SCSI packet in the iopb map

**getq**(9F)
get the next message from a queue

**getrbuf**(9F)
get a raw buffer header

**hat_getkpfnnum**(9F)
get page frame number for kernel address

---

modified 22 Jan 1997
SunOS 5.6
9F-29
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inb(9F)</td>
<td>read from an I/O port</td>
</tr>
<tr>
<td>inl(9F)</td>
<td>See inb(9F)</td>
</tr>
<tr>
<td>insq(9F)</td>
<td>insert a message into a queue</td>
</tr>
<tr>
<td>inw(9F)</td>
<td>See inb(9F)</td>
</tr>
<tr>
<td>IOC_CONVERT_FROM(9F)</td>
<td>determine if there is a need to translate M_IOCTL contents.</td>
</tr>
<tr>
<td>kmem_alloc(9F)</td>
<td>allocate kernel memory</td>
</tr>
<tr>
<td>kmem_free(9F)</td>
<td>See kmem_alloc(9F)</td>
</tr>
<tr>
<td>kmem_zalloc(9F)</td>
<td>See kmem_alloc(9F)</td>
</tr>
<tr>
<td>kstat_create(9F)</td>
<td>create and initialize a new kstat</td>
</tr>
<tr>
<td>kstat_delete(9F)</td>
<td>remove a kstat from the system</td>
</tr>
<tr>
<td>kstat_install(9F)</td>
<td>add a fully initialized kstat to the system</td>
</tr>
<tr>
<td>kstat_named_init(9F)</td>
<td>initialize a named kstat</td>
</tr>
<tr>
<td>kstat_queue(9F)</td>
<td>update I/O kstat statistics</td>
</tr>
<tr>
<td>kstat_runq_back_to_waitq(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>kstat_runq_enter(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>kstat_runq_exit(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>kstat_waitq_enter(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>kstat_waitq_exit(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>kstat_waitq_to_runq(9F)</td>
<td>See kstat_queue(9F)</td>
</tr>
<tr>
<td>linkb(9F)</td>
<td>concatenate two message blocks</td>
</tr>
<tr>
<td>makecom(9F)</td>
<td>make a packet for SCSI commands</td>
</tr>
<tr>
<td>makecom_g0(9F)</td>
<td>See makecom(9F)</td>
</tr>
<tr>
<td>makecom_g0_s(9F)</td>
<td>See makecom(9F)</td>
</tr>
<tr>
<td>makecom_g1(9F)</td>
<td>See makecom(9F)</td>
</tr>
<tr>
<td>makecom_g5(9F)</td>
<td>See makecom(9F)</td>
</tr>
<tr>
<td>makedevice(9F)</td>
<td>make device number from major and minor numbers</td>
</tr>
<tr>
<td>max(9F)</td>
<td>return the larger of two integers</td>
</tr>
<tr>
<td>min(9F)</td>
<td>return the lesser of two integers</td>
</tr>
<tr>
<td>minphys(9F)</td>
<td>See physio(9F)</td>
</tr>
<tr>
<td>mkiocb(9F)</td>
<td>allocates a STREAMS ioctl block for M_IOCTL messages in the kernel.</td>
</tr>
<tr>
<td>mod_info(9F)</td>
<td>See mod_install(9F)</td>
</tr>
<tr>
<td>mod_install(9F)</td>
<td>add, remove or query a loadable module</td>
</tr>
<tr>
<td>mod_remove(9F)</td>
<td>See mod_install(9F)</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

msgdsize(9F) return the number of bytes in a message
msgpullup(9F) concatenate bytes in a message
mt-streams(9F) STREAMS multithreading
mutex(9F) mutual exclusion lock routines
mutex_destroy(9F) See mutex(9F)
mutex_enter(9F) See mutex(9F)
mutex_exit(9F) See mutex(9F)
mutex_init(9F) See mutex(9F)
mutex_owned(9F) See mutex(9F)
mutex_tryenter(9F) See mutex(9F)

nochpoll(9F) error return function for non-pollable devices
nodev(9F) error return function
noenable(9F) prevent a queue from being scheduled
nulldev(9F) zero return function
numtos(9F) See stoi(9F)
OTHERQ(9F) get pointer to queue’s partner queue
otherq(9F) See OTHERQ(9F)
outb(9F) write to an I/O port
outl(9F) See outb(9F)
outw(9F) See outb(9F)

pci_config_get16(9F) See pci_config_get8(9F)
pci_config_get32(9F) See pci_config_get8(9F)
pci_config_get64(9F) See pci_config_get8(9F)
pci_config_get8(9F) read or write single datum of various sizes
to the PCI Local Bus Configuration space

pci_config_getb(9F) See pci_config_get8(9F)
pci_config_getl(9F) See pci_config_get8(9F)
pci_config_getll(9F) See pci_config_get8(9F)
pci_config_getw(9F) See pci_config_get8(9F)
pci_config_put16(9F) See pci_config_get8(9F)
pci_config_put32(9F) See pci_config_get8(9F)
pci_config_put64(9F) See pci_config_get8(9F)
pci_config_put8(9F) See pci_config_get8(9F)
pci_config_putl(9F) See pci_config_get8(9F)

modified 22 Jan 1997

SunOS 5.6

9F-31
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_putll(9F)</td>
<td>See pci_config_get8(9F)</td>
</tr>
<tr>
<td>pci_config_putw(9F)</td>
<td>See pci_config_get8(9F)</td>
</tr>
<tr>
<td>pci_config_setup(9F)</td>
<td>setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space</td>
</tr>
<tr>
<td>pci_config_teardown(9F)</td>
<td>See pci_config_setup(9F)</td>
</tr>
<tr>
<td>physio(9F)</td>
<td>perform physical I/O</td>
</tr>
<tr>
<td>pm_busy_component(9F)</td>
<td>control device components’ availability for power management</td>
</tr>
<tr>
<td>pm_create_components(9F)</td>
<td>create or destroy power-manageable components</td>
</tr>
<tr>
<td>pm_destroy_components(9F)</td>
<td>See pm_create_components(9F)</td>
</tr>
<tr>
<td>pm_get_normal_power(9F)</td>
<td>get or set a device component’s normal power level</td>
</tr>
<tr>
<td>pm_idle_component(9F)</td>
<td>See pm_busy_component(9F)</td>
</tr>
<tr>
<td>pm_set_normal_power(9F)</td>
<td>See pm_get_normal_power(9F)</td>
</tr>
<tr>
<td>pollwakeup(9F)</td>
<td>inform a process that an event has occurred</td>
</tr>
<tr>
<td>proc_ref(9F)</td>
<td>See proc_signal(9F)</td>
</tr>
<tr>
<td>proc_signal(9F)</td>
<td>send a signal to a process</td>
</tr>
<tr>
<td>proc_unref(9F)</td>
<td>See proc_signal(9F)</td>
</tr>
<tr>
<td>ptob(9F)</td>
<td>convert size in pages to size in bytes</td>
</tr>
<tr>
<td>pullupmsg(9F)</td>
<td>concatenate bytes in a message</td>
</tr>
<tr>
<td>put(9F)</td>
<td>call a STREAMS put procedure</td>
</tr>
<tr>
<td>putbq(9F)</td>
<td>place a message at the head of a queue</td>
</tr>
<tr>
<td>putctl(9F)</td>
<td>send a control message to a queue</td>
</tr>
<tr>
<td>putctl1(9F)</td>
<td>send a control message with a one-byte parameter to a queue</td>
</tr>
<tr>
<td>putnext(9F)</td>
<td>send a message to the next queue</td>
</tr>
<tr>
<td>putnextctl(9F)</td>
<td>send a control message to a queue</td>
</tr>
<tr>
<td>putnextctl1(9F)</td>
<td>send a control message with a one-byte parameter to a queue</td>
</tr>
<tr>
<td>putq(9F)</td>
<td>put a message on a queue</td>
</tr>
<tr>
<td>qbufcall(9F)</td>
<td>call a function when a buffer becomes available</td>
</tr>
<tr>
<td>qenable(9F)</td>
<td>enable a queue</td>
</tr>
<tr>
<td>qprocsoff(9F)</td>
<td>See qprocson(9F)</td>
</tr>
<tr>
<td>qprocson(9F)</td>
<td>enable, disable put and service routines</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

- `qreply(9F)` - send a message on a stream in the reverse direction
- `qsize(9F)` - find the number of messages on a queue
- `qtimeout(9F)` - execute a function after a specified length of time
- `qunbufcall(9F)` - cancel a pending qbufcall request
- `qtimeout(9F)` - cancel previous qtimeout function call
- `qwait(9F)` - STREAMS wait routines
- `qwait_sig(9F)` - See `qwait(9F)`
- `qwriter(9F)` - asynchronous STREAMS perimeter upgrade
- `RD(9F)` - get pointer to the read queue
- `rd(9F)` - See `RD(9F)`
- `repinsb(9F)` - See `inb(9F)`
- `repinsd(9F)` - See `inb(9F)`
- `repinsw(9F)` - See `inb(9F)`
- `repoutsb(9F)` - See `outb(9F)`
- `repoutsd(9F)` - See `outb(9F)`
- `repoutsw(9F)` - See `outb(9F)`
- `rmalloc(9F)` - allocate space from a resource map
- `rmallocmap(9F)` - allocate and free resource maps
- `rmallocmap_wait(9F)` - See `rmallocmap(9F)`
- `rmalloc_wait(9F)` - allocate space from a resource map, wait if necessary
- `rmfree(9F)` - free space back into a resource map
- `rmfreemap(9F)` - See `rmallocmap(9F)`
- `rmvb(9F)` - remove a message block from a message
- `rmvq(9F)` - remove a message from a queue
- `rw_destroy(9F)` - See `rwlock(9F)`
- `rw_downgrade(9F)` - See `rwlock(9F)`
- `rw_enter(9F)` - See `rwlock(9F)`
- `rw_exit(9F)` - See `rwlock(9F)`
- `rw_init(9F)` - See `rwlock(9F)`
- `rwlock(9F)` - readers/writer lock functions
- `rw_read_locked(9F)` - See `rwlock(9F)`
- `rw_tryenter(9F)` - See `rwlock(9F)`
rw_tryupgrade(9F)  See rwlock(9F)
samestr(9F)  See SAMESTR(9F)
SAMESTR(9F)
scsi_abort(9F)
scsi_alloc_consistent_buf(9F)
scsi_cname(9F)
scsi_destroy_pkt(9F)
scsi_dmafree(9F)  See scsi_dmaget(9F)
scsi_dmaget(9F)
scsi_dname(9F)
scsi_errmsg(9F)
scsi_free_consistent_buf(9F)
scsi_hba_attach(9F)  See scsi_hba_attach_setup(9F)
scsi_hba_attach_setup(9F)
scsi_hba_detach(9F)  See scsi_hba_attach_setup(9F)
scsi_hba_fini(9F)
scsi_hba_init(9F)
scsi_hba_lookup_capstr(9F)
scsi_hba_pkt_alloc(9F)  See scsi_hba_pkt_alloc(9F)
scsi_hba_pkt_free(9F)
scsi_hba_probe(9F)
scsi_hba_tran_alloc(9F)  See scsi_hba_tran_alloc(9F)
scsi_hba_tran_free(9F)
scsi_ifgetcap(9F)  See scsi_ifgetcap(9F)
scsi_ifsetcap(9F)
scsi_init_pkt(9F)
scsi_log(9F)
scsi_mname(9F)  See scsi_cname(9F)
scsi_pktalloc(9F)
scsi_pktfree(9F)  See scsi_pktalloc(9F)
scsi_poll(9F)
run a polled SCSI command on behalf of a target driver
utility for probing a scsi device
scsi_probe(9F)

See rwlock(9F)
See SAMESTR(9F)
test if next queue is in the same stream
abort a SCSI command
allocate an I/O buffer for SCSI DMA
decode a SCSI name
free an allocated SCSI packet and its DMA resource
See scsi_dmaget(9F)
SCSI dma utility routines
See scsi_cname(9F)
display a SCSI request sense message
free a previously allocated SCSI DMA I/O buffer
See scsi_hba_attach_setup(9F)
SCSI HBA attach and detach routines
See scsi_hba_attach_setup(9F)
See scsi_hba_init(9F)
SCSI Host Bus Adapter system initialization and completion routines
return index matching capability string
allocate and free a scsi_pkt structure
See scsi_hba_pkt_alloc(9F)
default SCSI HBA probe function
allocate and free transport structures
See scsi_hba_tran_alloc(9F)
get/set SCSI transport capability
See scsi_ifgetcap(9F)
prepare a complete SCSI packet
display a SCSI-device-related message
See scsi_cname(9F)
SCSI packet utility routines
See scsi_pktalloc(9F)
run a polled SCSI command on behalf of a target driver
utility for probing a scsi device
scsi_probe(9F)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_resalloc(9F)</td>
<td>See scsi_pktalloc(9F)</td>
</tr>
<tr>
<td>scsi_reset(9F)</td>
<td>reset a SCSI bus or target</td>
</tr>
<tr>
<td>scsi_reset_notify(9F)</td>
<td>notify target driver of bus resets</td>
</tr>
<tr>
<td>scsi_resfree(9F)</td>
<td>See scsi_pktalloc(9F)</td>
</tr>
<tr>
<td>scsi_rname(9F)</td>
<td>See scsi_cname(9F)</td>
</tr>
<tr>
<td>scsi_slave(9F)</td>
<td>utility for SCSI target drivers to establish the presence of a target</td>
</tr>
<tr>
<td>scsi_sname(9F)</td>
<td>See scsi_cname(9F)</td>
</tr>
<tr>
<td>scsi_sync_pkt(9F)</td>
<td>synchronize CPU and I/O views of memory</td>
</tr>
<tr>
<td>scsi_transport(9F)</td>
<td>request by a SCSI target driver to start a command</td>
</tr>
<tr>
<td>scsi_unprobe(9F)</td>
<td>free resources allocated during initial probing</td>
</tr>
<tr>
<td>scsi_unslave(9F)</td>
<td>See scsi_unprobe(9F)</td>
</tr>
<tr>
<td>sema_destroy(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>sema_init(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>sema_p(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>semaphore(9F)</td>
<td>semaphore functions</td>
</tr>
<tr>
<td>sema_p_sig(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>sema_tryp(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>sema_v(9F)</td>
<td>See semaphore(9F)</td>
</tr>
<tr>
<td>sprintf(9F)</td>
<td>format characters in memory</td>
</tr>
<tr>
<td>stoi(9F)</td>
<td>convert between an integer and a decimal string</td>
</tr>
<tr>
<td>strchr(9F)</td>
<td>find a character in a string</td>
</tr>
<tr>
<td>strcmp(9F)</td>
<td>compare two null terminated strings.</td>
</tr>
<tr>
<td>strcpy(9F)</td>
<td>copy a string from one location to another.</td>
</tr>
<tr>
<td>strlen(9F)</td>
<td>determine the number of non-null bytes in a string</td>
</tr>
<tr>
<td>strlog(9F)</td>
<td>submit messages to the log driver</td>
</tr>
<tr>
<td>strncmp(9F)</td>
<td>See strcmp(9F)</td>
</tr>
<tr>
<td>strncpy(9F)</td>
<td>See strcpy(9F)</td>
</tr>
<tr>
<td>strqget(9F)</td>
<td>get information about a queue or band of the queue</td>
</tr>
<tr>
<td>strqset(9F)</td>
<td>change information about a queue or band of the queue</td>
</tr>
<tr>
<td>swab(9F)</td>
<td>swap bytes in 16-bit halfwords</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>testb(9F)</td>
<td>check for an available buffer</td>
</tr>
<tr>
<td>timeout(9F)</td>
<td>execute a function after a specified length of time</td>
</tr>
<tr>
<td>uiomove(9F)</td>
<td>copy kernel data using uio structure</td>
</tr>
<tr>
<td>unbufcall(9F)</td>
<td>cancel a pending bufcall request</td>
</tr>
<tr>
<td>unfreezestr(9F)</td>
<td>See freezestr(9F)</td>
</tr>
<tr>
<td>unlinkb(9F)</td>
<td>remove a message block from the head of a message</td>
</tr>
<tr>
<td>untimeout(9F)</td>
<td>cancel previous timeout function call</td>
</tr>
<tr>
<td>ureadc(9F)</td>
<td>add character to a uio structure</td>
</tr>
<tr>
<td>uwritec(9F)</td>
<td>remove a character from a uio structure</td>
</tr>
<tr>
<td>va_arg(9F)</td>
<td>handle variable argument list</td>
</tr>
<tr>
<td>va_copy(9F)</td>
<td>See va_arg(9F)</td>
</tr>
<tr>
<td>va_end(9F)</td>
<td>See va_arg(9F)</td>
</tr>
<tr>
<td>va_start(9F)</td>
<td>See va_arg(9F)</td>
</tr>
<tr>
<td>vcmn_err(9F)</td>
<td>See cmn_err(9F)</td>
</tr>
<tr>
<td>vsprintf(9F)</td>
<td>format characters in memory</td>
</tr>
<tr>
<td>WR(9F)</td>
<td>get pointer to the write queue for this module or driver</td>
</tr>
<tr>
<td>wr(9F)</td>
<td>See WR(9F)</td>
</tr>
</tbody>
</table>
NAME  adjmsg – trim bytes from a message

SYNOPSIS  
#include <sys/stream.h>

int adjmsg(mblk_t *mp, ssize_t len);

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

ARGUMENTS  
mp  Pointer to the message to be trimmed.

len  The number of bytes to be removed.

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

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

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

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

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

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

SEE ALSO  STREAMS Programming Guide

modified 20 Nov 1996  SunOS 5.6  9F-37
**NAME**
allocb – allocate a message block

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

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

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

**ARGUMENTS**
- **size**: The number of bytes in the message block.
- **pri**: Priority of the request (no longer used).

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

A STREAMS message block is composed of three structures. The first structure is a message block (mblk_t). See msgb(9S). The mblk_t structure points to a data block structure (dblk_t). See datab(9S). Together these two structures describe the message type (if applicable) and the size and location of the third structure, the data buffer. The data buffer contains the data for this message block.

The fields in the mblk_t structure are initialized as follows:
- **b_cont**: set to NULL
- **b_rptr**: points to the beginning of the data buffer
- **b_wptr**: points to the beginning of the data buffer
- **b_datap**: points to the dblk_t structure

The fields in the dblk_t structure are initialized as follows:
- **db_base**: points to the first byte of the data buffer
- **db_lim**: points to the last byte + 1 of the buffer
- **db_type**: set to M_DATA

The following figure identifies the data structure members that are affected when a message block is allocated.
RETURN VALUES
A pointer to the allocated message block of type M_DATA on success.
A NULL pointer on failure.

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

EXAMPLES
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.

```c
send_error(q, err)
{
queue_t *q;
unsigned char err;

mblk_t *bp;

if ((bp = allocb(1, BPRI_HI)) == NULL) /* allocate msg. block */
    return(0);

bp->b_datap->db_type = M_ERROR; /* set msg type to M_ERROR */
*bp->b_wptr++ = err; /* increment write pointer */

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

SEE ALSO
RD(9F), bufcall(9F), esballoc(9F), esbcall(9F), putnext(9F), testb(9F), datab(9S),
msgb(9S)

Writing Device Drivers
STREAMS Programming Guide

NOTES
The pri argument is no longer used, but is retained for compatibility with existing drivers.
NAME  anocancel – prevent cancellation of asynchronous I/O request

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

int anocancel();

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

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

RETURN VALUES
anocancel() returns ENXIO.

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

Writing Device Drivers
### NAME
aphysio – perform asynchronous physical I/O

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

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

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

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

### DESCRIPTION
aphysio() performs asynchronous I/O operations between the device and the address space described by `aio_reqp->aio_uio`. Prior to the start of the transfer, aphysio() verifies the requested operation is valid. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, `strat`, is then called one or more times to perform the physical I/O operations. aphysio() does not wait for each transfer to complete, but returns as soon as the necessary requests have been made.

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

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

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

---

modified 9 Nov 1994

SunOS 5.6

9F-41
aphysio() returns:
0 on success.
error number on failure.

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

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

Writing Device Drivers

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

BUGS  
Cancellation is not supported in this release. The address of the function anocancel(9F) must be used as the cancel argument.
## NAME
ASSERT, assert – expression verification

## SYNOPSIS
```c
#include <sys/debug.h>
void ASSERT(EX);
```

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

## ARGUMENTS
- `EX` boolean expression.

## DESCRIPTION
`ASSERT()` is a macro which checks to see if the expression `EX` is true. If it is not then `ASSERT()` causes an error message to be logged to the console and the system to panic. `ASSERT()` works only if the preprocessor symbol `DEBUG` is defined.

## CONTEXT
`ASSERT()` can be used from user or interrupt context.

## SEE ALSO
- *Writing Device Drivers*
NAME backq – get pointer to the queue behind the current queue

SYNOPSIS
#include <sys/stream.h>
queue_t *backq(queue_t *cq);

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

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

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

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

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

SEE ALSO
queue(9S)
Writing Device Drivers
STREAMS Programming Guide
NAME  bcanput – test for flow control in specified priority band

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

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

INTERFACE  Level 1 (DDI/DKI).

ARGUMENTS  
- `q`  Pointer to the message queue.
- `pri`  Message priority.

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

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

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

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

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

SEE ALSO  bcanputnext(9F), canput(9F), canputnext(9F), putbq(9F), putnext(9F)

Writing Device Drivers
STREAMS Programming Guide

WARNINGS  Drivers are responsible for both testing a queue with bcanput() and refraining from placing a message on the queue if bcanput() fails.
**NAME**
bcmp – compare two byte arrays

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

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

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

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

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

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

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

**SEE ALSO**
- `strcmp(9F)`
- *Writing Device Drivers*

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

SYNOPSIS
#include <sys/types.h>
void bcopy(const void *from, void *to, size_t bcount);

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

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

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

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

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

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

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

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

modified 1 May 1996 SunOS 5.6 9F-47
bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0],
   bp->b_bcount);

SEE ALSO  
copyin(9F), copyout(9F)  
Writing Device Drivers

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

bioclone – clone another buffer

SYNOPSIS

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

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

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

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

DESCRIPTION

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

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

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

modified 20 Nov 1996

SunOS 5.6

9F-49
**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**
A device driver can use `bioclone()` for disk striping. For each disk in the stripe, a clone buffer is created which performs I/O to a portion of the original buffer.

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

static u_int
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) {
```
SEE ALSO biodone(9F), bp_mapin(9F), freerbuf(9F), getbuf(9F), buf(9S)

Writing Device Drivers

modified 20 Nov 1996

SunOS 5.6

9F-51
NAME  biodone – release buffer after buffer I/O transfer and notify blocked threads

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

void biodone(struct buf *bp);

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

ARGUMENTS  bp  Pointer to a buf(9S) structure.

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

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

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

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

```c
#define RAMDNBLK 1000  /* Number of blocks in RAM disk */
#define RAMDBSIZ 512    /* Number of bytes per block */
char ramdblks[RAMDNBLK][RAMDBSIZ];  /* Array containing RAM disk */

static int ramdstrategy(struct buf *bp)
{
  daddr_t blkno = bp->b_bblkno;  /* get block number */
  if ((blkno < 0) || (blkno >= RAMDNBLK)) {
    /* If requested block is outside RAM disk limits, test for EOF which could result from a direct (physio) request. */
    if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
    /*
    if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
```
* If read is for block beyond RAM disk
* limits, mark EOF condition.
*/
21 bp->b_resid = bp->b_bcount; /* compute return value */
22
23 } else { /* I/O attempt is beyond */
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)

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

NOTES Drivers that use the b_iiodone field of the buf(9S) structure to specify a substitute com-
pletion routine should save the value of b_iiodone before changing it, and then restore the
old value before calling biodone() to release the buffer.
NAME    bioerror – indicate error in buffer header

SYNOPSIS    #include <sys/types.h>
    #include <sys/buf.h>
    #include <sys/ddi.h>
    void bioerror(struct buf *bp, int error);

INTERFACE LEVEL    Solaris DDI specific (Solaris DDI)

ARGUMENTS    bp       Pointer to the buf(9S) structure describing the transfer.
    error    Error number to be set, or zero to clear an error indication.

DESCRIPTION    If error is non-zero, bioerror() indicates an error has occurred in the buf(9S) structure. A subsequent call to geterror(9F) will return error.
    If error is 0, the error indication is cleared and a subsequent call to geterror(9F) will return 0.

CONTEXT    bioerror() can be called from any context.

SEE ALSO    strategy(9E), geterror(9F), getrbuf(9F), buf(9S)
NAME  biofini – uninitialize a buffer structure

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

void biofini(struct buf *bp);

INTERFACE   LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
bp  Pointer to the buffer header structure.

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

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

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

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

Writing Device Drivers
NAME  bioinit – initialize a buffer structure

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

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

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
- `bp`  Pointer to the buffer header structure.

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

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

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

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

Writing Device Drivers
### NAME
biomodi®ed – check if a buffer is modified

### SYNOPSIS
```
#include <sys/ddi.h>
#include <sys/sunddi.h>
int biomodi®ed(struct buf *bp);
```

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

### ARGUMENTS
- `bp`: Pointer to the buffer header structure.

### DESCRIPTION
The `biomodi®ed()` function returns status to indicate if the buffer is modified. The `biomodi®ed()` function is only supported for paged-I/O request, that is the B_PAGEIO flag must be set in the `b_flags` field of the `buf(9S)` structure. The `biomodi®ed()` 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 `biomodi®ed()` function can be used to detect any modifications to the memory pages while I/O is in progress.

### RETURN VALUES
The `biomodi®ed()` function returns the following values:
- 1: Buffer is modified.
- 0: Buffer is not modified.
- -1: Buffer is not used for paged I/O request.

### CONTEXT
`biomodi®ed()` can be called from any context.

### EXAMPLES
A device driver can use `biomodi®ed()` for disk mirroring. An application is allowed to `mmap` a file which can reside on a disk which is mirrored by multiple submirrors. If the file system writes the file to the backing store, it is written to all submirrors in parallel. It must be ensured that the copies on all submirrors are identical. The `biomodi®ed()` 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.

### SEE ALSO
- `bp_mapin(9F)`, `buf(9S)`
- *Writing Device Drivers*

---

modified 20 Nov 1996

SunOS 5.6

9F-57
NAME  bioreset – reuse a private buffer header after I/O is complete

SYNOPSIS  #include <sys/buf.h>
#include <sys/ddi.h>
void bioreset(struct buf *bp);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI)

ARGUMENTS  bp  Pointer to the buf (9S) structure.

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

CONTEXT  bioreset() can be called from any context.

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

NOTES  bp must not describe a transfer in progress.
NAME    biosize – returns size of a buffer structure

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

size_t biosize(void);

INTERFACE LEVEL    Solaris DDI specific (Solaris DDI).

DESCRIPTION    The biosize() function returns the size in bytes of the buf(9S) structure. The biosize() function is used by drivers in combination with kmem_alloc(9F) and bionit(9F) to allocate buffer structures embedded in other data structures.

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

SEE ALSO    biofini(9F), bioinit(9F), getbuf(9F), kmem_alloc(9F), buf(9S)

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

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

int biowait(struct buf *bp);

INTERFACE LEVEL
ARGUMENTS  bp  Pointer to the buf structure describing the transfer.

DESCRIPTION  Drivers allocating their own buf structures with getrbuf(9F) can use the biowait() function to suspend the current thread and wait for completion of the transfer. Drivers must call biodone(9F) when the transfer is complete to notify the thread blocked by biowait(). biodone() is usually called in the interrupt routine.

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

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

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

Writing Device Drivers
<table>
<thead>
<tr>
<th>NAME</th>
<th>bp_mapin – allocate virtual address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/types.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>#include &lt;sys/buf.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>void bp_mapin(struct buf *bp);</code></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td><code>bp</code> Pointer to the buffer header structure.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td><code>bp_mapin()</code> is used to map virtual address space to a page list maintained by the buffer header during a paged-I/O request. <code>bp_mapin()</code> allocates system virtual address space, maps that space to the page list, and returns the starting address of the space in the <code>bp-&gt;b_un.b_addr</code> field of the <code>buf(9S)</code> structure. Virtual address space is then deallocated using the <code>bp_mapout(9F)</code> function. If a null page list is encountered, <code>bp_mapin()</code> returns without allocating space and no mapping is performed.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td><code>bp_mapin()</code> can be called from user context only.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>bp_mapout(9F), buf(9S)</code></td>
</tr>
</tbody>
</table>

*Writing Device Drivers*

modified 13 Sep 1992

SunOS 5.6

9F-61
### NAME
bp_mapout – deallocate virtual address space

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

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

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

### ARGUMENTS
- `bp`  
  Pointer to the buffer header structure.

### DESCRIPTION
`bp_mapout()` deallocates system virtual address space allocated by a previous call to `bp_mapin`(9F). `bp_mapout()` should only be called on buffers which have been allocated and are owned by the device driver. It must not be called on buffers passed to the driver through the `strategy`(9E) entry point (for example a filesystem). Because `bp_mapin`(9F) does not keep a reference count, `bp_mapout()` will wipe out any kernel mapping that a layer above the device driver might rely on.

### CONTEXT
`bp_mapout()` can be called from user context only.

### SEE ALSO
- `strategy`(9E), `bp_mapin`(9F), `buf`(9S)
- *Writing Device Drivers*
### NAME  
**btop** – convert size in bytes to size in pages (round down)

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

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

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

### ARGUMENTS  
- `numbytes`  
  Number of bytes.

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

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

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

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

*Writing Device Drivers*

--

modified 11 Apr 1991

SunOS 5.6

9F-63
NAME | btopr – convert size in bytes to size in pages (round up)

SYNOPSIS | #include <sys/ddi.h>

unsigned long btopr(unsigned long numbytes);

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

ARGUMENTS | numbytes Number of bytes.

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

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

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

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

Writing Device Drivers
NAME
bufcall – call a function when a buffer becomes available

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

int bufcall (size_t size, uint pri, void (*func)(intptr_t), intptr_t arg);

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

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

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

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

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

EXAMPLES
The purpose of this srv(9E) service routine is to add a header to all M_DATA messages.

Service routines must process all messages on their queues before returning, or arrange to be rescheduled.

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

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

If the buffer allocation was successful, initialize the header (lines 25–28), make the message type M_PROTO (line 29), link the M_DATA message to it (line 30), and pass it on (line 31).
Note that this example ignores the bookkeeping needed to handle `bufcall()` and `timeout(9F)` cancellation for ones that are still outstanding at close time.

```c
1 struct hdr {
2    unsigned int h_size;
3    int h_version;
4};
5
6 void xxxsrv(q)
7    queue_t *q;
8 {
9    mblk_t *bp;
10   mblk_t *mp;
11   struct hdr *hp;
12
13    while ((mp = getq(q)) != NULL) {
14       if (mp->b_datap->db_type >= QPCTL ||
15          canputnext(q)) {
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, (long)q)) /* try to reschedule */
24                     timeout(qenable, (caddr_t)q, drv_usectohz(500000));
25                     return (0);
26             } else { /* normal priority, canputnext failed */
27                 putbq(q, mp); /* put back on the message queue */
28                 return (0);
29             }
30          }
31      } else { /* normal priority, canputnext failed */
32          putbq(q, mp); /* put back on the message queue */
33          return (0);
34      }
35    }
36    return (0);
37 }
```
Kernel Functions for Drivers

SEE ALSO  
srv(9E), allocb(9F), canputnext(9F), esbaloc(9F), esbbcall(9F), putnext(9F), qenable(9F), testb(9F), timeout(9F), unbufcall(9F)

Writing Device Drivers
STREAMS Programming Guide

WARNINGS  
Even when `func` is called by `bufcall()`, `allocb(9F)` can fail if another module or driver had allocated the memory before `func` was able to call `allocb(9F)`.
NAME

`bzero` – clear memory for a given number of bytes

SYNOPSIS

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

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

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

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

DESCRIPTION

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

CONTEXT

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

SEE ALSO

`bcopy(9F), clrbuf(9F), kmem_zalloc(9F)`

*Writing Device Drivers*

WARNINGS

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

### SYNOPSIS
```
#include <sys/stream.h>
int canput(queue_t *q);
```

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

**ARGUMENTS**
- `q` Pointer to the message queue.

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

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

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

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

### SEE ALSO
- `bcanput(9F)`, `bcanputnext(9F)`, `canputnext(9F)`, `putbq(9F)`, `putnext(9F)`

*Writing Device Drivers*
*STREAMS Programming Guide*

### WARNINGS
Drivers are responsible for both testing a queue with `canput()` and refraining from placing a message on the queue if `canput()` fails.
**NAME**
canputnext, bcanputnext – test for room in next module’s message queue

**SYNOPSIS**
```c
#include <sys/stream.h>
int canputnext(queue_t *q);
int bcanputnext(queue_t *q, unsigned char pri);
```

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

**ARGUMENTS**
- `q` Pointer to a message queue belonging to the invoking module.
- `pri` Minimum priority level.

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

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

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

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

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

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

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

**SEE ALSO**
- `bcanput(9F), canput(9F)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
NAME     clrbuf – erase the contents of a buffer

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

void clrbuf(struct buf *bp);

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

ARGUMENTS  bp  Pointer to the buf(9S) structure.

DESCRIPTION  clrbuf() zeros a buffer and sets the b_resid member of the buf(9S) structure to 0. Zeros are placed in the buffer starting at bp->b_un.b_addr for a length of bp->b_bcount bytes. b_un.b_addr and b_bcount are members of the buf(9S) data structure.

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

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

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

SYNOPSIS  
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
void cmn_err(int level, char *format, ...);
#include <sys/varargs.h>
void vcmn_err(int level, char *format, va_list ap);

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

ARGUMENTS  
cmn_err()  
level  A constant indicating the severity of the error condition.
format  The message to be displayed.

vcmn_err()  
level  A constant indicating the severity of the error condition.
format  The message to be displayed.
ap  The variable argument list passed to the function.

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

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

CE_CONT  Used to continue another message or to display an informative message not associated with an error. Note that multiple CE_CONT messages without a newline may not appear on the system console or in the system buffer 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 buffer.

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

- An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.
- An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (ll) before any other conversion character is ignored.
- A character indicating the type of conversion to be applied:
  - d, D, o, O, x, X, u
    The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X), or unsigned decimal (u), respectively, and displayed. The letters abcdef are used for x and X conversion.
  - c
    The character value of the argument is displayed.
  - b
    The %b conversion specification allows bit values to be displayed meaningfully. Each %b takes an integer value and a format string from the argument list. The first character of the format string should be the output base encoded as a control character. This base is used to display the integer argument. The remaining groups of characters in the format string consist of a bit number (between 1 and 32, also encoded as a control character) and the next characters (up to the next control character or \'\0\') give the name of the bit field. The string corresponding to the bit fields set in the integer argument is displayed after the numerical value. See the EXAMPLES section.
  - s
    The argument is taken to be a string (character pointer), and characters from the string are displayed until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.
  - %
    Copy a %; no argument is converted.
The first character in `format` affects where the message will be written:

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

To display the contents of the system buffer, use the `dmesg(1M)` command.

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

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

**RETURN VALUES**

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

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

**CONTEXT**

`cmn_err()` can be called from user or kernel context.

**EXAMPLES**

This first example shows how `cmn_err()` can record tracing and debugging information only in the system buffer (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 buffer (line 28).

```c
1 struct reg {
2   uchar_t data;
3   uchar_t csr;
4 };  
5
6 struct xxstate {
7   ...
8   dev_info_t *dip;
9   struct reg *regp;
10  ...
11 };  
12
13 dev_t dev;
14 struct xxstate *xsp;
15  ...
16 #ifdef DEBUG  /* in debugging mode, log function call */
17    cmn_err(CE_CONT, ""%,d: xxopen function called.",
18              ddi_binding_name(xsp->dip), getminor(dev));
```
# Kernel Functions for Drivers

```c
19  #endif /* end DEBUG */
20  …
21 /∗ display device power failure on system console ∗/
22  if ((xsp->regp->csr & POWER) == OFF)
23    cmn_err(CE_NOTE, "%s%d: xxopen: Power is 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));

The second 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");

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

    reg=0xd<Intr,,Enable>

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 buffer and on the system console. Note the use of vsprintf() to construct the error message before calling cmn_err().

```
cmn_err(9F)

Pass formatted string to cmn_err(9F):

```c
#include <cmn_err.h>

int cmn_err(int level, const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    return vfprintf(stderr, fmt, ap);
}
```

SEE ALSO
dmesg(1M), kernel(1M), printf(3S), 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
At times, a driver may encounter error conditions requiring the attention of a primary or secondary system console monitor. These conditions may mean halting multiuser processing; however, this must be done with caution. Except during the debugging stage, a driver should never stop the system.

See the “Debugging” chapter in Writing Device Drivers.

BUGS
cmn_err() does not provide all of the functionality provided by printf(3S).
Kernel Functions for Drivers

condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS

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

void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
int cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
int cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

- `cvp` A pointer to an abstract data type `kcondvar_t`.
- `mp` A pointer to a mutual exclusion lock (kmutex_t), initialized by `mutex_init(9F)` and held by the caller.
- `name` Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)
- `type` The constant `CV_DRIVER`.
- `arg` A type-specific argument, drivers should pass arg as `NULL`.
- `timeout` A time, in absolute ticks since boot, when `cv_timedwait()` or `cv_timedwait_sig()` should return.

DESCRIPTION

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

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

modified 7 May 1997

SunOS 5.6

9F-77
condvar (9F)  Kernel Functions for Drivers

**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 **drv_getparm(9F)** with the argument **LBOLT**.

### RETURN VALUES

0  For **cv_wait_sig()** and **cv_timedwait_sig()** indicates that the condition was not necessarily signaled and the function returned because a signal (as in **kill(2)**) was pending.

-1  For **cv_timedwait()** and **cv_timedwait_sig()** indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.

> 0  For **cv_wait_sig()**, **cv_timedwait()** or **cv_timedwait_sig()** indicates that the condition was met and the function returned due to a call to **cv_signal()** or **cv_broadcast()**.

### 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

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);

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);

SEE ALSO  kill(2), drv_getparm(9F), mutex(9F), mutex_init(9F)

Writing Device Drivers
NAME  
copyb – copy a message block

SYNOPSIS

```c
#include <sys/stream.h>
mblk_t *copyb(mblk_t *bp);
```

INTERFACE

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

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

DESCRIPTION

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

RETURN VALUES

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

CONTEXT

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

EXAMPLES

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

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

```c
struct retrans {
    mblk_t *r_mp;
    int r_address;
    queue_t *r_outq;
    struct retrans *r_next;
} ;

struct protoheader {
    ...
    int h_address;
    ...
} ;

mblk_t *header;

void retransmit(struct retrans *ret)
{
    mblk_t *bp, *mp;
}
```

struct protoheader *php;

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

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

Writing Device Drivers
STREAMS Programming Guide
NAME  
copyin – copy data from a user program to a driver buffer

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

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

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

ARGUMENTS  
userbuf  
User program source address from which data is transferred.

driverbuf  
Driver destination address to which data is transferred.

cn  
Number of bytes transferred.

DESCRIPTION  
copyin() copies data from a user program source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address. Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move according to address alignment.

RETURN VALUES  
Under normal conditions a 0 is returned indicating a successful copy. Otherwise, a -1 is returned if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

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

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

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

```c
1 struct device { /* layout of physical device registers */
2   int control; /* physical device control word */
3   int status; /* physical device status word */
4   short recv_char; /* receive character from device */
5   short xmit_char; /* transmit character to device */
6   
7};
8 extern struct device xx_addr[]; /* phys. device regs. location */
9 ... 
```

9F-82 SunOS 5.6 modified 1 May 1996
10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,
11 cred_t *cred_p, int *rval_p)
12 {
13     register struct device *rp = &xx_addr[getminor(dev) >> 4];
14     switch (cmd) {
15         case XX_SETREGS: /* copy device regs. to user program */
16             if (copyin(arg, rp, sizeof(struct device)))
17                 return(EFAULT);
18             break;
19         ...  /* ... */  
20     }  /* ... */
21     ...  /* ... */
22 }  /* ... */

SEE ALSO 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.

modified 1 May 1996 SunOS 5.6 9F-83
NAME  
copymsg – copy a message

SYNOPSIS  
#include <sys/stream.h>
mblk_t *copymsg(mblk_t *mp);

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

ARGUMENTS  
mp  
Pointer to the message to be copied.

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

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

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

EXAMPLES  
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).

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

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

Writing Device Drivers
STREAMS Programming Guide
NAME copyout – copy data from a driver to a user program

SYNOPSIS

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

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

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

ARGUMENTS

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

DESCRIPTION `copyout()` copies data from driver buffers to user data space. Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

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

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

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

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

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

```c
1 struct device { /* layout of physical device registers */
2   int control; /* physical device control word */
3   int status; /* physical device status word */
4   short recv_char; /* receive character from device */
5   short xmit_char; /* transmit character to device */
6 };
7
8 extern struct device xx_addr[]; /* phys. device regs. location */
9 . . .
10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,
11    cred_t *cred_p, int *rval_p)
```
12  ...
13  {
14    register struct device *rp = &xx_addr[getminor(dev) >> 4];
15    switch (cmd) {
16      case XX_GETREGS: /* copy device regs. to user program */
17        if (copyout(rp, arg, sizeof(struct device)))
18          return(EFAULT);
19        break;
20      ...
21      ...
22    }
23    ...
24  }

SEE ALSO 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.
NAME     csx_AccessConfigurationRegister – read or write a PC Card Configuration Register

SYNOPSIS  
#include <sys/pccard.h>

int32_t csx_AccessConfigurationRegister(client_handle_t ch, access_config_reg_t *acr);

INTERFACE  
LEVEL       Solaris DDI Specific (Solaris DDI)

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

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

STRUCTURE  
MEMBERS     The structure members of access_config_reg_t are:

uint32_t  Socket;  /* socket number */
uint32_t  Action;  /* register access operation */
uint32_t  Offset;  /* config register offset */
uint32_t  Value;  /* value read or written */

The fields are defined as follows:

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

Action     May be set to CONFIG_REG_READ or CONFIG_REG_WRITE. All other values in the Action field are reserved for future use. If the Action field is set to CONFIG_REG_WRITE, the Value field is written to the specified configuration register. Card Services does not read the configuration register after a write operation. For that reason, the Value field is only updated by a CONFIG_REG_READ request.

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

Value      Contains the value read from the PC Card Configuration Register for a read operation. For a write operation, the Value field contains the value to write to the configuration register. As noted above, on return from a write request, the Value field is the value written to the PC Card and not any changed value that may have resulted from the write request (that is, no read after write is performed).

A client must be very careful when writing to the COR (Configuration Option Register) at offset 0. This has the potential to change the type of interrupt request generated by the PC Card or place the card in the reset state. Either request may have undefined results. The client should read the register to determine the appropriate setting for the interrupt mode (Bit 6) before writing to the register.
If a client wants to reset a PC Card, the `csx_ResetFunction(9F)` function should be used. Unlike `csx_AccessConfigurationRegister()`, the `csx_ResetFunction(9F)` function generates a series of event notifications to all clients using the PC Card, so they can re-establish the appropriate card state after the reset operation is complete.

**RETURN VALUES**

- **CS_SUCCESS**
  Successful operation.

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

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

- **CS_BAD_HANDLE**
  Client handle is invalid.

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

- **CS_UNSUPPORTED_FUNCTION**
  No PCMCIA hardware installed.

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

- `csx_ParseTuple(9F)`, `csx_RegisterClient(9F)`, `csx_RequestConfiguration(9F)`, `csx_ResetFunction(9F)`
- *PC Card 95 Standard, PCMCIA/JEIDA*
NAME  csx_ConvertSize – convert device sizes

SYNOPSIS  
```
#include <sys/pccard.h>
int32_t csx_ConvertSize(convert_size_t *cs);
```  

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  
- `cs`  Pointer to a `convert_size_t` structure.

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

STRUCTURE MEMBERS  The structure members of `convert_size_t` are:

- `uint32_t Attributes;`
- `uint32_t bytes;`
- `uint32_t devsize;`

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>This is a bit-mapped field that identifies the type of size conversion to be performed. The field is defined as follows:</td>
</tr>
<tr>
<td></td>
<td><strong>CONVERT_BYTES_TO_DEVSIZE</strong> Converts bytes to devsize format</td>
</tr>
<tr>
<td></td>
<td><strong>CONVERT_DEVSIZE_TO_BYTES</strong> Converts devsize format to bytes</td>
</tr>
<tr>
<td>bytes</td>
<td>If <code>CONVERT_BYTES_TO_DEVSIZE</code> is set, the value in the <code>bytes</code> field is converted to a <code>devsize</code> format and returned in the <code>devsize</code> field.</td>
</tr>
<tr>
<td>devsize</td>
<td>If <code>CONVERT_DEVSIZE_TO_BYTES</code> is set, the value in the <code>devsize</code> field is converted to a <code>bytes</code> value and returned in the <code>bytes</code> field.</td>
</tr>
</tbody>
</table>

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

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

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

`PC Card 95 Standard, PCMCIA/JEIDA`
NAME  csx_ConvertSpeed – convert device speeds

SYNOPSIS  #include <sys/pccard.h>
            int32_t csx_ConvertSpeed(convert_speed_t *cs);

INTERFACE
LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  cs  Pointer to a convert_speed_t structure.

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

STRUCTURE
MEMBERS  The structure members of convert_speed_t are:
            uint32_t Attributes;
            uint32_t nS;
            uint32_t devspeed;

            The fields are defined as follows:
            Attributes  This is a bit-mapped field that identifies the type of speed conversion to be
                        performed. The field is defined as follows:
                        | CONVERT_NS_TO_DEVSPD  | Converts nS to devspeed format
                        | CONVERT_DEVSPD_TO_NS  | Converts devspeed format to nS

            nS  If CONVERT_NS_TO_DEVSPD is set, the value in the nS field is con-
                verted to a devspeed format and returned in the devspeed field.

            devspeed  If CONVERT_DEVSPD_TO_NS is set, the value in the devspeed field is
                      converted to an nS value and returned in the nS field.

RETURN VALUES  CS_SUCCESS  Successful operation.
               CS_BAD_SPEED  Invalid nS or devspeed.
               CS_BAD_ATTRIBUTE  Bad Attributes value.
               CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

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

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

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996  SunOS 5.6  9F-91
NAME  csx_CS_DDI_Info – obtain DDI information

SYNOPSIS  
#include <sys/pccard.h>

int32_t csx_CS_DDI_Info(cs_ddi_info_t *cdi);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  
*cdi  Pointer to a cs_ddi_info_t structure.

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

STRUCTURE MEMBERS  
The structure members of cs_ddi_info_t are:

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

The fields are defined as follows:
- **Socket**  This field must be set to the physical socket number that the client is interested in getting information about.
- **driver_name**  This field must be set to a string containing the name of the client driver to get information about.

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

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

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

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

EXAMPLES  
The following example shows how a client might call csx_CS_DDI_Info() in the client’s xx_getinfo function to return the dip or the instance number:
static int
pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
               void **result)
{
    int error = DDI_SUCCESS;
    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

modified 19 Jul 1996  SunOS 5.6  9F-93
<table>
<thead>
<tr>
<th><strong>NAME</strong></th>
<th>csx_DeregisterClient – remove client from Card Services list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYNOPSIS</strong></td>
<td><code>#include &lt;sys/pccard.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int32_t csx_DeregisterClient(client_handle_t ch);</code></td>
</tr>
<tr>
<td><strong>INTERFACE LEVEL</strong></td>
<td>Solaris DDI Specific (Solaris DDI)</td>
</tr>
<tr>
<td><strong>ARGUMENTS</strong></td>
<td><code>ch</code> Client handle returned from <code>csx_RegisterClient(9F)</code>.</td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
<td>This function removes a client from the list of registered clients maintained by Card Services. The Client Handle returned by <code>csx_RegisterClient(9F)</code> is passed in the <code>client_handle_t</code> argument. The client must have returned all requested resources before this function is called. If any resources have not been released, <code>CS_IN_USE</code> is returned.</td>
</tr>
<tr>
<td><strong>RETURN VALUES</strong></td>
<td><code>CS_SUCCESS</code> Successful operation.</td>
</tr>
<tr>
<td></td>
<td><code>CS_BAD_HANDLE</code> Client handle is invalid.</td>
</tr>
<tr>
<td></td>
<td><code>CS_IN_USE</code> Resources not released by this client.</td>
</tr>
<tr>
<td></td>
<td><code>CS_UNSUPPORTED_FUNCTION</code> No PCMCIA hardware installed.</td>
</tr>
<tr>
<td><strong>CONTEXT</strong></td>
<td>This function may be called from user or kernel context.</td>
</tr>
<tr>
<td><strong>SEE ALSO</strong></td>
<td><code>csx_RegisterClient(9F)</code></td>
</tr>
<tr>
<td></td>
<td><code>PC Card 95 Standard, PCMCIA/JEIDA</code></td>
</tr>
<tr>
<td><strong>WARNINGS</strong></td>
<td>Clients should be prepared to receive callbacks until Card Services returns from this request successfully.</td>
</tr>
</tbody>
</table>

9F-94 SunOS 5.6 modified 19 Jul 1996
NAME  csx_DupHandle – duplicate access handle

SYNOPSIS  
#include <sys/pccard.h>

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

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

handle1  The access handle returned from csx_RequestIO(9F) or csx_RequestWindow(9F) that is to be duplicated.

handle2  A pointer to the newly-created duplicated data access handle.

flags  The access attributes that will be applied to the new handle.

DESCRIPTION  
This function duplicates the handle, handle1, into a new handle, handle2, that has the access attributes specified in the flags argument. Both the original handle and the new handle are active and can be used with the common access functions. Both handles must be explicitly freed when they are no longer necessary.

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

<table>
<thead>
<tr>
<th>WIN_ACC_NEVER_SWAP</th>
<th>Host endian byte ordering</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

WIN_ACC_BIG_ENDIAN and WIN_ACC_LITTLE_ENDIAN describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When WIN_ACC_BIG_ENDIAN or WIN_ACC_LITTLE_ENDIAN is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

When WIN_ACC NEVER_SWAP is specified, byte swapping will not be invoked in the data access functions.
The ability to specify the order in which the CPU will reference data is provided by the following flags bits. Only one of the following bits may be specified:

**WIN_ACC_STRICT_ORDER**
The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

**WIN_ACC_UNORDERED_OK**
The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

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

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

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

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

**RETURN VALUES**

- **CS_SUCCESS**: Successful operation.
- **CS_FAILURE**: Error in flags argument or handle could not be duplicated for some reason.
- **CS_UNSUPPORTED_FUNCTION**: No PCMCIA hardware installed.

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

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

*PC Card 95 Standard, PCMCIA/JEIDA*

9F-96
SunOS 5.6
modified 19 Jul 1996
NAME  csx_Error2Text – convert error return codes to text strings

SYNOPSIS  

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

int32_t csx_Error2Text(error2text_t *er);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

*er*  Pointer to an *error2text_t* structure.

DESCRIPTION  

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

STRUCTURE MEMBERS  

The structure members of *error2text_t* are:

```c
  uint32_t item; /* the error code */
  char text[CS_ERROR_MAX_BUFSIZE]; /* the error text */
```

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

"{unknown Card Services return code}"

RETURN VALUES  

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

CONTEXT  

This function may be called from user or kernel context.

EXAMPLES  

Sample code illustrating the usage of this function:

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

SEE ALSO  

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

modified 19 Jul 1996  SunOS 5.6  9F-97
NAME       csx_Event2Text – convert events to text strings

SYNOPSIS   #include <sys/pccard.h>
            int32_t csx_Event2Text(event2text_t *ev);

INTERFACE  Solaris DDI Specific (Solaris DDI)

LEVEL      ARGUMENTS       ev   Pointer to an event2text_t structure.

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

STRUCTURE  The structure members of event2text_t are:

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

MEMBERS    The fields are defined as follows:

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

RETURN VALUES

            CS_SUCCESS  Successful operation.
            CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

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

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

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

SEE ALSO   csx_event_handler(9E), csx_Error2Text(9F)

PC Card 95 Standard, PCMCIA/JEIDA
### NAME
csx_FreeHandle – free access handle

### SYNOPSIS
```
#include <sys/pccard.h>
int32_t csx_FreeHandle(acc_handle_t *handle);
```

### INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

### ARGUMENTS
- `handle` The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.

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

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

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

### SEE ALSO
- `csx_DupHandle(9F)`, `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`
- *PC Card 95 Standard*, PCMCIA/JEIDA

---

modified 19 Jul 1996

SunOS 5.6

9F-99
**NAME**

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

**SYNOPSIS**

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

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

**INTERFACE LEVEL**

Solaris DDI Specific (Solaris DDI)

**ARGUMENTS**

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

**DESCRIPTION**

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

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

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

**RETURN VALUES**

These functions return the value read from the mapped address.

**CONTEXT**

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

**SEE ALSO**

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

`PC Card 95 Standard`, `PCMCIA/JEIDA`
NAME     csx_GetFirstClient, csx_GetNextClient – return first or next client

SYNOPSIS #include <sys/pccard.h>
            int32_t csx_GetFirstClient(get_firstnext_client_t *fnc);
            int32_t csx_GetNextClient(get_firstnext_client_t *fnc);

INTERFACE LEVEL Solaris DDI Specific (Solaris DDI)

ARGUMENTS fnc Pointer to a get_firstnext_client_t structure.

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

STRUCTURE MEMBERS The structure members of get_firstnext_client_t are:
uint32_t Socket; /**< socket number */
uint32_t Attributes; /**< attributes */
client_handle_t client_handle; /**< client handle */
uint32_t num_clients; /**< number of clients */

The fields are defined as follows:
Socket     If the CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY attribute is set,
            return information only on the PC card installed in this socket.
Attributes This field indicates the type of client. The field is bit-mapped; the fol-
            lowing bits are defined:

<table>
<thead>
<tr>
<th>Stored Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS</td>
<td>Return information on all clients</td>
</tr>
<tr>
<td>CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY</td>
<td>Return client information for the specified socket only</td>
</tr>
</tbody>
</table>

client_handle The client handle of the PC card driver is returned in this field.
num_clients   The number of clients is returned in this field.

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

CONTEXT This function may be called from user or kernel context.
<table>
<thead>
<tr>
<th>SEE ALSO</th>
<th>csx_event_handler(9E)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>PC Card 95 Standard, PCMCIA/JEIDA.</em></td>
</tr>
</tbody>
</table>

9F-102       SunOS 5.6       modified 16 May 1997
NAME

csx_GetFirstTuple, csx_GetNextTuple – return Card Information Structure tuple

SYNOPSIS

#include <sys/pccard.h>

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

INTERFACE

LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch  Client handle returned from csx_RegisterClient(9F).
tu  Pointer to a tuple_t structure.

DESCRIPTION

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

STRUCTURE

MEMBERS

The structure members of tuple_t are:

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

The fields are defined as follows:

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

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

<table>
<thead>
<tr>
<th>TUPLE_RETURN_LINK</th>
<th>TUPLE_RETURN_IGNORED_TUPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return link tuples if set</td>
<td>Return ignored tuples if set</td>
</tr>
</tbody>
</table>

The following are link tuples and will not be returned by this function unless the TUPLE_RETURN_LINK bit in the Attributes field is set:

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

Ignored tuples will not be returned by this function unless the TUPLE_RETURN_IGNORED_TUPLES bit in the Attributes field is set (see tuple(9S)).

The CIS is parsed from the location setup by the previous csx_GetFirstTuple() or csx_GetNextTuple() request.

modified 20 Dec 1996  SunOS 5.6  9F-103
DesiredTuple

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

TupleCode

TupleLink

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

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

RETURN VALUES

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

CONTEXT

These functions may be called from user or kernel context.

SEE ALSO

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

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

modified 16 May 1997

SunOS 5.6

9F-105
### NAME
`csx_GetMappedAddr` – return mapped virtual address

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

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

### INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

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

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

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

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

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

*PC Card 95 Standard, PCMCIA/JEIDA*
NAME
csx_GetStatus – return the current status of a PC Card and its socket

SYNOPSIS
#include <sys/pccard.h>
int32_t csx_GetStatus(client_handle_t ch, get_status_t *gs);

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

ARGUMENTS
ch Client handle returned from csx_RegisterClient(9F).
gs Pointer to a get_status_t structure.

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

STRUCTURE MEMBERS
The structure members of get_status_t are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.</td>
</tr>
<tr>
<td>CardState</td>
<td>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:</td>
</tr>
<tr>
<td>CS_STATUS_WRITE_PROTECTED</td>
<td>Card is write protected</td>
</tr>
<tr>
<td>CS_STATUS_CARD_LOCKED</td>
<td>Card is locked</td>
</tr>
<tr>
<td>CS_STATUS_EJECTION_REQUEST</td>
<td>Ejection request in progress</td>
</tr>
<tr>
<td>CS_STATUS_INSERTION_REQUEST</td>
<td>Insertion request in progress</td>
</tr>
<tr>
<td>CS_STATUS_BATTERY_DEAD</td>
<td>Card battery is dead (BVD1)</td>
</tr>
<tr>
<td>CS_STATUS_BATTERY_LOW</td>
<td>Card battery is low (BVD2)</td>
</tr>
<tr>
<td>CS_STATUS_CARD_READY</td>
<td>Card is READY</td>
</tr>
<tr>
<td>CS_STATUS_CARD_INSERTED</td>
<td>Card is inserted</td>
</tr>
<tr>
<td>CS_STATUS_REQ_ATTN</td>
<td>Extended status attention request</td>
</tr>
<tr>
<td>CS_STATUS_RES_EVT1</td>
<td>Extended status reserved event status</td>
</tr>
<tr>
<td>CS_STATUS_RES_EVT2</td>
<td></td>
</tr>
<tr>
<td>CS_STATUS_RES_EVT3</td>
<td></td>
</tr>
</tbody>
</table>

modified 19 Jul 1996
SunOS 5.6
Note: 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:

<table>
<thead>
<tr>
<th>Bit Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_STATUS_VCC_50</td>
<td>5.0 Volts Vcc Indicated</td>
</tr>
<tr>
<td>CS_STATUS_VCC_33</td>
<td>3.3 Volts Vcc Indicated</td>
</tr>
<tr>
<td>CS_STATUS_VCC_XX</td>
<td>X.X Volts Vcc Indicated</td>
</tr>
</tbody>
</table>

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

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

The state reported in the SocketState field may be different from the state reported in the CardState field.

Clients should normally depend only on the state reported in the CardState field.
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_BAD_SOCKET</td>
<td>Error getting socket state.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

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

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

SEE ALSO  csx_RegisterClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996
NAME  csx_GetTupleData – return the data portion of a tuple

SYNOPSIS  
#include <sys/pccard.h>

int32_t csx_GetTupleData(client_handle_t ch, tuple_t *tu);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  
- *ch*  Client handle returned from csx_RegisterClient(9F).
- *tu*  Pointer to a tuple_t structure.

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

STRUCTURE MEMBERS  
The structure members of tuple_t are:

- 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 */

The fields are defined as follows:

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

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

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

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

- **TupleDataMax**  This field is the size of the tuple data buffer that Card Services uses to return raw tuple data from csx_GetTupleData(9F). It can be larger than the number of bytes in the tuple data body. Card Services ignores any value placed here by the client.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TupleDataLen</td>
<td>This field is the actual size of the tuple data body. It represents the number of tuple data body bytes returned.</td>
</tr>
<tr>
<td>TupleData</td>
<td>This field is an array of bytes containing the raw tuple data body contents.</td>
</tr>
<tr>
<td>TupleCode</td>
<td>Initialized by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code>; the client must not modify the value in this field.</td>
</tr>
<tr>
<td>TupleLink</td>
<td>Initialized by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code>; the client must not modify the value in this field.</td>
</tr>
</tbody>
</table>

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_BAD_ARGS</td>
<td>Data from prior <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> is corrupt.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_NO_MORE_ITEMS</td>
<td>Card Services was not able to read the tuple from the PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

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

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

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

## SYNOPSIS

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

int32_t csx_MakeDeviceNode(client_handle_t ch, make_device_node_t *dn);
int32_t csx_RemoveDeviceNode(client_handle_t ch, remove_device_node_t *dn);
```

## INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

## ARGUMENTS

- **ch**: Client handle returned from `csx_RegisterClient(9F)`.
- **dn**: Pointer to a `make_device_node_t` or `remove_device_node_t` structure.

## DESCRIPTION

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

## STRUCTURE MEMBERS

### make_device_node_t

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

### remove_device_node_t

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

### devnode_desc_t

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

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

The following `Action` values are defined for `csx_MakeDeviceNode()`:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_DEVICE_NODE</td>
<td>Create <code>NumDevNodes</code> minor nodes</td>
</tr>
</tbody>
</table>

The following `Action` values are defined for `csx_RemoveDeviceNode()`:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMOVE_DEVICE_NODE</td>
<td>Remove <code>NumDevNodes</code> minor nodes</td>
</tr>
<tr>
<td>REMOVE_ALL_DEVICE_NODES</td>
<td>Remove all minor nodes for this client</td>
</tr>
</tbody>
</table>
For \texttt{csx\_MakeDeviceNode()}, if the \texttt{Action} field is:

\begin{itemize}
\item \textbf{CREATE\_DEVICE\_NODE} \\
\hspace{1em} The \texttt{NumDevNodes} field must be set to the number of minor devices to create, and the client must allocate the quantity of \texttt{devnode\_desc\_t} structures specified by \texttt{NumDevNodes} and fill out the fields in the \texttt{devnode\_desc\_t} structure with the appropriate minor node information.
\hspace{1em} The meanings of the fields in the \texttt{devnode\_desc\_t} structure are identical to the parameters of the same name to the \texttt{ddi\_create\_minor\_node(9F)} DDI function.
\end{itemize}

For \texttt{csx\_RemoveDeviceNode()}, if the \texttt{Action} field is:

\begin{itemize}
\item \textbf{REMOVE\_DEVICE\_NODE} \\
\hspace{1em} The \texttt{NumDevNodes} field must be set to the number of minor devices to remove, and the client must allocate the quantity of \texttt{devnode\_desc\_t} structures specified by \texttt{NumDevNodes} and fill out the fields in the \texttt{devnode\_desc\_t} structure with the appropriate minor node information.
\hspace{1em} The meanings of the fields in the \texttt{devnode\_desc\_t} structure are identical to the parameters of the same name to the \texttt{ddi\_remove\_minor\_node(9F)} DDI function.
\item \textbf{REMOVE\_ALL\_DEVICE\_NODES} \\
\hspace{1em} The \texttt{NumDevNodes} field must be set to 0 and the \texttt{devnode\_desc\_t} structure pointer must be set to NULL. All device nodes for this client will be removed from the filesystem.
\end{itemize}

\textbf{RETURN VALUES} \\
\texttt{CS\_SUCCESS} \hspace{1em} Successful operation. \\
\texttt{CS\_BAD\_HANDLE} \hspace{1em} Client handle is invalid. \\
\texttt{CS\_BAD\_ATTRIBUTE} \hspace{1em} The value of one or more arguments is invalid. \\
\texttt{CS\_BAD\_ARGS} \hspace{1em} Action is invalid. \\
\texttt{CS\_OUT\_OF\_RESOURCE} \hspace{1em} Unable to create or remove device node. \\
\texttt{CS\_UNSUPPORTED\_FUNCTION} \hspace{1em} No PCMCIA hardware installed.

\textbf{CONTEXT} \\
These functions may be called from user or kernel context.

\textbf{SEE ALSO} \texttt{csx\_RegisterClient(9F), ddi\_create\_minor\_node(9F), ddi\_remove\_minor\_node(9F)} \textit{PC Card 95 Standard, PCMCIA/JEIDA}
**NAME**  
`csx_MapLogSocket` – return the physical socket number associated with the client handle

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

int32_t csx_MapLogSocket(client_handle_t ch, map_log_socket_t *ls);
```

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

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

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

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

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

The fields are defined as follows:

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

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

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

**RETURN VALUES**  
- **CS_SUCCESS**  
  Successful operation.
- **CS_BAD_HANDLE**  
  Client handle is invalid.
- **CS_UNSUPPORTED_FUNCTION**  
  No PCMCIA hardware installed.

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

**SEE ALSO**  
- `csx_RegisterClient(9F)`
- *PC Card 95 Standard, PCMCIA/JEIDA*
**NAME**  
`csx_MapMemPage` – map the memory area on a PC Card

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

int32_t csx_MapMemPage(window_handle_t *wh, map_mem_page_t *mp);
```

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

**ARGUMENTS**  
- `wh`  
  Window handle returned from `csx_RequestWindow(9F)`.
- `mp`  
  Pointer to a `map_mem_page_t` structure.

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

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

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

The fields are defined as follows:
- **CardOffset**  
  The absolute offset in bytes from the beginning of the PC Card to map into system memory.
- **Page**  
  Used internally by Card Services; clients must set this field to 0 before calling this function.

**RETURN VALUES**  
- **CS_SUCCESS**  
  Successful operation.
- **CS_BAD_HANDLE**  
  Client handle is invalid.
- **CS_BAD_OFFSET**  
  Offset is invalid.
- **CS_BAD_PAGE**  
  Page is not zero.
- **CS_NO_CARD**  
  No PC Card in socket.
- **CS_UNSUPPORTED_FUNCTION**  
  No PCMCIA hardware installed.

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

**SEE ALSO**  
- `csx_ModifyWindow(9F)`, `csx_ReleaseWindow(9F)`, `csx_RequestWindow(9F)`
- *PC Card 95 Standard*, PCMCIA/JEIDA

---

modified 19 Jul 1996

SunOS 5.6

9F-115
NAME  csx_ModifyConfiguration – modify socket and PC Card Configuration Register

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

int32_t csx_ModifyConfiguration(client_handle_t ch, modify_config_t *mc);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  
- `ch`  Client handle returned from `csx_RegisterClient(9F)`.
- `mc`  Pointer to a `modify_config_t` structure.

DESCRIPTION  This function allows a socket and PC Card configuration to be modified. This function can only modify a configuration requested via `csx_RequestConfiguration(9F)`.

STRUCTURE MEMBERS  The structure members of `modify_config_t` are:
- `uint32_t Socket; /* socket number */`
- `uint32_t Attributes; /* attributes to modify */`
- `uint32_t Vpp1; /* Vpp1 value */`
- `uint32_t Vpp2; /* Vpp2 value */`

The fields are defined as follows:

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

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

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONF_ENABLE_IRQ_STEERING</td>
<td>Enable IRQ Steering</td>
</tr>
<tr>
<td>CONF_IRQ_CHANGE_VALID</td>
<td>IRQ change valid</td>
</tr>
<tr>
<td>CONF_VPP1_CHANGE_VALID</td>
<td>Vpp1 change valid</td>
</tr>
<tr>
<td>CONF_VPP2_CHANGE_VALID</td>
<td>Vpp2 change valid</td>
</tr>
<tr>
<td>CONF_VSOVERRIDE</td>
<td>Override VS pins</td>
</tr>
</tbody>
</table>

- **CONF_ENABLE_IRQ_STEERING**
  Set to connect the PC Card IREQ line to a previously selected system interrupt.

- **CONF_IRQ_CHANGE_VALID**
  Set to request the IRQ steering enable to be changed.

- **CONF_VPP1 CHANGE_VALID**
  These bits are set to request a change to the corresponding voltage level for the PC Card.

- **CONF_VPP2 CHANGE_VALID**
  These bits are set to request a change to the corresponding voltage level for the PC Card.
CONF_VSOVERRIDE
For Low Voltage keyed cards, must be set if a client desires to apply a voltage inappropriate for this card to any pin. After card insertion and prior to the first csx_RequestConfiguration(9F) call for this client, the voltage levels applied to the card will be those specified by the Card Interface Specification. (See WARNINGS.)

Vpp1, Vpp2 These fields all represent voltages expressed in tenths of a volt. Values from 0 to 25.5 volts may be set. To be valid, the exact voltage must be available from the system. To be compliant with the PC Card 95 Standard, systems must always support 5.0 volts for both Vcc and Vpp. (See WARNINGS.)

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_HANDLE Client handle is invalid or csx_RequestConfiguration(9F) not done.
CS_BAD_SOCKET Error getting/setting socket hardware parameters.
CS_BAD_VPP Requested Vpp is not available on socket.
CS_NO_CARD No PC Card in socket.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

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

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

PC Card 95 Standard, PCMCIA/JEIDA

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

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

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

IRQ priority can only be changed by first releasing the current configuration and IRQ resources with csx_ReleaseConfiguration(9F) and csx_ReleaseIRQ(9F), requesting new IRQ resources and a new configuration with csx_RequestIRQ(9F), followed by csx_RequestConfiguration(9F).
Vcc can not be changed using `csx_ModifyConfiguration()`. Vcc may be changed by first invoking `csx_ReleaseConfiguration(9F)`, followed by `csx_RequestConfiguration(9F)` with a new Vcc value.
NAME

csx_ModifyWindow – modify window attributes

SYNOPSIS

#include <sys/pccard.h>
int32_t csx_ModifyWindow(window_handle_t wh, modify_win_t *mw);

INTERFACE

LEVEL
Solaris DDI Specific (Solaris DDI)

ARGUMENTS

wh Window handle returned from csx_RequestWindow(9F).

mw Pointer to a modify_win_t structure.

DESCRIPTION

This function modifies the attributes of a window allocated by the

csx_RequestWindow(9F) function.

Only some of the window attributes or the access speed field may be modified by this

request. The csx_MapMemPage(9F) function is also used to set the offset into PC Card

memory to be mapped into system memory for paged windows. The

csx_RequestWindow(9F) and csx_ReleaseWindow(9F) functions must be used to change

the window base or size.

STRUCTURE

MEMBERS

The structure members of modify_win_t are:

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

The fields are defined as follows:

Attributes

This field is bit-mapped and defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIN_MEMORY_TYPE_CM</td>
<td>Window points to Common Memory area</td>
</tr>
<tr>
<td>WIN_MEMORY_TYPE_AM</td>
<td>Window points to Attribute Memory area</td>
</tr>
<tr>
<td>WIN_ENABLE</td>
<td>Enable Window</td>
</tr>
<tr>
<td>WIN_ACCESS_SPEED_VALID</td>
<td>AccessSpeed valid</td>
</tr>
</tbody>
</table>

WIN_MEMORY_TYPE_CM

Set this to map the window to Common Memory.

WIN_MEMORY_TYPE_AM

Set this to map the window to Attribute Memory.

WIN_ENABLE

The client must set this to enable the window.

WIN_ACCESS_SPEED_VALID

The client must set this when the AccessSpeed field has a value that

the client wants set for the window.
AccessSpeed

The bit definitions for this field use the format of the extended speed byte of the Device ID tuple. If the mantissa is 0 (noted as reserved in the PC Card 95 Standard), the lower bits are a binary code representing a speed from the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Reserved - do not use)</td>
</tr>
<tr>
<td>1</td>
<td>250 nsec</td>
</tr>
<tr>
<td>2</td>
<td>200 nsec</td>
</tr>
<tr>
<td>3</td>
<td>150 nsec</td>
</tr>
<tr>
<td>4</td>
<td>100 nsec</td>
</tr>
<tr>
<td>5 - 7</td>
<td>(Reserved - do not use)</td>
</tr>
</tbody>
</table>

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

RETURN VALUES

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

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

- `csx_ModifyWindow(9F)`, `csx_MapMemPage(9F)`, `csx_ReleaseWindow(9F)`, `csx_RequestWindow(9F)`
- PC Card 95 Standard, PCMCIA/JEIDA
### NAME
`csx_Parse_CISTPL_BATTERY` – parse the Battery Replacement Date tuple

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

int32_t csx_Parse_CISTPL_BATTERY(client_handle_t ch, tuple_t *tu,
                                cistpl_date_t *cb);
```

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

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

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

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

### STRUCTURE
**MEMBERS**
The structure members of `cistpl_date_t` are:

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

The fields are defined as follows:
- `rday` This field indicates the date on which the battery was last replaced.
- `xday` This field indicates the date on which the battery should be replaced.

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

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

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

---

modified 20 Dec 1996

SunOS 5.6

9F-121
csx_Parse_CISTPL_BYTEORDER (9F)  --  parse the Byte Order tuple

#include <sys/pccard.h>

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

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

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

The structure members of cistpl_byteorder_t are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>This field specifies the byte order for multi-byte numeric data.</td>
</tr>
<tr>
<td>map</td>
<td>This field specifies the byte mapping for 16-bit or wider cards.</td>
</tr>
</tbody>
</table>

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLBYTEORD_LOW</td>
<td>Little endian order</td>
</tr>
<tr>
<td>TPLBYTEORD_HIGH</td>
<td>Big endian order</td>
</tr>
<tr>
<td>TPLBYTEORD_VS</td>
<td>Vendor specific</td>
</tr>
<tr>
<td>TPLBYTEMAP_LOW</td>
<td>Byte zero is least significant byte</td>
</tr>
<tr>
<td>TPLBYTEMAP_HIGH</td>
<td>Byte zero is most significant byte</td>
</tr>
<tr>
<td>TPLBYTEMAP_VS</td>
<td>Vendor specific mapping</td>
</tr>
</tbody>
</table>

This function returns:

- 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.

9F-122  SunOS 5.6    modified 20 Dec 1996
Kernel Functions for Drivers

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

modified 20 Dec 1996

SunOS 5.6

9F-123
NAME

csx_Parse_CISTPL_CFTABLE_ENTRY – parse 16-bit Card Configuration Table Entry tuple

SYNOPSIS

#include <sys/pccard.h>

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

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch  Client handle returned from csx_RegisterClient(9F).

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

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

DESCRIPTION

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

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

STRUCTURE MEMBERS

The structure members of cistpl_cftable_entry_t are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td>flags</td>
<td>Which descriptions are valid</td>
</tr>
<tr>
<td>uint32_t</td>
<td>ifc</td>
<td>Interface description information</td>
</tr>
<tr>
<td>uint32_t</td>
<td>pin</td>
<td>Values for PRR</td>
</tr>
<tr>
<td>uint32_t</td>
<td>index</td>
<td>Configuration index number</td>
</tr>
<tr>
<td>cistpl_cftable_entry_pd_t</td>
<td>pd</td>
<td>Power requirements description</td>
</tr>
<tr>
<td>cistpl_cftable_entry_speed_t</td>
<td>speed</td>
<td>Device speed description</td>
</tr>
<tr>
<td>cistpl_cftable_entry_io_t</td>
<td>io</td>
<td>Device I/O map</td>
</tr>
<tr>
<td>cistpl_cftable_entry_irq_t</td>
<td>irq</td>
<td>Device IRQ utilization</td>
</tr>
<tr>
<td>cistpl_cftable_entry_mem_t</td>
<td>mem</td>
<td>Device memory space</td>
</tr>
<tr>
<td>cistpl_cftable_entry_misc_t</td>
<td>misc</td>
<td>Miscellaneous device features</td>
</tr>
</tbody>
</table>

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>This field is bit-mapped and defined as follows:</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_DEFAULT</td>
<td>This is a default configuration</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF</td>
<td>If configuration byte exists</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_PWR</td>
<td>Power information exists</td>
</tr>
</tbody>
</table>
ifc

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_MEMORY</td>
<td>Memory interface</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_IO_MEM</td>
<td>IO and memory</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_CUSTOM_0</td>
<td>Custom interface 0</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_CUSTOM_1</td>
<td>Custom interface 1</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_CUSTOM_2</td>
<td>Custom interface 2</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_CUSTOM_3</td>
<td>Custom interface 3</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_MASK</td>
<td>Interface type mask</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_BVD</td>
<td>BVD active in PRR</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_WP</td>
<td>WP active in PRR</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_RDY</td>
<td>RDY active in PRR</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_IF_MWAIT</td>
<td>WAIT - mem cycles</td>
</tr>
</tbody>
</table>

pin

This is a value for the Pin Replacement Register.

index

This 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 */
```

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_PWR_VCC</td>
<td>Vcc description valid</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_PWR_VPP1</td>
<td>Vpp1 description valid</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_PWR_VPP2</td>
<td>Vpp2 description valid</td>
</tr>
</tbody>
</table>
The structure members of \texttt{cistpl_cftable_entry_pwr_t} are:

\begin{verbatim}
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;
\end{verbatim}

The fields are defined as follows:

- \texttt{nomV}, \texttt{minV_flags}, \texttt{maxV_flags}, \texttt{staticI_flags}, \texttt{avgI}, \texttt{peakI_flags}, \texttt{pdownI_flags}

These fields are bit-mapped and defined as follows:

\begin{table}[h]
\begin{tabular}{|c|l|}
\hline
\texttt{CISTPL\_CFTABLE\_PD\_NOMV} & Nominal supply voltage \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_MINV} & Minimum supply voltage \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_MAXV} & Maximum supply voltage \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_STATICI} & Continuous supply current \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_AVGI} & Maximum current required averaged over 1 second \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_PEAKI} & Maximum current required averaged over 10mS \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_PDOWNI} & Power down supply current required \\
\hline
\end{tabular}
\end{table}

- \texttt{nomV\_flags}, \texttt{minV\_flags}, \texttt{maxV\_flags}, \texttt{staticI\_flags}, \texttt{avgI\_flags}, \texttt{peakI\_flags}, \texttt{pdownI\_flags}

These fields are bit-mapped and defined as follows:

\begin{table}[h]
\begin{tabular}{|c|l|}
\hline
\texttt{CISTPL\_CFTABLE\_PD\_ EXISTS} & This parameter exists \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_MUL10} & Multiply return value by 10 \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_NC\_SLEEP} & No connection on sleep/power down \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_ZERO} & Zero value required \\
\hline
\texttt{CISTPL\_CFTABLE\_PD\_NC} & No connection ever \\
\hline
\end{tabular}
\end{table}
The structure members of `cistpl_cftable_entry_speed_t` are:

```c
uint32_t flags; /* which timing information is present */
uint32_t wait; /* max WAIT time in device speed format */
uint32_t nS_wait; /* max WAIT time in nS */
uint32_t rdybys; /* max RDY/BSY time in device speed format */
uint32_t nS_rdybys; /* 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 fields are defined as follows:

**flags**

This field is bit-mapped and defined as follows:

<table>
<thead>
<tr>
<th>CISTPL_CFTABLE_TPCE_FS_TD_WAIT</th>
<th>WAIT timing exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_TD_RDY</td>
<td>RDY/BSY timing exists</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_TD_RSVD</td>
<td>RSVD timing exists</td>
</tr>
</tbody>
</table>

The structure members of `cistpl_cftable_entry_io_t` are:

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

The fields are defined as follows:

**flags**

This field is bit-mapped and defined as follows:

<table>
<thead>
<tr>
<th>CISTPL_CFTABLE_TPCE_FS_IO_BUS</th>
<th>Bus width mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_IO_BUS8</td>
<td>8-bit flag</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_IO_BUS16</td>
<td>16-bit flag</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_IO_RANGE</td>
<td>I/O address ranges exist</td>
</tr>
</tbody>
</table>

The structure members of `cistpl_cftable_entry_io_range_t` are:

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

The structure members of `cistpl_cftable_entry_irq_t` are:

```c
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:

```c
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 fields are defined as follows:

**flags**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_MEM3</td>
<td>Space descriptors</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_MEM2</td>
<td>host_addr=card_addr</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_MEM1</td>
<td>Card address=0, any host address</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_FS_MEM_HOST</td>
<td>If host address is present in MEM3</td>
</tr>
</tbody>
</table>

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 fields are defined as follows:

**flags**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_MTC_MASK</td>
<td>Max twin cards mask</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_AUDIO</td>
<td>Audio on BVD2</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_READONLY</td>
<td>R/O storage</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_PWRDOWN</td>
<td>Powerdown capable</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DRQ_MASK</td>
<td>DMAREQ mask</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DRQ_SPK</td>
<td>DMAREQ on SPKR</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DRQ_IOIS</td>
<td>DMAREQ on IOIS16</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DRQ_INP</td>
<td>DMAREQ on INPACK</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DMA_8</td>
<td>DMA width 8 bits</td>
</tr>
<tr>
<td>CISTPL_CFTABLE_TPCE_MI_DMA_16</td>
<td>DMA width 16 bits</td>
</tr>
</tbody>
</table>

**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

modified 20 Dec 1996

SunOS 5.6

9F-129
NAME  csx_Parse_CISTPL_CONFIG – parse Configuration tuple

SYNOPSIS  
#include <sys/pccard.h>

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

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

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

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

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

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

STRUCTURE MEMBERS  The structure members of cistpl_config_t are:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>present</td>
<td>register present flags /*</td>
</tr>
<tr>
<td>nr</td>
<td>number of config registers found /*</td>
</tr>
<tr>
<td>hr</td>
<td>highest config register index found /*</td>
</tr>
<tr>
<td>regs</td>
<td>reg offsets /*</td>
</tr>
<tr>
<td>base</td>
<td>base offset of config registers /*</td>
</tr>
<tr>
<td>last</td>
<td>last config index /*</td>
</tr>
</tbody>
</table>

The fields are defined as follows:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>present</td>
<td>This field indicates which configuration registers are present on the PC Card.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_OPTION_REG_PRESENT</td>
<td>Configuration Option Register present</td>
</tr>
<tr>
<td>CONFIG_STATUS_REG_PRESENT</td>
<td>Configuration Status Register present</td>
</tr>
<tr>
<td>CONFIG_PINREPL_REG_PRESENT</td>
<td>Pin Replacement Register present</td>
</tr>
<tr>
<td>CONFIG_COPY_REG_PRESENT</td>
<td>Copy Register present</td>
</tr>
<tr>
<td>CONFIG_EXSTAT_REG_PRESENT</td>
<td>Extended Status Register present</td>
</tr>
<tr>
<td>CONFIG_IOBASE0_REG_PRESENT</td>
<td>IO Base 0 Register present</td>
</tr>
</tbody>
</table>

9F-130 SunOS 5.6 modified 20 Dec 1996
Kernel Functions for Drivers  
csx_Parse_CISTPL_CONFIG (9F)

<table>
<thead>
<tr>
<th>CONFIG_IOBASE1_REG_PRESENT</th>
<th>IO Base 1 Register present</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_IOBASE2_REG_PRESENT</td>
<td>IO Base2 Register present</td>
</tr>
<tr>
<td>CONFIG_IOBASE3_REG_PRESENT</td>
<td>IO Base3 Register present</td>
</tr>
<tr>
<td>CONFIG_IOLIMIT_REG_PRESENT</td>
<td>IO Limit Register present</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

PC Card 95 Standard, PCMCIA/JEIDA

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

modified 20 Dec 1996  SunOS 5.6  9F-131
NAME  csx_Parse_CISTPL_DATE – parse the Card Initialization Date tuple

SYNOPSIS  
#include <sys/pccard.h>

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

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

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

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

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

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

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

STRUCTURE MEMBERS  
The structure members of cistpl_date_t are:

  uint32_t time;
  uint32_t day;

The fields are defined as follows:

time  This field indicates the time at which the PC Card was initialized.

day  This field indicates the date the PC Card was initialized.

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

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

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

PC Card 95 Standard, PCMCIA/JEIDA
Kernel Functions for Drivers

NAME

csx_Parse_CISTPL_DEVICE, csx_Parse_CISTPLDEVICE_A, csx_Parse_CISTPL_DEVICE_OC, csx_Parse_CISTPLDEVICE_OA – parse Device Information tuples

SYNOPSIS

#include <sys/pccard.h>

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

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch Client handle returned from csx_RegisterClient(9F).

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

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

DESCRIPTION

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

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

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

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

STRUCTURE MEMBERS

The structure members of cistpl_device_t are:

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

modified 20 Dec 1996

SunOS 5.6

9F-133
The structure members of *cistpl_device_node_t* are:

```c
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
- **CISTPLDEVICE OC_Vcc_MASK** Mask for Vcc value
- **CISTPLDEVICE OC_Vcc5** 5.0 volt operation
- **CISTPLDEVICE OC_Vcc33** 3.3 volt operation
- **CISTPLDEVICE OC_VccXX** X.X volt operation
- **CISTPLDEVICE OC_VccYY** Y.Y volt operation

**speed**

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

**nS_speed**

The device speed value described in nanosecond units.

**size**

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

**size_in_bytes**

The device size value described in byte units.

**type**

This is the device type code field which is defined as follows:

- **CISTPL_DEVICE_DTYPE_NULL** No device
- **CISTPLDEVICE DTYPE ROM** Masked ROM
- **CISTPLDEVICE DTYPE OTPROM** One Time Programmable ROM
- **CISTPLDEVICE DTYPEEPROM** UV EPROM
- **CISTPLDEVICE DTYPE EEPROM** EEPROM
- **CISTPLDEVICE DTYPE FLASH** FLASH
- **CISTPLDEVICE DTYPE SRAM** Static RAM
<table>
<thead>
<tr>
<th>Kernel Functions for Drivers</th>
<th>csx_Parse_CISTPL_DEVICE (9F)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CISTPL_DEVICE_DTYPE_DRAM</strong></td>
<td>Dynamic RAM</td>
</tr>
<tr>
<td><strong>CISTPLDEVICE_DTYPE_FUNCSPEC</strong></td>
<td>Function-specific memory address range</td>
</tr>
<tr>
<td><strong>CISTPLDEVICE_DTYPE_EXTEND</strong></td>
<td>Extended type follows</td>
</tr>
</tbody>
</table>

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

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

**SEE ALSO**
- `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_Parse_CISTPL_JEDEC_C(9F)`, `csx_RegisterClient(9F)`, `csx.ValidateCIS(9F)`, `tuple(9S)`
- *PC Card 95 Standard, PCMCIA/JEIDA*
NAME | csx_Parse_CISTPL_DEVICEGEO – parse the Device Geo tuple

SYNOPSIS | #include <sys/pccard.h>
int32_t csx_Parse_CISTPL_DEVICEGEO(client_handle_t ch, tuple_t *tp, cistpl_devicegeo_t *pt);

INTERFACE LEVEL | Solaris DDI Specific (Solaris DDI)

ARGUMENTS | ch | Client handle returned from csx_RegisterClient(9F).
tp | Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
pt | Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo tuple information upon return from this function.

DESCRIPTION | This function parses the Device Geo tuple, CISTPL_DEVICEGEO, into a form usable by PC Card drivers.
The CISTPL_DEVICEGEO tuple describes the device geometry of common memory partitions.

STRUCTURE MEMBERS | The structure members of cistpl_devicegeo_t are:
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part;
uint32_t info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil;
The fields are defined as follows:

info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus | This field indicates the card interface width in bytes for the given partition.
info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs | This field indicates the minimum erase block size for the given partition.
info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs | This field indicates the minimum read block size for the given partition.
info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs | This field indicates the minimum write block size for the given partition.
info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part | This field indicates the segment partition subdivisions for the given partition.
info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil | This field indicates the hardware interleave.
<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

csx_GetFirstTuple(9F), csx_GetNextTuple(9F), csx_GetTupleData(9F),
csx_Parse_CISTPLDEVICEGEO_A(9F), csx_RegisterClient(9F), tuple(9S)

*PC Card 95 Standard, PCMCIA/JEIDA*

modified 16 May 1997

SunOS 5.6

9F-137
NAME        csx_Parse_CISTPL_DEVICEGEO_A – parse the Device Geo A tuple  
SYNOPSIS    #include <sys/pccard.h>  
            int32_t csx_Parse_CISTPL_DEVICEGEO_A(client_handle_t ch, tuple_t *tp,  
                                         cistpl_devicegeo_t *pt);  
INTERFACE   Solaris DDI Specific (Solaris DDI)  
LEVEL       
ARGUMENTS   ch          Client handle returned from csx_RegisterClient(9F).  
            tp          Pointer to a tuple_t structure (see tuple(9S)) returned by a call to  
                        csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).  
            pt          Pointer to a cistpl_devicegeo_t structure which contains the parsed Device Geo  
                        A tuple information upon return from this function.  
DESCRIPTION This function parses the Device Geo A tuple, CISTPL_DEVICEGEO_A, into a form usable  
            by PC Card drivers.  
            The CISTPL_DEVICEGEO_A tuple describes the device geometry of attribute memory  
            partitions.  
STRUCTURE   The structure members of cistpl_devicegeo_t are:  
MEMBERS     uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus;  
            uint32_t    info[CISTPLDEVICEGEO_MAX_PARTITIONS].ebs;  
            uint32_t    info[CISTPLDEVICEGEO_MAX_PARTITIONS].rbs;  
            uint32_t    info[CISTPLDEVICEGEO_MAX_PARTITIONS].wbs;  
            uint32_t    info[CISTPLDEVICEGEO_MAX_PARTITIONS].part;  
            uint32_t    info[CISTPLDEVICEGEO_MAX_PARTITIONS].hwil;  
            The fields are defined as follows:  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus  
                        This field indicates the card interface width in bytes for the given partition.  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs  
                        This field indicates the minimum erase block size for the given partition.  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs  
                        This field indicates the minimum read block size for the given partition.  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs  
                        This field indicates the minimum write block size for the given partition.  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part  
                        This field indicates the segment partition subdivisions for the given partition.  
            info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil  
                        This field indicates the hardware interleave for the given partition.
Kernel Functions for Drivers

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

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

**SEE ALSO**
- `csx_GetFirstTuple(9F)`, `csx_GetNextTuple(9F)`, `csx_GetTupleData(9F)`,
- `csx_Parse_CISTPL DEVICEGEO(9F)`, `csx_RegisterClient(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*

modified 16 May 1997

SunOS 5.6

9F-139
NAME

csx_Parse_CISTPL_FORMAT – parse the Data Recording Format tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_FORMAT(client_handle_t ch, tuple_t *tu,
cistpl_format_t *pt);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch Client handle returned from csx_RegisterClient(9F).

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

pt Pointer to a cistpl_format_t structure which contains the parsed
CISTPL_FORMAT tuple information upon return from this function.

DESCRIPTION

This function parses the Data Recording Format tuple, CISTPL_FORMAT, into a form useful by PC Card drivers.

The CISTPL_FORMAT tuple indicates the data recording format for a device partition.

STRUCTURE MEMBERS

The structure members of cistpl_format_t are:

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

The fields are defined as follows:

- type This field indicates the type of device:
  - TPLFMTTYPE_DISK disk-like device
  - TPLFMTTYPE_MEM memory-like device
  - TPLFMTTYPE_VS vendor-specific device

- edc_length This field indicates the error detection code length.

- edc_type This field indicates the error detection code type.

- offset This field indicates the offset of the first byte of data in this partition.

- nbytes This field indicates the number of bytes of data in this partition.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev.disk.bksize</code></td>
<td>This field indicates the block size, for disk devices.</td>
</tr>
<tr>
<td><code>dev.disk.nblocks</code></td>
<td>This field indicates the number of blocks, for disk devices.</td>
</tr>
<tr>
<td><code>dev.disk.edcloc</code></td>
<td>This field indicates the location of the error detection code, for disk devices.</td>
</tr>
<tr>
<td><code>dev.mem.flags</code></td>
<td>This field provides flags, for memory devices. Valid flags are:</td>
</tr>
<tr>
<td><code>TPLFMTFLAGS_ADDR</code></td>
<td>address is valid</td>
</tr>
<tr>
<td><code>TPLFMTFLAGS_AUTO</code></td>
<td>automatically map memory region</td>
</tr>
<tr>
<td><code>dev.mem.reserved</code></td>
<td>This field is reserved.</td>
</tr>
<tr>
<td><code>dev.mem.address</code></td>
<td>This field indicates the physical address, for memory devices.</td>
</tr>
<tr>
<td><code>dev.mem.edcloc</code></td>
<td>This field indicates the location of the error detection code, for memory devices.</td>
</tr>
</tbody>
</table>

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

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

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

csx_Parse_CISTPL_FUNCE – parse Function Extension tuple

SYNOPSIS

#include <sys/pccard.h>

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

INTERFACE

LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Client handle returned from csx_RegisterClient(9F).</td>
</tr>
<tr>
<td>tu</td>
<td>Pointer to a tuple_t structure (see tuple(9S)) returned by a call to csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).</td>
</tr>
<tr>
<td>cf</td>
<td>Pointer to a cistpl_funce_t structure which contains the parsed CISTPL_FUNCE tuple information upon return from this function.</td>
</tr>
<tr>
<td>fid</td>
<td>The function ID code to which this CISTPL_FUNCE tuple refers. See csx_Parse_CISTPL_FUNCID(9F).</td>
</tr>
</tbody>
</table>

DESCRIPTION

This function parses the Function Extension tuple, CISTPL_FUNCE, into a form usable by PC Card drivers.

The CISTPL_FUNCE tuple is used to describe information about a specific PC Card function. The information provided is determined by the Function Identification tuple, CISTPL_FUNCID, that is being extended. Each function has a defined set of extension tuples.

STRUCTURE

MEMBERS

The structure members of cistpl_funce_t are:

```c
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 */
    } data_modem;
};
```

9F-142 SunOS 5.6 modified 20 Dec 1996
```c
uint32_t cm; /* command protocols */
uint32_t ex; /* escape mechanisms */
uint32_t dy; /* standardized data encryption */
uint32_t ef; /* miscellaneous end user features */
uint32_t ncd; /* number of country codes */
uchar_t cd[16]; /* CCITT country code */
}
data_modem;
struct fax {
    uint32_t uf; /* highest data rate in DTE/UART */
    uint32_t fm; /* CCITT modulation standards */
    uint32_t fy; /* standardized data encryption */
    uint32_t fs; /* feature selection */
    uint32_t ncf; /* number of country codes */
    uchar_t cf[16]; /* CCITT country codes */
}
fax;
struct voice {
    uint32_t uv; /* highest data rate */
    uint32_t nsr; uint32_t sr[16]; /* voice sampling rates (100) */
    uint32_t nss; uint32_t ss[16]; /* voice sample sizes (10) */
    uint32_t nsc; uint32_t sc[16]; /* voice compression methods */
}
voice;
struct lan {
    uint32_t tech; /* network technology */
    uint32_t speed; /* media bit or baud rate */
    uint32_t media; /* network media supported */
    uint32_t con; /* open/closed connector standard */
    uint32_t id_sz; /* length of lan station id */
    uchar_t id[16]; /* station ID */
}
lan;
}
data;

The fields are defined as follows:

**function**

This field identifies the type of extended information provided about a function by the CISTPL_FUNCE tuple. This field is defined as follows:

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_SUB_SERIAL</td>
<td>Serial port interface</td>
</tr>
<tr>
<td>TPLFE_SUB_MODEM_COMMON</td>
<td>Common modem interface</td>
</tr>
<tr>
<td>TPLFE_SUB_MODEM_DATA</td>
<td>Data modem services</td>
</tr>
<tr>
<td>TPLFE_SUB_MODEM_FAX</td>
<td>Fax modem services</td>
</tr>
<tr>
<td>TPLFE_SUB_VOICE</td>
<td>Voice services</td>
</tr>
</tbody>
</table>

modified 20 Dec 1996
SunOS 5.6
9F-143
### 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.

<table>
<thead>
<tr>
<th>subfunction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_CAP_MODEM_DATA</td>
<td>Capabilities of the data modem interface</td>
</tr>
<tr>
<td>TPLFE_CAP_MODEM_FAX</td>
<td>Capabilities of the fax modem interface</td>
</tr>
<tr>
<td>TPLFE_CAP_MODEM_VOICE</td>
<td>Capabilities of the voice modem interface</td>
</tr>
<tr>
<td>TPLFE_CAP_SERIAL_DATA</td>
<td>Serial port interface for data modem services</td>
</tr>
<tr>
<td>TPLFE_CAP_SERIAL_FAX</td>
<td>Serial port interface for fax modem services</td>
</tr>
<tr>
<td>TPLFE_CAP_SERIAL_VOICE</td>
<td>Serial port interface for voice modem services</td>
</tr>
</tbody>
</table>

### ua

This is the serial port UART identification and is defined as follows:

<table>
<thead>
<tr>
<th>UART Identification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_UA_8250</td>
<td>Intel 8250</td>
</tr>
<tr>
<td>TPLFE_UA_16450</td>
<td>NS 16450</td>
</tr>
<tr>
<td>TPLFE_UA_16550</td>
<td>NS 16550</td>
</tr>
</tbody>
</table>

### uc

This identifies the serial port UART capabilities and is defined as follows:

<table>
<thead>
<tr>
<th>UART Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_UC_PARITY_SPACE</td>
<td>Space parity supported</td>
</tr>
<tr>
<td>TPLFE_UC_PARITY_MARK</td>
<td>Mark parity supported</td>
</tr>
<tr>
<td>TPLFE_UC_PARITY_ODD</td>
<td>Odd parity supported</td>
</tr>
<tr>
<td>TPLFE_UC_PARITY_EVEN</td>
<td>Even parity supported</td>
</tr>
<tr>
<td>TPLFE_UC_CS5</td>
<td>5 bit characters supported</td>
</tr>
<tr>
<td>TPLFE_UC_CS6</td>
<td>6 bit characters supported</td>
</tr>
<tr>
<td>TPLFE_UC_CS7</td>
<td>7 bit characters supported</td>
</tr>
<tr>
<td>TPLFE_UC_CS8</td>
<td>8 bit characters supported</td>
</tr>
<tr>
<td>TPLFE_UC_STOP_1</td>
<td>1 stop bit supported</td>
</tr>
<tr>
<td>TPLFE_UC_STOP_15</td>
<td>1.5 stop bits supported</td>
</tr>
<tr>
<td>TPLFE_UC_STOP_2</td>
<td>2 stop bits supported</td>
</tr>
</tbody>
</table>

### fc

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

<table>
<thead>
<tr>
<th>Flow Control Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_FC_TX_XONOFF</td>
<td>Transmit XON/XOFF</td>
</tr>
<tr>
<td>TPLFE_FC_RX_XONOFF</td>
<td>Receiver XON/XOFF</td>
</tr>
<tr>
<td>TPLFE_FC_TX_HW</td>
<td>Transmit hardware flow control (CTS)</td>
</tr>
<tr>
<td>TPLFE_FC_RX_HW</td>
<td>Receiver hardware flow control (RTS)</td>
</tr>
<tr>
<td>TPLFE_FC_TRANS</td>
<td>Transparent flow control</td>
</tr>
</tbody>
</table>
This identifies the modem modulation standards and is defined as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_MS_BELL103</td>
<td>300bps</td>
</tr>
<tr>
<td>TPLFE_MS_V21</td>
<td>300bps (V.21)</td>
</tr>
<tr>
<td>TPLFE_MS_V23</td>
<td>600/1200bps (V.23)</td>
</tr>
<tr>
<td>TPLFE_MS_V22AB</td>
<td>1200bps (V.22A V.22B)</td>
</tr>
<tr>
<td>TPLFE_MS_BELL212</td>
<td>2400bps (US Bell 212)</td>
</tr>
<tr>
<td>TPLFE_MS_V22BIS</td>
<td>2400bps (V.22bis)</td>
</tr>
<tr>
<td>TPLFE_MS_V26</td>
<td>2400bps leased line (V.26)</td>
</tr>
<tr>
<td>TPLFE_MS_V26BIS</td>
<td>2400bps (V.26bis)</td>
</tr>
<tr>
<td>TPLFE_MS_V27BIS</td>
<td>4800/2400bps leased line (V.27bis)</td>
</tr>
<tr>
<td>TPLFE_MS_V29</td>
<td>9600/7200/4800 leased line (V.29)</td>
</tr>
<tr>
<td>TPLFE_MS_V32</td>
<td>Up to 9600bps (V.32)</td>
</tr>
<tr>
<td>TPLFE_MS_V32BIS</td>
<td>Up to 14400bps (V.32bis)</td>
</tr>
<tr>
<td>TPLFE_MS_VFAST</td>
<td>Up to 28800 V.FAST</td>
</tr>
</tbody>
</table>

This identifies modem error correction/detection protocols and is defined as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_EM_MNP</td>
<td>MNP levels 2-4</td>
</tr>
<tr>
<td>TPLFE_EM_V42</td>
<td>CCITT LAPM (V.42)</td>
</tr>
</tbody>
</table>

This identifies modem data compression protocols and is defined as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_DC_V42BI</td>
<td>CCITT compression V.42</td>
</tr>
<tr>
<td>TPLFE_DC_MNP5</td>
<td>MNP compression (uses MNP 2, 3 or 4)</td>
</tr>
</tbody>
</table>

This identifies modem command protocols and is defined as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_CM_AT1</td>
<td>ANSI/EIA/TIA 602 &quot;Action&quot; commands</td>
</tr>
<tr>
<td>TPLFE_CM_AT2</td>
<td>ANSI/EIA/TIA 602 &quot;ACE/DCE IF Params&quot;</td>
</tr>
<tr>
<td>TPLFE_CM_AT3</td>
<td>ANSI/EIA/TIA 602 &quot;Ace Parameters&quot;</td>
</tr>
<tr>
<td>TPLFE_CM_MNP_AT</td>
<td>MNP specification AT commands</td>
</tr>
<tr>
<td>TPLFE_CM_V25BIS</td>
<td>V.25bis calling commands</td>
</tr>
<tr>
<td>TPLFE_CM_V25A</td>
<td>V.25bis test procedures</td>
</tr>
<tr>
<td>TPLFE_CM_DMCL</td>
<td>DMCL command mode</td>
</tr>
</tbody>
</table>

This identifies the modem escape mechanism and is defined as follows:

modified 20 Dec 1996
<table>
<thead>
<tr>
<th><strong>TPLFE_EX_BREAK</strong></th>
<th>BREAK support standardized</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TPLFE_EX_PLUS</strong></td>
<td>+++ returns to command mode</td>
</tr>
<tr>
<td><strong>TPLFE_EX_UD</strong></td>
<td>User defined escape character</td>
</tr>
</tbody>
</table>

**dy** This identifies modem standardized data encryption and is a reserved field for future use and must be set to 0.

**ef** This identifies modem miscellaneous features and is defined as follows:

| **TPLFE_EF_CALLERID** | Caller ID is supported |

**fm** This identifies fax modulation standards and is defined as follows:

| **TPLFE_FM_V21C2**            | 300bps (V.21-C2) |
| **TPLFE_FM_V27TER**           | 4800/2400bps (V.27ter) |
| **TPLFE_FM_V29**              | 9600/7200/4800 leased line (V.29) |
| **TPLFE_FM_V17**              | 14.4K/12K/9600/7200bps (V.17) |
| **TPLFE_FM_V33**              | 14.4K/12K/9600/7200 leased line (V.33) |

**fs** This identifies the fax feature selection and is defined as follows:

| **TPLFE_FS_T3** | Group 2 (T.3) service class |
| **TPLFE_FS_T4** | Group 3 (T.4) service class |
| **TPLFE_FS_T6** | Group 4 (T.6) service class |
| **TPLFE_FS_ECM** | Error Correction Mode |
| **TPLFE_FS_VOICEREQ** | Voice requests allowed |
| **TPLFE_FS_POLLING** | Polling support |
| **TPLFE_FS_FTP** | File transfer support |
| **TPLFE_FS_PASSWORD** | Password support |

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

| **TPLFE_LAN_TECH_ARCNET** | Arcnet |
| **TPLFE_LAN_TECHETHERNET** | Ethernet |
| **TPLFE_LAN_TECH_TOKENRING** | Token Ring |
| **TPLFE_LAN_TECH_LOCALTALK** | Local Talk |
media  This identifies the LAN media type and is defined as follows:

<table>
<thead>
<tr>
<th>Media Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFE_LAN_MEDIA_INHERENT</td>
<td>Generic interface</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_UTP</td>
<td>Unshielded twisted pair</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_STP</td>
<td>Shielded twisted pair</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_THIN_COAX</td>
<td>Thin coax</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_THICK_COAX</td>
<td>Thick coax</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_FIBER</td>
<td>Fiber</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_SSR_902</td>
<td>Spread spectrum radio 902-928 MHz</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_SSR_2_4</td>
<td>Spread spectrum radio 2.4 GHz</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_SSR_5_4</td>
<td>Spread spectrum radio 5.4 GHz</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_DIFFUSE_IR</td>
<td>Diffuse infra red</td>
</tr>
<tr>
<td>TPLFE_LAN_MEDIA_PTP_IR</td>
<td>Point to point infra red</td>
</tr>
</tbody>
</table>

RETURN VALUES

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

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

- `csx_GetFirstTuple(9F)`
- `csx_GetTupleData(9F)`
- `csx_Parse_CISTPL_FUNCID(9F)`
- `csx_RegisterClient(9F)`
- `csxValidateCIS(9F)`
- `tuple(9S)`

PC Card 95 Standard, PCMCIA/JEIDA

modified 20 Dec 1996 SunOS 5.6 9F-147
NAME  csx_Parse_CISTPL_FUNCID – parse Function Identification tuple

SYNOPSIS  
```
#include <sys/pccard.h>
int32_t csx_Parse_CISTPL_FUNCID(client_handle_t ch, tuple_t *tu,
cistpl_funcid_t *cf);
```

INTERFACE  
LEVEL  Solaris DDI Specific (Solaris DDI)

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

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

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

DESCRIPTION  
This function parses the Function Identification tuple, CISTPL_FUNCID, into a form
usable by PC Card drivers.

The CISTPL_FUNCID tuple is used to describe information about the functionality pro-
vided by a PC Card. Information is also provided to enable system utilities to decide if
the PC Card should be configured during system initialization. If additional function
specific information is available, one or more function extension tuples of type
CISTPL_FUNCE follow this tuple (see csx_Parse_CISTPL_FUNCE(9F)).

STRUCTURE  
MEMBERS  The structure members of cistpl_funcid_t are:

```
uint32_t function;  /* PC Card function code */
uint32_t sysinit;   /* system initialization mask */
```

The fields are defined as follows:

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFUNC_MULTI</td>
<td>Vendor-specific multifunction card</td>
</tr>
<tr>
<td>TPLFUNC_MEMORY</td>
<td>Memory card</td>
</tr>
<tr>
<td>TPLFUNC_SERIAL</td>
<td>Serial I/O port</td>
</tr>
<tr>
<td>TPLFUNC_PARALLEL</td>
<td>Parallel printer port</td>
</tr>
<tr>
<td>TPLFUNC_FIXED</td>
<td>Fixed disk, silicon or removable</td>
</tr>
<tr>
<td>TPLFUNC_VIDEO</td>
<td>Video interface</td>
</tr>
<tr>
<td>TPLFUNC_LAN</td>
<td>Local Area Network adapter</td>
</tr>
<tr>
<td>TPLFUNC_AIMS</td>
<td>Auto Incrementing Mass Storage</td>
</tr>
<tr>
<td>TPLFUNC_SCSI</td>
<td>SCSI bridge</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

<table>
<thead>
<tr>
<th>TPLFUNC_SECURITY</th>
<th>Security cards</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLFUNC_VENDOR_SPECIFIC</td>
<td>Vendor specific</td>
</tr>
<tr>
<td>TPLFUNC_UNKNOWN</td>
<td>Unknown function(s)</td>
</tr>
</tbody>
</table>

**sysinit**

This field is bit-mapped and defined as follows:

<table>
<thead>
<tr>
<th>TPLINIT_POST</th>
<th>POST should attempt configure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLINIT_ROM</td>
<td>Map ROM during sys init</td>
</tr>
</tbody>
</table>

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

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

PC Card 95 Standard, PCMCIA/JEIDA

modified 20 Dec 1996

SunOS 5.6

9F-149
**NAME**

csx_Parse_CISTPL_GEOMETRY – parse the Geometry tuple

**SYNOPSIS**

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

int32_t csx_Parse_CISTPL_GEOMETRY(client_handle_t ch, tuple_t *tu, 
cistpl_geometry_t *pt);
```

**INTERFACE LEVEL**

Solaris DDI Specific (Solaris DDI)

**ARGUMENTS**

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

**DESCRIPTION**

This function parses the Geometry tuple, CISTPL_GEOMETRY, into a form usable by PC Card drivers.

The CISTPL_GEOMETRY tuple indicates the geometry of a disk-like device.

**STRUCTURE MEMBERS**

The structure members of `cistpl_geometry_t` are:

- `uint32_t spt;`
- `uint32_t tpc;`
- `uint32_t ncyl;`

The fields are defined as follows:

- **spt**
  - This field indicates the number of sectors per track.
- **tpc**
  - This field indicates the number of tracks per cylinder.
- **ncyl**
  - This field indicates the number of cylinders.

**RETURN VALUES**

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

**CONTEXT**

This function may be called from user or kernel context.

**SEE ALSO**

- `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_RegisterClient(9F)`, `csx_ValidateCIS(9F)`, `tuple(9S)`
- PC Card 95 Standard, PCMCIA/JEIDA

---

9F-150 SunOS 5.6 modified 24 Jan 1997
NAME

csx_Parse_CISTPL_JEDEC_C, csx_Parse_CISTPL_JEDEC_A – parse JEDEC Identifier tuples

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_JEDEC_C(client_handle_t ch, tuple_t *tu, cistpl_jedec_t *cj);

int32_t csx_Parse_CISTPL_JEDEC_A(client_handle_t ch, tuple_t *tu, cistpl_jedec_t *cj);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch Client handle returned from csx_RegisterClient(9F).

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

cj Pointer to a cistpl_jedec_t structure which contains the parsed CISTPL_JEDEC_C or CISTPL_JEDEC_A tuple information upon return from these functions, respectively.

DESCRIPTION

csx_Parse_CISTPL_JEDEC_C() and csx_Parse_CISTPL_JEDEC_A() parse the JEDEC Identifier tuples, CISTPL_JEDEC_C and CISTPL_JEDEC_A, respectively, into a form usable by PC Card drivers.

The CISTPL_JEDEC_C and CISTPL_JEDEC_A tuples are optional tuples provided for cards containing programmable devices. They describe information for Common Memory or Attribute Memory space, respectively.

STRUCTURE MEMBERS

The structure members of cistpl_jedec_t are:

    uint32_t nid; /* # of JEDEC identifiers present */
    jedec_ident_t jid[CISTPL_JEDEC_MAX_IDENTIFIERS];

The structure members of jedec_ident_t are:

    uint32_t id;  /* manufacturer id */
    uint32_t info; /* manufacturer specific info */

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD No PC Card in socket.

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

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

These functions may be called from user or kernel context.

modified 20 Dec 1996 SunOS 5.6 9F-151
SEE ALSO

csx_GetFirstTuplet(9F), csx_GetTupleData(9F), csx_Parse_CISTPL_DEVICE(9F),
csx_RegisterClient(9F), csx.ValidateCIS(9F), tuple(9S)

PC Card 95 Standard, PCMCIA/JEIDA

9F-152

SunOS 5.6

modified 20 Dec 1996
NAME  csx_Parse_CISTPL_LINKTARGET – parse the Link Target tuple

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

INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

ARGUMENTS
ch     Client handle returned from csx_RegisterClient(9F).
tu     Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
        csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
pt     Pointer to a cistpl_linktarget_t structure which contains the parsed
        CISTPL_LINKTARGET tuple information upon return from this function.

DESCRIPTION
This function parses the Link Target tuple, CISTPL_LINKTARGET, into a form usable by
PC Card drivers.

The CISTPL_LINKTARGET tuple is used to verify that tuple chains other than the primary
chain are valid. All secondary tuple chains are required to contain this tuple as the first
tuple of the chain.

STRUCTURE MEMBERS
The structure members of cistpl_linktarget_t are:

    uint32_t length;
    char tpltg_tag[CIS_MAX_TUPLE_DATA_LEN];

The fields are defined as follows:

length     This field indicates the number of bytes in tpltg_tag.
tpltg_tag  This field provides the Link Target tuple information.

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

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

SEE ALSO
    csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F),
    csx_ValidateCIS(9F), tuple(9S)
    PC Card 95 Standard, PCMCIA/JEIDA

modified 24 Jan 1997  SunOS 5.6  9F-153
NAME  csx_Parse_CISTPL_LONGLINK_A, csx_Parse_CISTPL_LONGLINK_C – parse the Long Link A and C tuples

SYNOPSIS  #include <sys/pccard.h>
            
            int32_t csx_Parse_CISTPL_LONGLINK_A(client_handle_t ch, tuple_t *tu,  
                          cistpl_longlink_ac_t *pt);
            int32_t csx_Parse_CISTPL_LONGLINK_C(client_handle_t ch, tuple_t *tu,  
                          cistpl_longlink_ac_t *pt);

INTERFACE LEVEL Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

ch  Client handle returned from csx_RegisterClient(9F).

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

pt  Pointer to a cistpl_longlink_ac_t structure which contains the parsed CISTPL_LONGLINK_A or CISTPL_LONGLINK_C tuple information upon return from this function.

DESCRIPTION  

This function parses the Long Link A and C tuples, CISTPL_LONGLINK_A and CISTPL_LONGLINK_C, into a form usable by PC Card drivers. The CISTPL_LONGLINK_A and CISTPL_LONGLINK_C tuples provide links to Attribute and Common Memory.

STRUCTURE MEMBERS  

The structure members of cistpl_longlink_ac_t are:

    uint32_t  flags;
    uint32_t  tpll_addr;

The fields are defined as follows:

flags  This field indicates the type of memory:

CISTPL_LONGLINK_AC_AM long link to Attribute Memory
CISTPL_LONGLINK_AC_CM long link to Common Memory

tpll_addr  This field provides the offset from the beginning of the specified address space.

RETURN VALUES  

CS_SUCCESS  Successful operation.
CS_BAD_HANDLE  Client handle is invalid.
CS_UNKNOWN_TUPLE  Parser does not know how to parse tuple.
CS_NO_CARD  No PC Card in socket.
CS_NO_CIS  No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.
<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>This function may be called from user or kernel context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEE ALSO</td>
<td>csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx_ValidateCIS(9F), tuple(9S)</td>
</tr>
</tbody>
</table>

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

csx_Parse_CISTPL_LONGLINK_MFC – parse the Multi-Function tuple

### SYNOPSIS

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

int32_t csx_Parse_CISTPL_LONGLINK_MFC(client_handle_t ch, tuple_t *tu,
                                      cistpl_longlink_mfc_t *pt);
```

### INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

### ARGUMENTS

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

### DESCRIPTION

This function parses the Multi-Function tuple, CISTPL_LONGLINK_MFC, into a form usable by PC Card drivers.

The CISTPL_LONGLINK_MFC tuple describes the start of the function-specific CIS for each function on a multi-function card.

### STRUCTURE MEMBERS

The structure members of `cistpl_longlink_mfc_t` are:

- `uint32_t nfuncs;`
- `uint32_t nregs;`
- `uint32_t function[CIS_MAX_FUNCTIONS].tas`
- `uint32_t function[CIS_MAX_FUNCTIONS].addr`

The fields are defined as follows:

- `nfuncs`: This field indicates the number of functions on the PC card.
- `nregs`: This field indicates the number of configuration register sets.
- `function[CIS_MAX_FUNCTIONS].tas`: This field provides the target address space for each function on the PC card. This field can be one of:
  - `CISTPL_LONGLINK_MFC_TAS_AM`: CIS in attribute memory
  - `CISTPL_LONGLINK_MFC_TAS_CM`: CIS in common memory
- `function[CIS_MAX_FUNCTIONS].addr`: This field provides the target address offset for each function on the PC card.

### RETURN VALUES

- `CS_SUCCESS`: Successful operation.
- `CS_BAD_HANDLE`: Client handle is invalid.
- `CS_UNKNOWN_TUPLE`: Parser does not know how to parse tuple.
- `CS_NO_CARD`: No PC Card in socket.
- `CS_NO_CIS`: No Card Information Structure (CIS) on PC Card.
- `CS_UNSUPPORTED_FUNCTION`: No PCMCIA hardware installed.
| CONTEXT | This function may be called from user or kernel context. |
| SEE ALSO | csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_RegisterClient(9F), csx.ValidateCIS(9F), tuple(9S) |

*PC Card 95 Standard, PCMCIA/JEIDA*
NAME
  csx_Parse_CISTPL_MANFID – parse Manufacturer Identification tuple

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

INTERFACE LEVEL
  Solaris DDI Specific (Solaris DDI)

ARGUMENTS
  ch  Client handle returned from csx_RegisterClient(9F).
  tu  Pointer to a tuple_t structure (see tuple(9S)) returned by a call to
       csx_GetFirstTuple(9F) or csx_GetNextTuple(9F).
  cm  Pointer to a cistpl_manfid_t structure which contains the parsed
       CISTPL_MANFID tuple information upon return from this function.

DESCRIPTION
  This function parses the Manufacturer Identification tuple, CISTPL_MANFID, into a form
  usable by PC Card drivers.

  The CISTPL_MANFID tuple is used to describe the information about the manufacturer of
  a PC Card. There are two types of information, the PC Card’s manufacturer and a
  manufacturer card number.

STRUCTURE MEMBERS
  The structure members of cistpl_manfid_t are:

    uint32_t manf;    /* PCMCIA assigned manufacturer code */
    uint32_t card;    /* manufacturer information
                      (part number and/or revision) */

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

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

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

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

PC Card 95 Standard, PCMCIA/JEIDA

modified 24 Jan 1997 SunOS 5.6 9F-159
csx_Parse_CISTPL_SPCL (9F) - parse the Special Purpose tuple

SYNOPSIS

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

int32_t csx_Parse_CISTPL_SPCL(client_handle_t ch, tuple_t *tu,
                                 cistpl_spcl_t *csp);
```

INTERFACE

LEVEL Solaris DDI Specific (Solaris DDI)

ARGUMENTS

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

DESCRIPTION

This function parses the Special Purpose tuple, CISTPL_SPCL, into a form usable by PC Card drivers.

The CISTPL_SPCL tuple is identified by an identification field that is assigned by PCMCIA or JEIDA. A sequence field allows a series of CISTPL_SPCL tuples to be used when the data exceeds the size that can be stored in a single tuple; the maximum data area of a series of CISTPL_SPCL tuples is unlimited. Another field gives the number of bytes in the data field in this tuple.

STRUCTURE

MEMBERS

The structure members of `cistpl_date_t` are:

- `id` : This field contains a PCMCIA or JEIDA assigned value that identifies this series of one or more CISTPL_SPCL tuples. These field values are assigned by contacting either PCMCIA or JEIDA.
- `seq` : This field contains a data sequence number.
- `bytes` : This field contains the number of data bytes in the `data[CIS_MAX_TUPLE_DATA_LEN]`.
- `data` : The data component of this tuple.

The fields are defined as follows:

- `id` : This field contains a PCMCIA or JEIDA assigned value that identifies this series of one or more CISTPL_SPCL tuples. These field values are assigned by contacting either PCMCIA or JEIDA.
- `seq` : This field contains a data sequence number.
- `bytes` : This field contains the number of data bytes in the `data[CIS_MAX_TUPLE_DATA_LEN]`.
- `data` : The data component of this tuple.
Kernel Functions for Drivers

**csx_Parse_CISTPL_SPCL (9F)**

<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_UNKNOWN_TUPLE</td>
<td>Parser does not know how to parse tuple.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_NO_CIS</td>
<td>No Card Information Structure (CIS) on PC Card.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

**CONTEXT**

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

modified 20 Dec 1996

SunOS 5.6

9F-161
csx_Parse_CISTPL_SWIL (9F)

NAME
csx_Parse_CISTPL_SWIL – parse the Software Interleaving tuple

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

INTERFACE
LEVEL
Solaris DDI Specific (Solaris DDI)

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

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

pt Pointer to a cistpl_swil_t structure which contains the parsed CISTPL_SWIL
tuple information upon return from this function.

DESCRIPTION
This function parses the Software Interleaving tuple, CISTPL_SWIL, into a form usable by
PC Card drivers.
The CISTPL_SWIL tuple provides the software interleaving of data within a partition on
the card.

STRUCTURE
MEMBERS
The structure members of cistpl_swil_t are:

uint32_t intrlv;

The fields are defined as follows:
intrlv This field provides the software interleaving for a partition.

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

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

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

PC Card 95 Standard, PCMCIA/JEIDA

9F-162 SunOS 5.6 modified 24 Jan 1997
Kernel Functions for Drivers

NAME

csx_Parse_CISTPL_VERS_1 – parse Level-1 Version/Product Information tuple

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_Parse_CISTPL_VERS_1(client_handle_t ch, tuple_t *tu, cistpl_vers_1_t *cv1);

INTERFACE LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch Client handle returned from csx_RegisterClient(9F).

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

cv1 Pointer to a cistpl_vers_1_t structure which contains the parsed CISTPL_VERS_1 tuple information upon return from this function.

DESCRIPTION

This function parses the Level-1 Version/Product Information tuple, CISTPL_VERS_1, into a form usable by PC Card drivers.

The CISTPL_VERS_1 tuple is used to describe the card Level-1 version compliance and card manufacturer information.

STRUCTURE MEMBERS

The structure members of cistpl_vers_1_t are:

uint32_t major; /* major version number */

uint32_t minor; /* minor version number */

uint32_t ns; /* number of information strings */

char pi[CISTPL_VERS_1_MAX_PROD_STRINGS] /* pointers to product [CIS_MAX_TUPLE_DATA_LEN]; information strings */

RETURN VALUES

CS_SUCCESS Successful operation.

CS_BAD_HANDLE Client handle is invalid.

CS_UNKNOWN_TUPLE Parser does not know how to parse tuple.

CS_NO_CARD No PC Card in socket.

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

CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

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

PC Card 95 Standard, PCMCIA/JEIDA

modified 20 Dec 1996

SunOS 5.6

9F-163
NAME  csx_Parse_CISTPL_VERS_2 – parse Level-2 Version and Information tuple

SYNOPSIS  
#include <sys/pccard.h>

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

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

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

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

cv2  Pointer to a cistpl_vers_2_t structure which contains the parsed CISTPL_VERS_2
tuple information upon return from this function.

DESCRIPTION  This function parses the Level-2 Version and Information tuple, CISTPL_VERS_2, into a
form usable by PC Card drivers.

The CISTPL_VERS_2 tuple is used to describe the card Level-2 information which has the
logical organization of the card’s data.

STRUCTURE MEMBERS  The structure members of cistpl_vers_2_t are:

uint32_t vers;  /* version number */
uint32_t complly;  /* level of compliance */
uint32_t dindex;  /* byte address of first data byte in card */
uint32_t vspec8;  /* vendor specific (byte 8) */
uint32_t vspec9;  /* vendor specific (byte 9) */
uint32_t nhdr;  /* number of copies of CIS present on device */
char oem[CIS_MAX_TUPLE_DATA_LEN];  /* Vendor of software that
formatted card */
char info[CIS_MAX_TUPLE_DATA_LEN];  /* Informational message
about card */

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

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

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

PC Card 95 Standard, PCMCIA/JEIDA
NAME  csx_ParseTuple – generic tuple parser

SYNOPSIS  

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

int32 csx_ParseTuple(client_handle_t ch, tuple_t *tu, cisparse_t *cp, cisdata_t cd);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

- `ch`  Client handle returned from `csx_RegisterClient(9F)`.
- `tu`  Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- `cp`  Pointer to a `cisparse_t` structure that unifies all tuple parsing structures.
- `cd`  Extended tuple data for some tuples.

DESCRIPTION  This function is the generic tuple parser entry point.

STRUCTURE MEMBERS  The structure members of `cisparse_t` are:

```
typedef union cisparse_t {
    cistpl_config_t cistpl_config;
    cistpl_device_t cistpl_device;
    cistpl_vers_1_t cistpl_vers_1;
    cistpl_vers_2_t cistpl_vers_2;
    cistpl_jedec_t cistpl_jedec;
    cistpl_format_t cistpl_format;
    cistpl_geometry_t cistpl_geometry;
    cistpl_byteorder_t cistpl_byteorder;
    cistpl_date_t cistpl_date;
    cistpl_battery_t cistpl_battery;
    cistpl_org_t cistpl_org;
    cistpl_manfid_t cistpl_manfid;
    cistpl_funcid_t cistpl_funcid;
    cistpl_funce_t cistpl_funce;
    cistpl_cftable_entry_t cistpl_cftable_entry;
    cistpl_linktarget_t cistpl_linktarget;
    cistpl_longlink_ac_t cistpl_longlink_ac;
    cistpl_longlink_mfc_t cistpl_longlink_mfc;
    cistpl_spcl_t cistpl_spcl;
    cistpl_swil_t cistpl_swil;
    cistpl_bar_t cistpl_bar;
    cistpl_devicegeo_t cistpl_devicegeo;
    cistpl_longlink_cb_t cistpl_longlink_cb;
    cistpl_get_tuple_name_t cistpl_get_tuple_name;
} cisparse_t;
```
Kernel Functions for Drivers

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

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

SEE ALSO
- `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_Parse_CISTPL_BATTERY(9F)`, `csx_Parse_CISTPL_BYTEORDER(9F)`, `csx_Parse_CISTPL_CFTABLE_ENTRY(9F)`, `csx_Parse_CISTPL_CONFIG(9F)`, `csx_Parse_CISTPL_DATE(9F)`, `csx_Parse_CISTPL_DEVICE(9F)`, `csx_Parse_CISTPL_FUNCE(9F)`, `csx_Parse_CISTPL_FUNCID(9F)`, `csx_Parse_CISTPL_JEDEC_C(9F)`, `csx_Parse_CISTPL_MANFID(9F)`, `csx_Parse_CISTPL_SPCL(9F)`, `csx_Parse_CISTPL_VERS_1(9F)`, `csx_Parse_CISTPL_VERS_2(9F)`, `csx_RegisterClient(9F)`, `csx_ValidateCIS(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*

modified 20 Dec 1996
SunOS 5.6
9F-167
NAME

csx_Put8, csx_Put16, csx_Put32, csx_Put64 – write to device register

SYNOPSIS

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

void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);
```

INTERFACE

LEVEL Solaris DDI Specific (Solaris DDI)

ARGUMENTS

- `handle` The access handle returned from `csx_RequestIO(9F)`,
  `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- `offset` The offset in bytes from the base of the mapped resource.
- `value` The data to be written to the device.

DESCRIPTION

These functions generate a write of various sizes to the mapped memory or device regis-
ter.

The `csx_Put8()`, `csx_Put16()`, `csx_Put32()`, and `csx_Put64()` functions write 8 bits, 16 bits,
32 bits, and 64 bits of data, respectively, to the device address represented by the handle,
`handle`, at an offset in bytes represented by the offset, `offset`.

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

CONTEXT

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

SEE ALSO

`csx_DupHandle(9F)`, `csx_Get8(9F)`, `csx_GetMappedAddr(9F)`, `csx_RepGet8(9F)`,
`csx_RepPut8(9F)`, `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`

PC Card 95 Standard, PCMCIA/JEIDA

9F-168 SunOS 5.6 modified 19 Jul 1996
NAME  csx_RegisterClient – register a client

SYNOPSIS  #include <sys/pccard.h>

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

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

ch  Pointer to a client_handle_t structure.

mc  Pointer to a client_reg_t structure.

DESCRIPTION  This function registers a client with Card Services and returns a unique client handle for the client. The client handle must be passed to csx_DeregisterClient(9F) when the client terminates.

STRUCTURE MEMBERS  The structure members of client_reg_t are:

uint32_t Attributes;  
uint32_t EventMask;  
uint32_t event_callback_args_t event_callback_args;  
uint32_t Version;  
/* CS version to expect */
csfuntion_t *event_handler;  
*iblk_cookie;  
/* event iblk cookie */
ddi_idevice_cookie_t *idev_cookie;  
/* event idev cookie */
dev_info_t *dip;  
/* client’s dip */
driver_name[MODMAXNAMELEN];

The fields are defined as follows:

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFO_MEM_CLIENT</td>
<td>Memory client device driver</td>
</tr>
<tr>
<td>INFO_MTD_CLIENT</td>
<td>Memory Technology Driver client</td>
</tr>
<tr>
<td>INFO_IO_CLIENT</td>
<td>IO client device driver</td>
</tr>
<tr>
<td>INFO_CARD_SHARE</td>
<td>Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events</td>
</tr>
<tr>
<td>INFO_CARD_EXCL</td>
<td>Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events</td>
</tr>
</tbody>
</table>

modified 19 Jul 1996  SunOS 5.6  9F-169
INFO_MEM_CLIENT
INFO_MTD_CLIENT
INFO_IO_CLIENT

These bits are mutually exclusive (that is, only one bit may be
set), but one of the bits must be set.

INFO_CARD_SHARE
INFO_CARD_EXCL

If either of these bits is set, the client will receive a
CS_EVENT_REGISTRATION_COMPLETE event when Card Ser-
vice 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

driver_name

The client must set this field with a pointer to the client’s dip.
The client must copy a driver-unique name into this member. This
name must be identical across all instances of the driver.
RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_ATTRIBUTE</td>
<td>No client type or more than one client type specified.</td>
</tr>
<tr>
<td>CS_OUT_OF_RESOURCE</td>
<td>Card Services is unable to register client.</td>
</tr>
<tr>
<td>CS_BAD_VERSION</td>
<td>Card Services version is incompatible with client.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client has already registered for this socket.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

csx_DeregisterClient(9F), csx_RequestSocketMask(9F)

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996

SunOS 5.6
### NAME
csx_ReleaseConfiguration – release PC Card and socket configuration

### SYNOPSIS
```c
#include <sys/pccard.h>
int32_t csx_ReleaseConfiguration(client_handle_t ch, release_config_t *rc);
```

### INTERFACE LEVEL
Solaris DDI Specific (Solaris DDI)

### ARGUMENTS
- **ch**: Client handle returned from `csx_RegisterClient(9F)`.
- **rc**: Pointer to a `release_config_t` structure.

### DESCRIPTION
This function returns a PC Card and socket to a simple memory only interface and sets the card to configuration zero by writing a `0` to the PC card’s COR (Configuration Option Register).

Card Services may remove power from the socket if no clients have indicated their usage of the socket by an active `csx_RequestConfiguration(9F)` or `csx_RequestWindow(9F)`.

Card Services is prohibited from resetting the PC Card and is not required to cycle power through zero (0) volts.

After calling `csx_ReleaseConfiguration()`, any resources requested via the request functions `csx_RequestIO(9F)`, `csx_RequestIRQ(9F)`, or `csx_RequestWindow(9F)` that are no longer needed should be returned to Card Services via the corresponding `csx_ReleaseIO(9F)`, `csx_ReleaseIRQ(9F)`, or `csx_ReleaseWindow(9F)` functions.

`csx_ReleaseConfiguration()` must be called to release the current card and socket configuration before releasing any resources requested by the driver via the request functions named above.

### STRUCTURE MEMBERS
The structure members of `release_config_t` are:

```c
uint32_t Socket; /* socket number */
```

The `Socket` field is not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

### RETURN VALUES
- **CS_SUCCESS**: Successful operation.
- **CS_BAD_HANDLE**: Client handle is invalid or `csx_RequestConfiguration(9F)` not done.
- **CS_BAD_SOCKET**: Error getting or setting socket hardware parameters.
- **CS_NO_CARD**: No PC card in socket.
- **CS_UNSUPPORTED_FUNCTION**: No PCMCIA hardware installed.

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

---

9F-172 SunOS 5.6 modified 19 Jul 1996
SEE ALSO

csx_RegisterClient(9F), csx_RequestConfiguration(9F), csx_RequestIO(9F),
csx_RequestIRQ(9F), csx_RequestWindow(9F)

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996

SunOS 5.6

9F-173
csx_RepGet8 (9F)  Kernel Functions for Drivers

NAME  csx_RepGet8, csx_RepGet16, csx_RepGet32, csx_RepGet64 – read repetitively from the device register

SYNOPSIS  #include <sys/pccard.h>
void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  handle  The access handle returned from csx_RequestIO(9F), csx_RequestWindow(9F), or csx_DupHandle(9F).
hostaddr  Source host address.
offset  The offset in bytes from the base of the mapped resource.
repcount  Number of data accesses to perform.
flags  Device address flags.

DESCRIPTION  These functions generate multiple reads of various sizes from the mapped memory or device register.

The csx_RepGet8(), csx_RepGet16(), csx_RepGet32(), and csx_RepGet64() functions generate repcount reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, handle, at an offset in bytes represented by the offset, offset. The data read is stored consecutively into the buffer pointed to by the host address pointer, hostaddr.

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

When the flags argument is set to CS_DEV_AUTOINCR, these functions increment the device offset, offset, after each datum read operation. However, when the flags argument is set to CS_DEV_NO_AUTOINCR, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

9F-174  SunOS 5.6  modified 19 Jul 1996
These functions may be called from user, kernel, or interrupt context.

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

*PC Card 95 Standard, PCMCIA/JEIDA*
NAME  csx_RepPut8, csx_RepPut16, csx_RepPut32, csx_RepPut64 – write repetitively to the device register

SYNOPSIS  

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

void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);

void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset, uint32_t repcount, uint32_t flags);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

- **handle**  The access handle returned from `csx_RequestIO(9F)`, `csx_RequestWindow(9F)`, or `csx_DupHandle(9F)`.
- **hostaddr**  Source host address.
- **offset**  The offset in bytes from the base of the mapped resource.
- **repcount**  Number of data accesses to perform.
- **flags**  Device address flags.

DESCRIPTION  These functions generate multiple writes of various sizes to the mapped memory or device register.

The `csx_RepPut8()`, `csx_RepPut16()`, `csx_RepPut32()`, and `csx_RepPut64()` functions generate `repcount` writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, `handle`, at an offset in bytes represented by the offset, `offset`. The data written is read consecutively from the buffer pointed to by the host address pointer, `hostaddr`.

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

When the `flags` argument is set to `CS_DEV_AUTOINCR`, these functions increment the device offset, `offset`, after each datum write operation. However, when the `flags` argument is set to `CS_DEV_NO_AUTOINCR`, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.
These functions may be called from user, kernel, or interrupt context.

SEE ALSO  

- `csx_DupHandle(9F)`  
- `csx_Get8(9F)`  
- `csx_GetMappedAddr(9F)`  
- `csx_Put8(9F)`  
- `csx_RepGet8(9F)`  
- `csx_RequestIO(9F)`  
- `csx_RequestWindow(9F)`

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996

SunOS 5.6

9F-177
NAME

csx_RequestConfiguration – configure the PC Card and socket

SYNOPSIS

#include <sys/pccard.h>

int32_t csx_RequestConfiguration(client_handle_t ch, config_req_t *cr);

INTERFACE

LEVEL

Solaris DDI Specific (Solaris DDI)

ARGUMENTS

ch Client handle returned from csx_RegisterClient(9F).

cr Pointer to a config_req_t structure.

DESCRIPTION

This function configures the PC Card and socket. It must be used by clients that require
I/O or IRQ resources for their PC Card.

csx_RequestIO(9F) and csx_RequestIRQ(9F) must be used before calling this function to
specify the I/O and IRQ requirements for the PC Card and socket if necessary.

csx_RequestConfiguration() establishes the configuration in the socket adapter and PC
Card, and it programs the Base and Limit registers of multi-function PC Cards if these
registers exist. The values programmed into these registers depend on the I/O require-
ments of this configuration.

STRUCTURE

MEMBERS

The structure members of config_req_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* configuration attributes */
uint32_t Vcc; /* Vcc value */
uint32_t Vpp1; /* Vpp1 value */
uint32_t Vpp2; /* Vpp2 value */
uint32_t IntType; /* socket interface type - mem or IO */
uint32_t ConfigBase; /* offset from start of AM space */
uint32_t Status; /* value to write to STATUS register */
uint32_t Pin; /* value to write to PRR */
uint32_t Copy; /* value to write to COPY register */
uint32_t ConfigIndex; /* value to write to COR */
uint32_t Present; /* which config registers present */
uint32_t ExtendedStatus; /* value to write to EXSTAT register */

The fields are defined as follows:

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

Attributes This field is bit-mapped. It indicates whether the client wishes the IRQ
resources to be enabled and whether Card Services should ignore the VS
bits on the socket interface. The following bits are defined:
CONF_ENABLE_IRQ_STEERING
Enable IRQ Steering

CONF_VSOVERRIDE
Override VS pins

CONF_ENABLE_IRQ_STEERING
Set to connect the PC Card IREQ line to a system interrupt previously selected by a call to csx_RequestIRQ(9F).

If CONF_ENABLE_IRQ_STEERING is set, once csx_RequestConfiguration() has successfully returned, the client may start receiving IRQ callbacks at the IRQ callback handler established in the call to csx_RequestIRQ(9F).

CONF_VSOVERRIDE
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:

<table>
<thead>
<tr>
<th>SOCKET_INTERFACE_MEMORY</th>
<th>Memory only interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET_INTERFACE_MEMORY_AND_IO</td>
<td>Memory and I/O interface</td>
</tr>
</tbody>
</table>

ConfigBase
This field is the offset in bytes from the beginning of attribute memory of the configuration registers.

modified 19 Jul 1996
SunOS 5.6
9F-179
This field identifies which of the configuration registers are present. If present, the corresponding bit is set. This field is bit-mapped as follows:

<table>
<thead>
<tr>
<th>Configuration Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_OPTION_REG_PRESENT</td>
</tr>
<tr>
<td>CONFIG_STATUS_REG_PRESENT</td>
</tr>
<tr>
<td>CONFIG_PINREPL_REG_PRESENT</td>
</tr>
<tr>
<td>CONFIG_COPY_REG_PRESENT</td>
</tr>
<tr>
<td>CONFIG_ESR_REG_PRESENT</td>
</tr>
</tbody>
</table>

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 \texttt{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:

<table>
<thead>
<tr>
<th>Bit Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRR_WP_STATUS WRITE PROTECT mask</td>
</tr>
<tr>
<td>PRR_READY_STATUS READY mask</td>
</tr>
<tr>
<td>PRR_BVD2_STATUS BVD2 mask</td>
</tr>
<tr>
<td>PRR_BVD1_STATUS BVD1 mask</td>
</tr>
<tr>
<td>PRR_WP_EVENT WRITE PROTECT changed</td>
</tr>
<tr>
<td>PRR_READY_EVENT READY changed</td>
</tr>
<tr>
<td>PRR_BVD2_EVENT BVD2 changed</td>
</tr>
<tr>
<td>PRR_BVD1_EVENT BVD1 changed</td>
</tr>
</tbody>
</table>

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.
Kernel Functions for Drivers

RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid or \texttt{csx_RequestConfiguration()} not done.</td>
</tr>
<tr>
<td>CS_BAD_SOCKET</td>
<td>Error in getting or setting socket hardware parameters.</td>
</tr>
<tr>
<td>CS_BAD_VCC</td>
<td>Requested Vcc is not available on socket.</td>
</tr>
<tr>
<td>CS_BAD_VPP</td>
<td>Requested Vpp is not available on socket.</td>
</tr>
<tr>
<td>CS_NO_CARD</td>
<td>No PC Card in socket.</td>
</tr>
<tr>
<td>CS_BAD_TYPE</td>
<td>I/O and memory interface not supported on socket.</td>
</tr>
<tr>
<td>CS_CONFIGURATION_LOCKED</td>
<td>csx_RequestConfiguration() already done.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

CONTEXT

This function may be called from user or kernel context.

SEE ALSO

\texttt{csx_AccessConfigurationRegister(9F)}, \texttt{csx_GetStatus(9F)}, \texttt{csx_RegisterClient(9F)},
\texttt{csx_ReleaseConfiguration(9F)}, \texttt{csx_RequestIO(9F)}, \texttt{csx_RequestIRQ(9F)}

\textit{PC Card 95 Standard}, PCMCIA/JEIDA

modified 19 Jul 1996                       SunOS 5.6                      9F-181
NAME  csx_RequestIO, csx_ReleaseIO – request or release I/O resources for the client

SYNOPSIS  

`#include <sys/pccard.h>`

```c
int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir);
int32_t csx_ReleaseIO(client_handle_t ch, io_req_t *ir);
```

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

- `ch`  Client handle returned from `csx_RegisterClient(9F)`.
- `ir`  Pointer to an `io_req_t` structure.

DESCRIPTION  

The functions `csx_RequestIO()` and `csx_ReleaseIO()` request or release, respectively, I/O resources for the client.

If a client requires I/O resources, `csx_RequestIO()` must be called to request I/O resources from Card Services; then `csx_RequestConfiguration(9F)` must be used to establish the configuration. `csx_RequestIO()` can be called multiple times until a successful set of I/O resources is found. `csx_RequestConfiguration(9F)` only uses the last configuration specified.

`csx_RequestIO()` fails if it has already been called without a corresponding `csx_ReleaseIO()`.

`csx_ReleaseIO()` releases previously requested I/O resources. The Card Services window resource list is adjusted by this function. Depending on the adapter hardware, the I/O window might also be disabled.

STRUCTURE MEMBERS  

The structure members of `io_req_t` are:

```c
uint32_t Socket; /* socket number */

uint32_t Baseport1.base; /* IO range base port address */
acc_handle_t Baseport1.handle; /* IO range base address or port num */
uint32_t NumPorts1; /* first IO range number contiguous ports */
uint32_t Attributes1; /* first IO range attributes */

uint32_t Baseport2.base; /* IO range base port address */
acc_handle_t Baseport2.handle; /* IO range base address or port num */
uint32_t NumPorts2; /* second IO range number contiguous ports */
uint32_t Attributes2; /* second IO range attributes */

uint32_t IOAddrLines; /* number of IO address lines decoded */
```
The fields are defined as follows:

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

**BasePort1.base**

**BasePort1.handle**

**BasePort2.base**

**BasePort2.handle**
Two I/O address ranges can be requested by `csx_RequestIO()`. Each I/O address range is specified by the `BasePort`, `NumPorts`, and `Attributes` fields. If only a single I/O range is being requested, the `NumPorts2` field must be reset to 0.

When calling `csx_RequestIO()`, the `BasePort.base` field specifies the first port address requested. Upon successful return from `csx_RequestIO()`, the `BasePort.handle` field contains an access handle, corresponding to the first byte of the allocated I/O window, which the client must use when accessing the PC Card’s I/O space via the common access functions. A client *must not* make any assumptions as to the format of the returned `BasePort.handle` field value.

If the `BasePort.base` field is set to 0, Card Services returns an I/O resource based on the available I/O resources and the number of contiguous ports requested. When `BasePort.base` is 0, Card Services aligns the returned resource in the host system’s I/O address space on a boundary that is a multiple of the number of contiguous ports requested, rounded up to the nearest power of two. For example, if a client requests two I/O ports, the resource returned will be a multiple of two. If a client requests five contiguous I/O ports, the resource returned will be a multiple of eight.

If multiple ranges are being requested, at least one of the `BasePort.base` fields must be non-zero.

**NumPorts**
This field is the number of contiguous ports being requested.

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

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO_DATA_WIDTH_8</td>
<td>I/O resource uses 8-bit data path</td>
</tr>
<tr>
<td>IO_DATA_WIDTH_16</td>
<td>I/O resource uses 16-bit data path</td>
</tr>
<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>
</tbody>
</table>
WIN_ACC_UNORDERED_OK | May re-order references
WIN_ACC_MERGING_OK | Merge stores to consecutive locations
WIN_ACC_LOADCACHING_OK | May cache load operations
WIN_ACC_STORECACHING_OK | May cache store operations

For some combinations of host system busses and adapter hardware, the width of an I/O resource can not be set via RequestIO(); on those systems, the host bus cycle access type determines the I/O resource data path width on a per-cycle basis.

WIN_ACC_BIG_ENDIAN and WIN_ACC_LITTLE_ENDIAN describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When WIN_ACC_BIG_ENDIAN or WIN_ACC_LITTLE_ENDIAN is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

When WIN_ACC_NEVER_SWAP is specified, byte swapping will not be invoked in the data access functions.

The ability to specify the order in which the CPU will reference data is provided by the following Attributes bits. Only one of the following bits may be specified:

WIN_ACC STRICT_ORDER
The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

WIN_ACC UNORDERED_OK
The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

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

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

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

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

All other bits in the Attributes field must be set to 0.

IOAddrLines
This field is the number of I/O address lines decoded by the PC Card in the specified socket.

On some systems, multiple calls to csx_RequestIO() with different BasePort, NumPorts, and/or IOAddrLines values will have to be made to find an acceptable combination of parameters that can be used by Card Services to allocate I/O resources for the client. (See NOTES).

RETURN VALUES
CS_SUCCESS Successful operation.
CS_BAD_ATTRIBUTE Invalid Attributes specified.
CS_BAD_BASE BasePort value is invalid.
CS_BAD_HANDLE Client handle is invalid.
CS_CONFIGURATION_LOCKED csx_RequestConfiguration(9F) has already been done.
CS_IN_USE csx_RequestIO() has already been done without a corresponding csx_ReleaseIO().
CS_NO_CARD No PC Card in socket.
CS_BAD_WINDOW Unable to allocate I/O resources.
CS_OUT_OF_RESOURCE Unable to allocate I/O resources.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

CONTEXT These functions may be called from user or kernel context.
SEE ALSO csx_RegisterClient(9F), csx_RequestConfiguration(9F)
PC Card 95 Standard, PCMCIA/JEIDA

NOTES It is important for clients to try to use the minimum amount of I/O resources necessary. One way to do this is for the client to parse the CIS of the PC Card and call csx_RequestIO() first with any IOAddrLines values that are 0 or that specify a minimum number of address lines necessary to decode the I/O space on the PC Card. Also, if no convenient minimum number of address lines can be used to decode the I/O space on the PC Card, it is important to try to avoid system conflicts with well-known architectural hardware features.
NAME  csx_RequestIRQ, csx_ReleaseIRQ – request or release IRQ resource

SYNOPSIS  
#include <sys/pccard.h>
int32_t csx_RequestIRQ(client_handle_t ch, irq_req_t *ir);
int32_t csx_ReleaseIRQ(client_handle_t ch, irq_req_t *ir);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  
  ch  Client handle returned from csx_RegisterClient(9F).
  ir  Pointer to an irq_req_t structure.

DESCRIPTION  The function csx_RequestIRQ() requests an IRQ resource and registers the client’s IRQ handler with Card Services.

If a client requires an IRQ, csx_RequestIRQ() must be called to request an IRQ resource as well as to register the client’s IRQ handler with Card Services. The client will not receive callbacks at the IRQ callback handler until csx_RequestConfiguration(9F) or csx_ModifyConfiguration(9F) has successfully returned when either of these functions are called with the CONF_ENABLE_IRQ_STEERING bit set.

The function csx_ReleaseIRQ() releases a previously requested IRQ resource.

The Card Services IRQ resource list is adjusted by csx_ReleaseIRQ(). Depending on the adapter hardware, the host bus IRQ connection might also be disabled. Client IRQ handlers always run above lock level and so should take care to perform only Solaris operations that are appropriate for an above-lock-level IRQ handler.

csx_RequestIRQ() fails if it has already been called without a corresponding csx_ReleaseIRQ().

STRUCTURE MEMBERS  The structure members of irq_req_t are:

uint32_t Socket; /* socket number */
uint32_t Attributes; /* IRQ attribute flags */
csfunction_t *irq_handler; /* IRQ handler */
caddr_t irq_handler_arg; /* IRQ handler argument */
/ddi_iblock_cookie_t *iblk_cookie; /* IRQ interrupt block cookie */
/ddi_idevice_cookie_t *idev_cookie; /* IRQ interrupt device cookie */

The fields are defined as follows:

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

Attributes  This field is bit-mapped. It specifies details about the type of IRQ desired by the client. The following bits are defined:

9F-186  SunOS 5.6  modified 19 Jul 1996
<table>
<thead>
<tr>
<th>Kernel Functions for Drivers</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>csx_RequestIRQ(9F)</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>IRQ_TYPE_EXCLUSIVE</strong></th>
<th>IRQ is exclusive to this socket</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IRQ_ISR_ADDRESS_PROVIDED</strong></td>
<td>IRQ handler address provided</td>
</tr>
</tbody>
</table>

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

**IRQ_ISR_ADDRESS_PROVIDED**
This bit *must* be set. It indicates that the irq_handler field contains the address of the client's IRQ handler.

**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 and IRQ_ISR_ADDRESS_PROVIDED not set. |
| CS_BAD_HANDLE | Client handle is invalid or csx_RequestConfiguration(9F) not done. |
| CS_BAD_IRQ | Unable to allocate IRQ resources. |
| CS_IN_USE | csx_RequestIRQ() already done or a previous csx_RequestIRQ() has not been done for a corresponding csx_ReleaseIRQ(). |
| CS_CONFIGURATION_LOCKED | csx_RequestConfiguration(9F) already done or csx_ReleaseConfiguration(9F) has not been done. |
| CS_NO_CARD | No PC Card in socket. |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed. |

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

**SEE ALSO**
csx_ReleaseConfiguration(9F), csx_RequestConfiguration(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*

modified 19 Jul 1996
SunOS 5.6
9F-187
**NAME**

csx_RequestSocketMask, csx_ReleaseSocketMask – set or clear the client’s client event mask

**SYNOPSIS**

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

int32_t csx_RequestSocketMask(client_handle_t ch, request_socket_mask_t *sm);
int32_t csx_ReleaseSocketMask(client_handle_t ch, release_socket_mask_t *rm);
```

**INTERFACE LEVEL**

Solaris DDI Specific (Solaris DDI)

**ARGUMENTS**

- **ch** Client handle returned from csx_RegisterClient(9F).
- **sm** Pointer to a request_socket_mask_t structure.
- **rm** Pointer to a release_socket_mask_t structure.

**DESCRIPTION**

The function csx_RequestSocketMask() sets the client’s client event mask and enables the client to start receiving events at its event callback handler. Once this function returns successfully, the client can start receiving events at its event callback handler. Any pending events generated from the call to csx_RegisterClient(9F) will be delivered to the client after this call as well. This allows the client to set up the event handler mutexes before the event handler gets called.

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

The function csx_ReleaseSocketMask() clears the client’s client event mask.

**STRUCTURE MEMBERS**

The structure members of request_socket_mask_t are:

- `uint32_t Socket;` /* socket number */
- `uint32_t EventMask;` /* event mask to set or return */

The structure members of release_socket_mask_t are:

- `uint32_t Socket;` /* socket number */

The fields are defined as follows:

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

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

**RETURN VALUES**

- **CS_SUCCESS**: Successful operation.
- **CS_BAD_HANDLE**: Client handle is invalid.
- **CS_IN_USE**: `csx_ReleaseSocketMask()` has not been done.
- **CS_BAD_SOCKET**: `csx_RequestSocketMask()` has not been done.
- **CS_UNSUPPORTED_FUNCTION**: No PCMCIA hardware installed.

**CONTEXT**

These functions may be called from user or kernel context.

**SEE ALSO**

- `csx_event_handler(9E)`, `csx_GetEventMask(9F)`, `csx_RegisterClient(9F)`, `csx_SetEventMask(9F)`

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996

SunOS 5.6

9F-189
NAME | csx_RequestWindow, csx_ReleaseWindow – request or release window resources

SYNOPSIS | #include <sys/pccard.h>

int32_t csx_RequestWindow(client_handle_t ch, window_handle_t *wh,
                           win_req_t *wr);

int32_t csx_ReleaseWindow(window_handle_t wh);

INTERFACE LEVEL | Solaris DDI Specific (Solaris DDI)

ARGUMENTS | ch Client handle returned from csx_RegisterClient(9F).
wh Pointer to a window_handle_t structure.
wr Pointer to a win_req_t structure.

DESCRIPTION | The function csx_RequestWindow() requests a block of system address space be assigned to a PC Card in a socket.
The function csx_ReleaseWindow() releases window resources which were obtained by a call to csx_RequestWindow(). No adapter or socket hardware is modified by this function.
The csx_MapMemPage(9F) and csx_ModifyWindow(9F) functions use the window handle returned by csx_RequestWindow(). This window handle must be freed by calling csx_ReleaseWindow() when the client is done using this window.
The PC Card Attribute or Common Memory offset for this window is set by csx_MapMemPage(9F).

STRUCTURE MEMBERS | The structure members of win_req_t are:
uint32_t Socket; /* socket number */
uint32_t Attributes; /* window flags */
uint32_t Base.base; /* requested window base address */
acc_handle_t Base.handle; /* returned handle for base of window */
uint32_t win_params.AccessSpeed; /* window access speed */
uint32_t win_params.IOAddrLines; /* IO address lines decoded */
uint32_t ReqOffset; /* required window offset */

The fields are defined as follows:
Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
Attributes This field is bit-mapped. It is defined as follows:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIN_MEMORY_TYPE_IO</td>
<td>Window points to I/O space</td>
</tr>
<tr>
<td>WIN_MEMORY_TYPE_CM</td>
<td>Window points to Common Memory space</td>
</tr>
<tr>
<td>WIN_MEMORY_TYPE_AM</td>
<td>Window points to Attribute Memory space</td>
</tr>
<tr>
<td>WIN_ENABLE</td>
<td>Enable window</td>
</tr>
<tr>
<td>WIN_DATA_WIDTH_8</td>
<td>Set window to 8-bit data path</td>
</tr>
<tr>
<td>WIN_DATA_WIDTH_16</td>
<td>Set window to 16-bit data path</td>
</tr>
<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>
</tbody>
</table>

**WIN_MEMORY_TYPE_IO, WIN_MEMORY_TYPE_CM, WIN_MEMORY_TYPE_AM**

These bits select which type of window is being requested. One of these bits must be set.

**WIN_ENABLE**

The client must set this bit to enable the window.

**WIN_ACC_BIG_ENDIAN, WIN_ACC_LITTLE_ENDIAN**

These bits describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When either of these bits are set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

**WIN_ACC_NEVER_SWAP**

When this is specified, byte swapping will not be invoked in the data access functions.

The ability to specify the order in which the CPU will reference data is provided by the following *Attributes* bits, only one of which may be specified:
WIN_ACC STRICT_ORDER
The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

WIN_ACC_UNORDERED_OK
The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

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 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() 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Reserved - do not use)</td>
</tr>
<tr>
<td>1</td>
<td>250 nsec</td>
</tr>
<tr>
<td>2</td>
<td>200 nsec</td>
</tr>
<tr>
<td>3</td>
<td>150 nsec</td>
</tr>
<tr>
<td>4</td>
<td>100 nsec</td>
</tr>
<tr>
<td>5 - 7</td>
<td>(Reserved - do not use)</td>
</tr>
</tbody>
</table>

To request a window that supports the `WAIT` signal, OR-in the `WIN_USE_WAIT` bit to the `AccessSpeed` value before calling this function.

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

**win_params.IOAddrLines**

If the client is requesting an I/O window, the `IOAddrLines` field is the number of I/O address lines decoded by the PC Card in the specified socket. Access to the I/O window is not enabled until `csx_RequestConfiguration(9F)` has been invoked successfully.

**ReqOffset**

This field is a Solaris-specific extension that can be used by clients to generate optimum window offsets passed to `csx_MapMemPage(9F)`.

**RETURN VALUES**

- **CS_SUCCESS**: Successful operation.
- **CS_BAD_ATTRIBUTE**: Attributes are invalid.
- **CS_BAD_SPEED**: Speed is invalid.
- **CS_BAD_HANDLE**: Client handle is invalid.
- **CS_BAD_SIZE**: Window size is invalid.
- **CS_NO_CARD**: No PC Card in socket.
- **CS_OUT_OFRESOURCE**: 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
Kernel Functions for Drivers

NAME
csx_ResetFunction – reset a function on a PC card

SYNOPSIS
#include <sys/pccard.h>
int32_t csx_ResetFunction(client_handle_t ch, reset_function_t *rf);

INTERFACE
LEVEL
Solaris DDI Specific (Solaris DDI)

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

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

STRUCTURE
MEMBERS
The structure members of reset_function_t are:

  uint32_t Socket;    /* socket number */
  uint32_t Attributes; /* reset attributes */

The fields are defined as follows:
Socket Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
Attributes Must be 0.

RETURN VALUES
CS_SUCCESS Card Services has noted the reset request.
CS_IN_USE This Card Services implementation does not permit configured cards to be reset.
CS_BAD_HANDLE Client handle is invalid.
CS_NO_CARD No PC card in socket.
CS_BAD_SOCKET Specified socket or function number is invalid.
CS_UNSUPPORTED_FUNCTION No PCMCIA hardware installed.

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

SEE ALSO
csx_event_handler(9E), csx_RegisterClient(9F)
PC Card 95 Standard, PCMCIA/JEIDA

NOTES
csx_ResetFunction() has not been implemented in this release and always returns CS_IN_USE.

modified 19 Jul 1996

SunOS 5.6

9F-195
NAME  csx_SetEventMask, csx_GetEventMask – set or return the client event mask for the client

SYNOPSIS
#include <sys/pccard.h>
int32_t csx_SetEventMask(client_handle_t ch, sockevent_t *se);
int32_t csx_GetEventMask(client_handle_t ch, sockevent_t *se);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

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

DESCRIPTION
The function csx_SetEventMask() sets the client or global event mask for the client.
The function csx_GetEventMask() returns the client or global event mask for the client.
csx_RequestSocketMask(9F) must be called before calling csx_SetEventMask() for the
client event mask for this socket.

STRUCTURE MEMBERS
The structure members of sockevent_t are:

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

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONF_EVENT_MASK_GLOBAL</td>
<td>Client’s global event mask</td>
</tr>
<tr>
<td>CONF_EVENT_MASK_CLIENT</td>
<td>Client’s local event mask</td>
</tr>
</tbody>
</table>

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

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

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

Socket     Not used in Solaris, but for portability with other Card Services implemen-
            tations, it should be set to the logical socket number.
## RETURN VALUES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>CS_BAD_HANDLE</td>
<td>Client handle is invalid.</td>
</tr>
<tr>
<td>CS_BAD_SOCKET</td>
<td><code>csx_RequestSocketMask(9F)</code> not called for CONF_EVENT_MASK_CLIENT.</td>
</tr>
<tr>
<td>CS_UNSUPPORTED_FUNCTION</td>
<td>No PCMCIA hardware installed.</td>
</tr>
</tbody>
</table>

## CONTEXT

These functions may be called from user or kernel context.

## SEE ALSO

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

*PC Card 95 Standard, PCMCIA/JEIDA*

---

modified 19 Jul 1996

SunOS 5.6

9F-197
<table>
<thead>
<tr>
<th>NAME</th>
<th>csx_SetHandleOffset – set current access handle offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/pccard.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int32_t csx_SetHandleOffset(acc_handle_t handle, uint32_t offset);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Solaris DDI Specific (Solaris DDI)</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td>handle Access handle returned by csx_RequestIRQ(9F) or csx_RequestIO(9F).</td>
</tr>
<tr>
<td></td>
<td>offset New access handle offset.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This function sets the current offset for the access handle, handle, to offset.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>CS_SUCCESS Successful operation.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>This function may be called from user or kernel context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>csx_GetHandleOffset(9F), csx_RequestIO(9F), csx_RequestIRQ(9F)</td>
</tr>
<tr>
<td></td>
<td>PC Card 95 Standard, PCMCIA/JEIDA</td>
</tr>
</tbody>
</table>
NAME  csx_ValidateCIS – validate the Card Information Structure (CIS)

SYNOPSIS  
#include <sys/pccard.h>
int32_t csx_ValidateCIS(client_handle_t ch, cisinfo_t *ci);

INTERFACE LEVEL  Solaris DDI Specific (Solaris DDI)

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

*ci  Pointer to a cisinfo_t structure.

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

STRUCTURE MEMBERS  The structure members of cisinfo_t are:

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

The fields are defined as follows:

- Socket  Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
- Chains  This field returns the number of valid tuple chains located in the CIS. If 0 is returned, the CIS is not valid.
- Tuples  This field is a Solaris-specific extension and it returns the total number of tuples on all the chains in the PC Card’s CIS.

RETURN VALUES  
- CS_SUCCESS  Successful operation.
- CS_NO_CIS  No CIS on PC Card or CIS is invalid.
- CS_NO_CARD  No PC Card in socket.
- CS_UNSUPPORTED_FUNCTION  No PCMCIA hardware installed.

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

SEE ALSO  csx_GetFirstTuple(9F), csx_GetTupleData(9F), csx_ParseTuple(9F), csx_RegisterClient(9F)

PC Card 95 Standard, PCMCIA/JEIDA

modified 19 Jul 1996  SunOS 5.6  9F-199
NAME

datamsg – test whether a message is a data message

SYNOPSIS

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

int datamsg(unsigned char type);

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

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

DESCRIPTION

datamsg() tests the type of message to determine if it is a data message type (M_DATA, M_DELAY, M_PROTO, or M_PCPROTO).

RETURN VALUES

datamsg returns

1 if the message is a data message

0 otherwise.

CONTEXT

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

EXAMPLES

The put(9E) routine enqueues all data messages for handling by the srv(9E) (service) routine. All non-data messages are handled in the put(9E) routine.

1   xxx put(q, mp)
2    queue_t *q;
3   mblk_t *mp;
4 {
5      if (datamsg(mp->b_datap->db_type)) {
6          putq(q, mp);
7          return;
8      }
9    switch (mp->b_datap->db_type) {
10       case M_FLUSH:
11          ...
12   }

SEE ALSO

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

Writing Device Drivers
STREAMS Programming Guide
NAME

ddi_add_intr, ddi_get_iblock_cookie, ddi_remove_intr – hardware interrupt handling routines

SYNOPSIS

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

int ddi_get_iblock_cookie(dev_info_t *dip, u_int inumber,
                          ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_intr(dev_info_t *dip, u_int inumber, ddi_iblock_cookie_t *iblock_cookiep,
                  ddi_idevice_cookie_t *idevice_cookiep, u_int (*int_handler)(caddr_t),
                  caddr_t int_handler_arg);

void ddi_remove_intr(dev_info_t *dip, u_int inumber,
                      ddi_iblock_cookie_t iblock_cookie);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ddi_get_iblock_cookie()

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookiep Pointer to an interrupt block cookie.

ddi_add_intr()

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookiep Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
idevice_cookiep Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.
int_handler Pointer to interrupt handler.
int_handler_arg Argument for interrupt handler.

ddi_remove_intr()

dip Pointer to dev_info structure.
inumber Interrupt number.
iblock_cookie Block cookie which identifies the interrupt handler to be removed.

DESCRIPTION

ddi_get_iblock_cookie() 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 which interrupt specification to retrieve the cookie for. inumber is associated with information provided either by the device (see sbus(4)) or the hardware configuration file (see vme(4),

modified 4 Oct 1996

SunOS 5.6
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`; if the interrupt information cannot be found, it will return `DDI_INTR_NOTFOUND`.

### ddi_remove_intr()

`ddi_remove_intr()` removes an interrupt handler from the system. Unloadable drivers should call this routine during their `detach(9E)` routine to remove their interrupt handler from the system.

The device interrupt routine for this instance of the device will not execute after `ddi_remove_intr()` returns. `ddi_remove_intr()` may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to `ddi_remove_intr()` or deadlock may result.

### RETURN VALUES

`ddi_add_intr()` and `ddi_get_iblock_cookie()` return:

- `DDI_SUCCESS` on success.
- `DDI_INTR_NOTFOUND` on failure to find the interrupt.

### CONTEXT

`ddi_add_intr()`, `ddi_remove_intr()`, and `ddi_get_iblock_cookie()` can be called from user or kernel context.
SEE ALSO  

- driver.conf(4), eisa(4), isa(4), mca(4), sbus(4), sysbus(4), vme(4), attach(9E), detach(9E), ddi_intr_hilevel(9F), mutex(9F), mutex_init(9F), rw_init(9F), rwlock(9F), ddi_idevice_cookie(9S)

Writing Device Drivers

NOTES  

_ddi_get_iblock_cookie() must not be called after the driver adds an interrupt handler for the interrupt specification corresponding to _inumber_.

BUGS  

The _idevice_cookie_ should really point to a data structure that is specific to the bus architecture that the device operates on. Currently only VMEbus and SBus are supported and a single data structure is used to describe both.
ddi_add_softintr (9F) Kernel Functions for Drivers

NAME ddi_add_softintr, ddi_get_soft_iblock_cookie, ddi_remove_softintr, ddi_trigger_softintr
   − software interrupt handling routines

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

int ddi_get_soft_iblock_cookie(dev_info_t *dip, int preference,
       ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t *idp,
       ddi_iblock_cookie_t *iblock_cookiep, ddi_device_cookie_t *device_cookiep,
       u_int(*int_handler)(caddr_t int_handler_arg), caddr_t int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);
void ddi_trigger_softintr(ddi_softintr_t id);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS
ddi_get_soft_iblock_cookie()

   dip Pointer to a dev_info structure.
   preference The type of soft interrupt to retrieve the cookie for.
   iblock_cookiep Pointer to a location to store the interrupt block cookie.

ddi_add_softintr()

   dip Pointer to dev_info structure.
   preference A hint value describing the type of soft interrupt to generate.
   idp Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.
   iblock_cookiep Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
   device_cookiep Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).
   int_handler Pointer to interrupt handler.
   int_handler_arg Argument for interrupt handler.

ddi_remove_softintr()

   id The identifier specifying which soft interrupt handler to remove.

ddi_trigger_softintr()

   id The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.

DESCRIPTION
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.

9F-204 SunOS 5.6 modified 13 Oct 1994
Kernel Functions for Drivers

ddi_add_softintr(9F) adds a soft interrupt to the system. The user specified hint preference identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for preference should be the same as that used in the corresponding call to ddi_get_soft_iblock_cookie(). Refer to the description of ddi_get_soft_iblock_cookie() above.

The value returned in the location pointed at by idp is the soft interrupt identifier. This value is used in later calls to ddi_remove_softintr() and ddi_trigger_softintr() to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by iblock_cookiep is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see mutex_init(9F) and rw_init(9F)). Note that the interrupt block cookie is normally obtained using ddi_get_soft_iblock_cookie() to avoid the race conditions described above (refer to the description of ddi_get_soft_iblock_cookie() above). For this reason, iblock_cookiep is no longer useful and should be set to NULL.

idevice_cookiep is not used and should be set to NULL.

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

The interrupt handler must return DDI_INTR_CLAIMED if the interrupt was claimed, DDI_INTR_UNCLAIMED otherwise.
If successful, \texttt{ddi_add_softintr}() will return \texttt{DDI_SUCCESS}; if the interrupt information cannot be found, it will return \texttt{DDI_FAILURE}.

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

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

\textbf{RETURN VALUES} \texttt{ddi_add_softintr}() and \texttt{ddi_get_soft_iblock_cookie}() return:

- \texttt{DDI_SUCCESS} on success
- \texttt{DDI_FAILURE} on failure

\textbf{CONTEXT} These functions can be called from user or kernel context. \texttt{ddi_trigger_softintr}() may be called from high-level interrupt context as well.

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

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.

\begin{verbatim}
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;
    ...
};
\end{verbatim}
struct xxstate *xsp;
static u_int xxsoftintr(caddr_t);
static u_int xxhighintr(caddr_t);
...

The following code fragment would usually appear in the driver's attach routine.

`ddi_add_intr` is used to add the high-level interrupt handler and `ddi_add_softintr()` is used to add the low-level interrupt routine.

```c
static u_int
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 */
    }

    /* 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 */
    }
```

modified 13 Oct 1994
The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, ddi_trigger_softintr() is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```c
static u_int
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)
        ddi_trigger_softintr(xsp->id);
    ...
    return (DDI_INTRCLAIMED);
}

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

    return (DDI_FAILURE); /* fail attach */
    }
...
ifdef 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_intr_hilevel(9F), ddi_remove_intr(9F), mutex_init(9F)

Writing Device Drivers

NOTES ddi_add_softintr() may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for int_handler_arg in each of the calls to ddi_add_softintr(). Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For example, the argument could be a pointer to the device’s soft state structure, which could contain a 'which_service' field that the handler examines. The driver must set this field to the appropriate value before calling ddi_trigger_softintr().
<table>
<thead>
<tr>
<th>NAME</th>
<th>ddi_binding_name, ddi_get_name – return driver binding name</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/ddi.h>  
#include <sys/sunddi.h>  
char *ddi_binding_name(dev_info_t *dip);  
char *ddi_get_name(dev_info_t *dip); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| ARGUMENTS | dip A pointer to the device’s dev_info structure. |
| DESCRIPTION | ddi_binding_name() and ddi_get_name() return the driver binding name. This is the name used to select a driver for the device. This name is typically derived from the device name property or the device compatible property. The name returned may be a driver alias or the driver name. |
| RETURN VALUES | ddi_binding_name() and ddi_get_name() return the name used to bind a driver to a device. |
| CONTEXT | ddi_binding_name() and ddi_get_name() can be called from user, kernel, or interrupt context. |
| SEE ALSO | ddi_node_name(9F)  
Writing Device Drivers |
| WARNINGS | The name returned by ddi_binding_name() and ddi_get_name(9F) is read-only. |
**NAME**  
ddi_btop, ddi_btopr, ddi_ptob – page size conversions

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

unsigned long ddi_btop(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_btopr(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_ptob(dev_info_t *dip, unsigned long pages);
```

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

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

- **ddi_btop()** converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.
- **ddi_btopr()** converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.
- **ddi_ptob()** converts the given number of pages to the number of bytes that it corresponds to.

Because bus nexus may possess their own hardware address translation facilities, these routines should be used in preference to the corresponding DDI/DKI routines btop(9F), btopr(9F), and ptob(9F), which only deal in terms of the pagesize of the main system MMU.

**RETURN VALUES**  
- **ddi_btop()** and **ddi_btopr()** return the number of corresponding pages.  
  **ddi_ptob()** returns the corresponding number of bytes.  
  There are no error return values.

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

**EXAMPLES**  
This example finds the size (in bytes) of one page:
```
pagesize = ddi_ptob(dip, 1L);
```

**SEE ALSO**  
- btop(9F), btopr(9F), ptob(9F)

*Writing Device Drivers*

---

modified 11 Sep 1991    SunOS 5.6    9F-211
| NAME | ddi_copyin – copy data to a driver buffer |
| SYNOPSIS | #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>  
int ddi_copyin(const void ∗buf, void ∗driverbuf, size_t cn, int flags); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| ARGUMENTS | buf | Source address from which data is transferred. |
| | driverbuf | Driver destination address to which data is transferred. |
| | cn | Number of bytes transferred. |
| | flags | Set of flag bits that provide address space information about buf. |
| DESCRIPTION | This routine is designed for use in driver ioctl(9E) routines for drivers that support layered ioctls. ddi_copyin() copies data from a source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address. The flags argument is used to determine the address space information about buf. If the FKIOCTL flag is set, this indicates that buf is a kernel address, and ddi_copyin() behaves like bcopy(9F). Otherwise buf is interpreted as a user buffer address, and ddi_copyin() behaves like copyin(9F). Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move according to address alignment. |
| RETURN VALUES | ddi_copyin() returns 0, indicating a successful copy. It returns −1 if one of the following occurs: |
| | • paging fault; the driver tried to access a page of memory for which it did not have read or write access |
| | • invalid user address, such as a user area or stack area |
| | • invalid address that would have resulted in data being copied into the user block |
| | If a −1 is returned to the caller, driver entry point routines should return EFAULT. |
| CONTEXT | ddi_copyin() can be called from user or kernel context only. |
| EXAMPLES | 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. |
struct device {
    /* layout of physical device registers */
    int control;       /* physical device control word */
    int status;        /* physical device status word */
    short recv_char;   /* receive character from device */
    short xmit_char;   /* transmit character to device */
};

struct device_state {
    volatile struct device *regsp; /* pointer to device registers */
    kmutex_t reg_mutex; /* protect device registers */
    ...
}

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

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

    instance = getminor(dev);
    sp = ddi_get_soft_state(statep, instance);
    if (sp == NULL)
        return (ENXIO);
    rp = sp->regsp;
    ...

    switch (cmd) {
    case XX_SETREGS: /* copy data to temp. regs. buf */
        if (ddi_copyin(arg, &reg_buf,
                       sizeof (struct device), mode) != 0) {
            return (EFAULT);
        }
        mutex_enter(&sp->reg_mutex);
        /* Copy data from temporary device register
         * buffer to device registers.
         * e.g. rp->control = reg_buf.control;
         */
        mutex_exit(&sp->reg_mutex);
    }
}
```
37    break;
38    }
39  }

SEE ALSO  ioctl(9E), bcopy(9F), copyin(9F), copyout(9F), ddi_copyout(9F), uio-move(9F)

NOTES  The value of the flags argument to ddi_copyin() should be passed through directly from
       the mode argument of ioctl() untranslated.
       Driver defined locks should not be held across calls to this function.
```
NAME  ddi_copyout – copy data from a driver

SYNOPSIS  

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

int ddi_copyout(const void *driverbuf, void *buf, size_t cn, int flags);
```

INTERFACE  Solaris DDI specific (Solaris DDI).

LEVEL  ddi_copyout()

ARGUMENTS  

- `driverbuf`  Source address in the driver from which the data is transferred.
- `buf`  Destination address to which the data is transferred.
- `cn`  Number of bytes to copy.
- `flags`  Set of flag bits that provide address space information about `buf`.

DESCRIPTION  This routine is designed for use in driver ioctl(9E) routines for drivers that support layered ioctls. ddi_copyout() copies data from a driver buffer to a destination address, `buf`. The `flags` argument is used to determine the address space information about `buf`. If the FKOCTL flag is set, this indicates that `buf` is a kernel address, and `ddi_copyout()` behaves like bcopy(9F). Otherwise `buf` is interpreted as a user buffer address, and `ddi_copyout()` behaves like copyout(9F).

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

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

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

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

CONTEXT  ddi_copyout() can be called from user or kernel context only.

EXAMPLES  A driver ioctl(9E) routine (line 12) can be used to get or set device attributes or registers. In the XX_GETREGS condition (line 25), the driver copies the current device register values to another data area. If the specified argument contains an invalid address, an error code is returned.

```c
1 struct device { /* layout of physical device registers */
2   int control; /* physical device control word */
3   int status; /* physical device status word */
```

modified 1 May 1996  SunOS 5.6  9F-215
4 shortrecv_char; /* receive character from device */
5 short xmit_char; /* transmit character to device */
6 ;

7 struct device_state {
8 volatile struct device *regsp; /* pointer to device registers */
9 kmutex_t reg_mutex; /* protect device registers */
  ...
10 };

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

12 xxioctl(dev_t dev, int cmd, int arg, int mode,
13 cred_t *cred_p, int *rval_p)
14 {
15 struct device_state *sp;
16 volatile struct device *rp;
17 struct device reg_buf; /* temporary buffer for registers */
18 int instance;
19 instance = getminor(dev);
20 sp = ddi_get_soft_state(statep, instance);
21 if (sp == NULL)
22 return (ENXIO);
23 rp = sp->regsp;
  ...
24   switch (cmd) {
25   case XX_GETREGS: /* copy registers to arg */
26     mutex_enter(&sp->reg_mutex);
27     /*
28      * Copy data from device registers to
29      * temporary device register buffer
30      * e.g. reg_buf.control = rp->control;
31      */
32     mutex_exit(&sp->reg_mutex);
33     if (ddi_copyout(&reg_buf, arg,
34         sizeof (struct device), mode) != 0) {
35         return (EFAULT);
36     }
37     break;
38   }
39 }
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_create_minor_node (9F)  Kernel Functions for Drivers

NAME  
ddi_create_minor_node – create a minor node for this device

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

int ddi_create_minor_node(dev_info_t *dip, char *name, int spec_type, int minor_num,
char *node_type, int is_clone);

INTERFACE LEVEL ARGUMENTS  
Solaris DDI specific (Solaris DDI).

dip    A pointer to the device's dev_info structure.
name   The name of this particular minor device.
spec_type S_IFCHR or S_IFBLK for character or block minor devices respectively.
minor_num The minor number for this particular minor device.
node_type Any string that uniquely identifies the type of node. The following predefined node types are provided with this release:

DDI_NT_SERIAL    For serial ports
DDI_NT_SERIAL_MB For on board serial ports
DDI_NT_SERIAL_DO For dial out ports
DDI_NT_SERIAL_MB_DO For on board dial out ports
DDI_NT_BLOCK     For hard disks
DDI_NT_BLOCK_CHAN For hard disks with channel or target numbers
DDI_NT_CD        For CDROM drives
DDI_NT_CD_CHAN   For CDROM drives with channel or target numbers
DDI_NT_FD        For floppy disks
DDI_NT_TAPE      For tape drives
DDI_NT_NET       For network devices
DDI_NT_DISPLAY   For display devices
DDI_PSEUDO       For pseudo devices

is_clone    If the device is a clone device then this flag is set to CLONE_DEV else it is set to 0.

DESCRIPTION  
ddi_create_minor_node() provides the necessary information to enable the system to create the /dev and /devices hierarchies. The name is used to create the minor name of the block or character special file under the /devices hierarchy. At sign (@), slash (/), and space are not allowed. The spec_type specifies whether this is a block or character device. The minor_num is the minor number for the device. The node_type is used to create the names in the /dev hierarchy that refers to the names in
the /devices hierarchy. See disks(1M), ports(1M), tapes(1M), devlinks(1M). Finally
is_clone determines if this is a clone device or not.

**RETURN VALUES**

`ddi_create_minor_node()` returns:

- **DDI_SUCCESS** if it 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** if minor node creation failed.

**EXAMPLES**

The following example creates a data structure describing a minor device called `foo`
which has a minor number of 0. It is of type **DDI_NT_BLOCK** (a block device) and it is
not a clone device.

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

**SEE ALSO**

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

*Writing Device Drivers*
ddi_device_copy – copy data from one device register to another device register

SYNOPSIS

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

int ddi_device_copy(ddi_acc_handle_t src_handle, caddr_t src_addr, ssize_t src_advcnt, 
                     ddi_acc_handle_t dest_handle, caddr_t dest_addr, ssize_t dest_advcnt, 
                     size_t bytecount, uint_t dev_datasz);
```

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

- `src_handle`: The data access handle of the source device.
- `src_addr`: Base data source address.
- `src_advcnt`: Number of `dev_datasz` units to advance on every access.
- `dest_handle`: The data access handle of the destination device.
- `dest_addr`: Base data destination address.
- `dest_advcnt`: Number of `dev_datasz` units to advance on every access.
- `bytecount`: Number of bytes to transfer.
- `dev_datasz`: The size of each data word. Possible values are defined as:
  - `DDI_DATA_SZ01_ACC`: 1 byte data size
  - `DDI_DATA_SZ02_ACC`: 2 bytes data size
  - `DDI_DATA_SZ04_ACC`: 4 bytes data size
  - `DDI_DATA_SZ08_ACC`: 8 bytes data size

DESCRIPTION

`ddi_device_copy()` copies `bytecount` bytes from the source address, `src_addr`, to the destination address, `dest_addr`. The attributes encoded in the access handles, `src_handle` and `dest_handle`, govern how data is actually copied from the source to the destination. Only matching data sizes between the source and destination are supported.

Data will automatically be translated to maintain a consistent view between the source and the destination. The translation may involve byte-swapping if the source and the destination devices have incompatible endian characteristics.

The `src_advcnt` and `dest_advcnt` arguments specify the number of `dev_datasz` units to advance with each access to the device addresses. A value of 0 will use the same source and destination device address on every access. A positive value increments the corresponding device address by certain number of data size units in the next access. On the other hand, a negative value decrements the device address.
The `dev_datasz` argument determines the size of the data word on each access. The data size must be the same between the source and destination.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>ddi_device_copy() returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_SUCCESS</td>
</tr>
<tr>
<td>DDI_FAILURE</td>
</tr>
</tbody>
</table>

**CONTEXT**

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

**SEE ALSO**

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

*Writing Device Drivers*
NAME

ddi_device_zero ( 9F ) Kernel Functions for Drivers

SYNOPSIS

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

int ddi_device_zero(ddi_acc_handle_t handle, caddr_t dev_addr, size_t bytecount, ssize_t dev_advcnt, uint_t dev_datasz);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Beginning of the device address.

bytecount Number of bytes to zero.

dev_advcnt Number of dev_datasz units to advance on every access.

dev_datasz The size of each data word. Possible values are defined as:

<table>
<thead>
<tr>
<th>dev_datasz</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_DATA_SZ01_ACC</td>
<td>1 byte</td>
</tr>
<tr>
<td>DDI_DATA_SZ02_ACC</td>
<td>2 bytes</td>
</tr>
<tr>
<td>DDI_DATA_SZ04_ACC</td>
<td>4 bytes</td>
</tr>
<tr>
<td>DDI_DATA_SZ08_ACC</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

DESCRIPTION

ddi_device_zero() function fills the given, bytecount, number of byte of zeroes to the device register or memory.

The dev_advcnt argument determines the value of the device address, dev_addr, on each access. A value of 0 will use the same device address, dev_addr, on every access. A positive value increments the device address in the next access while a negative value decrements the address. The device address is incremented and decremented in dev_datasz units.

The dev_datasz argument determines the size of data word on each access.

RETURN VALUES

ddi_device_zero() returns:

- DDI_SUCCESS Successfully zeroed the data.
- DDI_FAILURE The byte count is not a multiple of dev_datasz.

CONTEXT

ddi_device_zero() can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_regs_map_free(9F), ddi_regs_map_setup(9F)

Writing Device Drivers

9F-222 SunOS 5.6 modified 25 Sep 1996
NAME

ddi_devid_compare, ddi_devid_free, ddi_devid_init, ddi_devid_register,
ddi_devid_sizeof, ddi_devid_unregister, ddi_devid_valid – Kernel interfaces for device
ids

SYNOPSIS

int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);
size_t ddi_devid_sizeof(ddi_devid_t devid);
int ddi_devid_init(dev_info_t *dip, u_short devid_type, u_short 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);
void ddi_devid_unregister(dev_info_t *dip);
int ddi_devid_valid(ddi_devid_t devid);

ARGUMENTS

devid The device id address.
devid1 The first of two device id addresses to be compared calling
   ddi_devid_compare().
devid2 The second of two device id addresses to be compared calling
   ddi_devid_compare().
dip A dev_info pointer, which identifies the device.
devid_type The following device id types may be accepted by the ddi_devid_init() function:
   DEVID_SCSI3_WWN World Wide Name associated with SCSI-3 devices.
   DEVID_SCSI_SERIAL Vendor ID and serial number associated with a
   SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device
   id must be used.
   DEVID_ENCAP Device id of another device. This is for layered device driver usage.
   DEVID_FAB Fabricated device id.
nbytes The length in bytes of device id.
retdevid The return address of the device id created by ddi_devid_init().

DESCRIPTION

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

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

modified 26 Nov 1996

SunOS 5.6

9F-223
**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 by the **ddi_devid_init()** function.

**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.

**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 device id pointed to by `devid1` is less than the device id pointed to by `devid2`.
- **0** The device id pointed to by `devid1` is equal to the device id pointed to by `devid2`.
- **1** The device id pointed to by `devid1` is greater than the device id pointed to by `devid2`.

**ddi_devid_sizeof( )** returns the size of the `devid` in numbers of bytes.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** `devid_compare(3), devid_deviceid_to_nmlist(3), devid_free(3), devid_free_nmlist(3), devid_get(3), devid_get_minor_name(3), devid_sizeof(3), libdevid(4), attributes(5), attach(9E), detach(9E)`

*Writing Device Drivers*
NAME  ddi_dev_is_needed – inform the system that a device's component is required

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

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

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
dip  A pointer to the device's dev_info structure.
component  The component of the driver which is needed
level  The power level at which the component is needed

DESCRIPTION  The ddi_dev_is_needed() function informs the system that a device component is needed at the specified power level. The level argument must be non-zero.

This function sets a component to the required level and sets all of the devices on which it depends (see pm(7D)) to their normal power levels. If component 0 of the device 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 di_dev_is_needed() returns.

The state of the device should be examined before each physical access. The ddi_dev_is_needed() function should be called to set a component to the required power level if the operation to be performed requires the component to be at a power level other than its current level.

The ddi_dev_is_needed() may cause re-entry of the driver. Deadlock may result if driver locks are held across the call to ddi_dev_is_needed().

RETURN VALUES  The ddi_dev_is_needed() function returns:
DDI_SUCCESS  Power successfully set to the requested level.
DDI_FAILURE  An error occurred.

EXAMPLES  A hypothetical disk driver might include this code:

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

static int
txxdisk_strategy(struct buf *bp)
{
    ...

mutex_enter(&xystate_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(&xystate_lock);
    if (ddi_dev_is_needed(xsp->mydip,
        XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
        bioerror(bp,EIO);
        biodone(bp);
        return (0);
    }
    mutex_enter(&xystate_lock);
}
    xsp->device_busy++;
    mutex_exit(&xystate_lock);

...
### NAME
ddi_dev_is_sid – tell whether a device is self-identifying

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

int ddi_dev_is_sid(dev_info_t *dip);
```

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

### ARGUMENTS
- `dip` A pointer to the device’s `dev_info` structure.

### DESCRIPTION
`ddi_dev_is_sid()` tells the caller whether the device described by `dip` is self-identifying, that is, a device that can unequivocally tell the system that it exists. This is useful for drivers that support both a self-identifying as well as a non-self-identifying variants of a device (and therefore must be probed).

### RETURN VALUES
- `DDI_SUCCESS` Device is self-identifying.
- `DDI_FAILURE` Device is not self-identifying.

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

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

### SEE ALSO
- `probe(9E)`
- *Writing Device Drivers*
NAME ddi_dev_nintrs – return the number of interrupt specifications a device has

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

int ddi_dev_nintrs(dev_info_t *dip, int *resultp);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION ddi_dev_nintrs() returns the number of interrupt specifications a device has in *resultp.

RETURN VALUES ddi_dev_nintrs() returns:

DDI_SUCCESS A successful return. The number of interrupt specifications that the device has is set in resultp.

DDI_FAILURE The device has no interrupt specifications.

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

SEE ALSO isa(4), sbus(4), vme(4), ddi_add_intr(9F), ddi_dev_nregs(9F), ddi_dev_regsize(9F)

Writing Device Drivers

modified 2 Dec 1993 SunOS 5.6 9F-229
**NAME**  
ddi_dev_nregs – return the number of register sets a device has

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

int ddi_dev_nregs(dev_info_t *dip, int *resultp);
```

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

**ARGUMENTS**  
- `dip` A pointer to the device’s `dev_info` structure.
- `resultp` Pointer to an integer that holds the number of register sets on return.

**DESCRIPTION**  
The function `ddi_dev_nregs()` returns the number of sets of registers the device has.

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

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

**SEE ALSO**  
`ddi_dev_nintrs(9F), ddi_dev_regsize(9F)`

*Writing Device Drivers*
NAME
ddi_dev_regsize – return the size of a device’s register

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

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

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

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

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

RETURN VALUES
ddi_dev_regsize() returns:

DDI_SUCCESS A successful return. The size, in bytes, of the specified register, is set in resultp.

DDI_FAILURE An invalid (nonexistent) register number was specified.

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

SEE ALSO
ddi_dev_nintrs(9F), ddi_dev_nregs(9F)
Writing Device Drivers

modified 24 Oct 1991
NAME

ddi_dma_addr_bind_handle (9F)

binds an address to a DMA handle

SYNOPSIS

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

int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, struct as *as, caddr_t addr,
size_t len, uint_t flags, int (*callback) (caddr_t, caddr_t arg,
ddi_dma_cookie_t *cookiep, uint_t *ccountp);

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle

The DMA handle previously allocated by a call to
ddi_dma_alloc_handle (9F).

as

A pointer to an address space structure. This parameter should be set to
NULL, which implies kernel address space.

addr

Virtual address of the memory object.

len

Length of the memory object in bytes.

flags

Valid flags include:
DDI_DMA_WRITE Transfer direction is from memory to I/O.
DDI_DMA_READ Transfer direction is from I/O to memory.
DDI_DMA_RDWR Both read and write.
DDI_DMA_REDZONE Establish an MMU redzone at end of the object.
DDI_DMA_PARTIAL Partial resource allocation.
DDI_DMA_CONSISTENT Nonsequential, random, and small block
transfers.
DDI_DMA_STREAMING Sequential, unidirectional, block-sized, and
block-aligned transfers.

callback

The address of a function to call back later if resources are not currently
available. The following special function addresses may also be used.
DDI_DMA_SLEEP Wait until resources are available.
DDI_DMA_DONTWAIT Do not wait until resources are available and do
not schedule a callback.

arg

Argument to be passed to the callback function, callback, if such a func-
tion is specified.

cookiep

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

ccountp

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

**DESCRIPTION**

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

`ddi_dma_addr_bind_handle()` fills in the first DMA cookie pointed to by `cookiep` with the appropriate address, length, and bus type. `*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)` the number of times specified by `*countp - 1`.

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

The `flags` argument contains information for mapping routines.

- **DDI_DMA_WRITE**
- **DDI_DMA_READ**
- **DDI_DMA_RDWR**
  - These flags describe the intended direction of the DMA transfer.

- **DDI_DMA_STREAMING**
  - This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the `minxfer` and `burst-sizes` 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. Even when `DDI_DMA_PARTIAL` is set, the system may decide to allocate resources for the entire object (less overhead) in which case `DDI_DMA_MAPPED` is returned.

The callback function `callback` indicates how a caller wants to handle the possibility of resources not being available. If `callback` is set to `DDI_DMA_DONTWAIT`, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If `callback` is set to `DDI_DMA_SLEEP`, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set and a DMA resource allocation fails, this value is assumed to be the address of a function to be called when resources become available. When the specified function is called, `arg` is passed to it as an argument. The specified callback function must return either `DDI_DMA_CALLBACK_RUNOUT` or `DDI_DMA_CALLBACK_DONE`.

`DDI_DMA_CALLBACK_RUNOUT` indicates that the callback function attempted to allocate DMA resources but failed. In this case, the callback function is put back on a list to be called again later. `DDI_DMA_CALLBACK_DONE` indicates that either the allocation of DMA resources was successful or the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are be available. The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.

**RETURN VALUES**

`ddi_dma_addr_bind_handle()` returns:

- `DDI_DMA_MAPPED` Successfully allocated resources for the entire object.
- `DDI_DMA_PARTIAL_MAP` Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the `DDI_DMA_PARTIAL` flag in `flags`.
- `DDI_DMA_INUSE` Another I/O transaction is using the DMA handle.
- `DDI_DMA_NORESOURCES` No resources are available at the present time.
- `DDI_DMA_NOMAPPING` The object cannot be reached by the device requesting the resources.
- `DDI_DMA_TOOBIG` The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration.

**CONTEXT**

`ddi_dma_addr_bind_handle()` can be called from user, kernel, or interrupt context, except when `callback` is set to `DDI_DMA_SLEEP`, in which case it can only be called from user or kernel context.
SEE ALSO ddi_dma_alloc_handle(9F), ddi_dma_free_handle(9F), ddi_dma_getwin(9F), ddi_dma_mem_alloc(9F), ddi_dma_mem_free(9F), ddi_dma_nextcookie(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F), ddi_dma_attr(9S), ddi_dma_cookie(9S)

Writing Device Drivers

NOTES If the driver permits partial mapping with the DDI_DMA_PARTIAL flag, the number of cookies in each window may exceed the size of the device’s scatter/gather list as specified in the dma_attr_sgllen field in the ddi_dma_attr(9S) structure. In this case, each set of cookies comprising a DMA window will satisfy the DMA attributes as described in the ddi_dma_attr(9S) structure in all aspects. The driver should set up its DMA engine and perform one transfer for each set of cookies sufficient for its scatter/gather list, up to the number of cookies for this window, before advancing to the next window using ddi_dma_getwin(9F).
**NAME**  
ddi_dma_addr_setup — easier DMA setup for use with virtual addresses

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

int ddi_dma_addr_setup(dev_info_t *dip, struct as *, caddr_t addr, size_t len,
                       u_int flags, int (*waitfp)(caddr_t), caddr_t arg, ddi_dma_lim_t *lim,
                       ddi_dma_handle_t *handlep);
```

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

**ARGUMENTS**
- **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() 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.

**SEE ALSO**  
ddi_dma_buf_setup(9F), ddi_dma_free(9F), ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_iopb_alloc(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

*Writing Device Drivers*
NAME
  ddi_dma_alloc_handle – allocate DMA handle

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

int ddi_dma_alloc_handle(dev_info_t *dip, ddi_dma_attr_t *attr,
    int (*callback) (caddr_t, caddr_t arg, ddi_dma_handle_t *handlep);

INTERFACE LEVEL
  Solaris DDI specific (Solaris DDI).

ARGUMENTS
  dip      Pointer to the device’s dev_info structure.
  attr     Pointer to a DMA attribute structure for this device (see
                  ddi_dma_attr(9S)).
  callback The address of a function to call back later if resources aren’t available
            now.  The following special function addresses may also be used.
            DDI_DMA_SLEEP   Wait until resources are available.
            DDI_DMA_DONTWAIT Do not wait until resources are available and do
                                not schedule a callback.
  arg      Argument to be passed to a callback function, if such a function is
                  specified.
  handlep  Pointer to the DMA handle to be initialized.

DESCRIPTION
  ddi_dma_alloc_handle() allocates a new DMA handle.  A DMA handle is an opaque
  object used as a reference to subsequently allocated DMA resources.
  ddi_dma_alloc_handle() accepts as parameters the device information referred to by dip
  and the device’s DMA attributes described by a ddi_dma_attr(9S) structure.  A successful
  call to ddi_dma_alloc_handle() fills in the value pointed to by handlep.  A DMA handle
  must only be used by the device for which it was allocated and is only valid for one I/O
  transaction at a time.

  The callback function, callback, indicates how a caller wants to handle the possibility of
  resources not being available.  If callback is set to DDI_DMA_DONTWAIT, then the caller
  does not care if the allocation fails, and can handle an allocation failure appropriately.  If
  callback is set to DDI_DMA_SLEEP, then the caller wishes to have the the allocation rou-
  tines wait for resources to become available.  If any other value is set, and a DMA
  resource allocation fails, this value is assumed to be a function to call at a later time when
  resources may become available.  When the specified function is called, it is passed arg as
  an argument.  The specified callback function must return either
  DDI_DMA_CALLBACK_RUNOUT or DDI_DMA_CALLBACK_DONE.
  DDI_DMA_CALLBACK_RUNOUT indicates that the callback routine attempted to allocate
  DMA resources but failed to do so, in which case the callback function is put back on a
  list to be called again later.  DDI_DMA_CALLBACK_DONE indicates either success at allo-
  cating DMA resources or the driver no longer wishes to retry.

modified 22 Sep 1996

SunOS 5.6

9F-237
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, so forth. When a DMA handle is no longer needed, `ddi_dma_free_handle(9F)` must be called to free the handle.

**RETURN VALUES**

`ddi_dma_alloc_handle()` returns:

- **DDI_SUCCESS**
  - Successfully allocated a new DMA handle.

- **DDI_DMA_BADATTR**
  - The attributes specified in the `ddi_dma_attr(9S)` structure make it impossible for the system to allocate potential DMA resources.

- **DDI_DMA_NORESOURCES**
  - No resources are available.

**CONTEXT**

`ddi_dma_alloc_handle()` can be called from user, kernel, or interrupt context, except when `callback` is set to `DDI_DMA_SLEEP`, in which case it can be called from user or kernel context only.

**SEE ALSO**

- `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_buf_bind_handle(9F)`, `ddi_dma_burstsizes(9F)`, `ddi_dma_free_handle(9F)`, `ddi_dma_unbind_handle(9F)`, `ddi_dma_attr(9S)`

*Writing Device Drivers*
NAME  ddi_dma_buf_bind_handle – binds a system buffer to a DMA handle

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

           int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle, struct buf *bp, uint_t flags, int (*callback)(caddr_t, caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  handle  The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).

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

            flags  Valid flags include:

                   DDI_DMA_WRITE  Transfer direction is from memory to I/O

                   DDI_DMA_READ   Transfer direction is from I/O to memory

                   DDI_DMA_RDWR   Both read and write

                   DDI_DMA_REDZONE  Establish an MMU redzone at end of the object.

                   DDI_DMA_PARTIAL  Partial resource allocation

                   DDI_DMA_CONSISTENT  Nonsequential, random, and small block transfers.

                   DDI_DMA_STREAMING  Sequential, unidirectional, block-sized, and block-aligned transfers.

            callback  The address of a function to call back later if resources are not available now. The following special function addresses may also be used.

                   DDI_DMA_SLEEP  wait until resources are available

                   DDI_DMA_DONTWAIT  do not wait until resources are available and do not schedule a callback.

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

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

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

modified 27 Jul 1996  SunOS 5.6  9F-239
ddi_dma_buf_bind_handle(9F) 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.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_DMA_WRITE</td>
<td>These flags describe the intended direction of the DMA transfer.</td>
</tr>
<tr>
<td>DDI_DMA_READ</td>
<td></td>
</tr>
<tr>
<td>DDI_DMA_RDWR</td>
<td></td>
</tr>
<tr>
<td>DDI_DMA_STREAMING</td>
<td>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 burst-sizes 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.</td>
</tr>
<tr>
<td>DDI_DMA_CONSISTENT</td>
<td>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.</td>
</tr>
<tr>
<td>DDI_DMA_REDZONE</td>
<td>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.</td>
</tr>
<tr>
<td>DDI_DMA_PARTIAL</td>
<td>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</td>
</tr>
</tbody>
</table>
Kernel Functions for Drivers

 decide to allocate resources for the entire object (less overhead) in which case DDI_DMA_MAPPED is returned.

The callback function, callback, indicates how a caller wants to handle the possibility of resources not being available. If callback is set to DDI_DMA_DONTWAIT, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If callback is set to DDI_DMA_SLEEP, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be the address of a function to call at a later time when resources may become available. When the specified function is called, it is passed arg as an argument. The specified callback function must return either DDI_DMA_CALLBACK_RUNOUT or DDI_DMA_CALLBACK_DONE.

DDI_DMA_CALLBACK_RUNOUT indicates that the callback function attempted to allocate DMA resources but failed to do so. In this case the callback function is put back on a list to be called again later. DDI_DMA_CALLBACK_DONE indicates either a successful allocation of DMA resources or that the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are be available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, etc.

RETURN VALUES

ddi_dma_buf_bind_handle() returns:

DDI_DMA_MAPPED Successfully allocated resources for the entire object.

DDI_DMA_PARTIAL_SUCCESS Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the DDI_DMA_PARTIAL flag in flags.

DDI_DMA_INUSE Another I/O transaction is using the DMA handle.

DDI_DMA_NORESOURCES No resources are available at the present time.

DDI_DMA_NOMAPPING The object cannot be reached by the device requesting the resources.

DDI_DMA_TOOBIG The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration.

CONTEXT
ddi_dma_buf_bind_handle() can be called from user, kernel, or interrupt context, except when callback is set to DDI_DMA_SLEEP, in which case it can be called from user or kernel context only.

modified 27 Jul 1996

SunOS 5.6

9F-241
SEE ALSO  ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_free_handle(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F),
ddi_dma_sync(9F), ddi_dma_unbind_handle(9F), buf(9S), ddi_dma_attr(9S),
ddi_dma_cookie(9S)

Writing Device Drivers

NOTES  If the driver permits partial mapping with the DDI_DMA_PARTIAL flag, the number of
cookies in each window may exceed the size of the device’s scatter/gather list as
specified in the dma_attr_sglfen field in the ddi_dma_attr(9S) structure. In this case,
each set of cookies comprising a DMA window will satisfy the DMA attributes as
described in the ddi_dma_attr(9S) structure in all aspects. The driver should set up its
DMA engine and perform one transfer for each set of cookies sufficient for its
scatter/gather list, up to the number of cookies for this window, before advancing to the
next window using ddi_dma_getwin(9F).
NAME

ddi_dma_buf_setup – easier DMA setup for use with buffer structures

SYNOPSIS

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

int ddi_dma_buf_setup(dev_info_t *dip, struct buf *bp, u_int flags,
    int (*waitfp)(caddr_t, caddr_t arg, ddi_dma_limit_t *lim,
    ddi_dma_handle_t *handlep);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

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_limit_sparc(9S) or ddi_dma_limit_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.

SEE ALSO

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_limit_sparc(9S), ddi_dma_limit_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
NAME
  ddi_dma_burstsizes – find out the allowed burst sizes for a DMA mapping

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

int ddi_dma_burstsizes(ddi_dma_handle_t handle);

INTERFACE LEVEL
  Solaris DDI specific (Solaris DDI).

ARGUMENTS
  handle A DMA handle that was filled in by a successful call to
          ddi_dma_setup(9F).

DESCRIPTION
  ddi_dma_burstsizes() returns the allowed burst sizes for a DMA mapping. This value is
  derived from the dlim_burstsizes member of the ddi_dma_lim_sparc(9S) structure, but
  it shows the allowable burstsizes after imposing on it the limitations of other device layers
  in addition to device’s own limitations.

RETURN VALUES
  ddi_dma_burstsizes() returns a binary encoded value of the allowable DMA burst sizes.
  See ddi_dma_lim_sparc(9S) for a discussion of DMA burst sizes.

CONTEXT
  This function can be called from user or interrupt context.

SEE ALSO
  ddi_dma_devalign(9F), ddi_dma_setup(9F), ddi_dma_lim_sparc(9S), ddi_dma_req(9S)

Writing Device Drivers
NAME
ddi_dma_coff – convert a DMA cookie to an offset within a DMA handle

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

int ddi_dma_coff(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep, off_t *offp);

INTERFACE LEVEL
Solaris SPARC DDI (Solaris SPARC DDI).

ARGUMENTS
handle The handle filled in by a call to ddi_dma_setup(9F).
cookiep A pointer to a DMA cookie (see ddi_dma_cookie(9S)) that contains the appropriate address, length and bus type to be used in programming the DMA engine.
offp A pointer to an offset to be filled in.

DESCRIPTION
ddi_dma_coff() converts the values in DMA cookie pointed to by cookiep to an offset (in bytes) from the beginning of the object that the DMA handle has mapped.
ddi_dma_coff() allows a driver to update a DMA cookie with values it reads from its device’s DMA engine after a transfer completes and convert that value into an offset into the object that is mapped for DMA.

RETURN VALUES
ddi_dma_coff() returns:

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

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

SEE ALSO
ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S)
Writing Device Drivers
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,
                   u_int *lenp);

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

ARGUMENTS
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.

SEE ALSO ddi_dma_movwin(9F), ddi_dma_setup(9F), ddi_dma_req(9S)
Writing Device Drivers
NAME  ddi_dma_devalign – find DMA mapping alignment and minimum transfer size

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

int ddi_dma_devalign(ddi_dma_handle_t handle, u_int *alignment, u_int *minxfr);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
handle  The DMA handle filled in by a successful call to ddi_dma_setup(9F).
alignment  A pointer to an unsigned integer to be filled in with the minimum required alignment for DMA. The alignment is guaranteed to be a power of two.
minxfr  A pointer to an unsigned integer to be filled in with the minimum effective transfer size (see ddi_iomin(9F), ddi_dma_lim_sparc(9S) and ddi_dma_lim_x86(9S)). This also is guaranteed to be a power of two.

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

RETURN VALUES  ddi_dma_devalign() returns:

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

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

SEE ALSO  ddi_dma_setup(9F), ddi_iomin(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
ddi_dmae (9F)

NAME

ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable,
ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty,
ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

int ddi_dmae_alloc( dev_info_t *dip, int chnl, int (*callback) (caddr_t), caddr_t arg);
int ddi_dmae_release( dev_info_t *dip, int chnl);
int ddi_dmae_prog( dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
                   ddi_dma_cookie_t *cookiep, int chnl);
int ddi_dmae_disable( dev_info_t *dip, int chnl);
int ddi_dmae_enable( dev_info_t *dip, int chnl);
int ddi_dmae_stop( dev_info_t *dip, int chnl);
int ddi_dmae_getcnt( dev_info_t *dip, int chnl, int *countp);
int ddi_dmae_1stparty( dev_info_t *dip, int chnl);
int ddi_dmae_getlim( dev_info_t *dip, ddi_dma_lim_t *limitsp);
int ddi_dmae_getattr( dev_info_t *dip, ddi_dma_attr_t *attrp);

INTERFACE LEVEL
ARGUMENTS

Solaris DDI specific (Solaris DDI).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip</td>
<td>A dev_info pointer, which identifies the device.</td>
</tr>
<tr>
<td>chnl</td>
<td>A DMA channel number, or an MCA bus arbitration level. On ISA or EISA buses this number must be 0, 1, 2, 3, 5, 6, or 7. On MCA buses this number must be in the range 0 to 14.</td>
</tr>
<tr>
<td>callback</td>
<td>The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_SLEEP Wait until resources are available.</td>
</tr>
<tr>
<td></td>
<td>DDI_DMA_DONTWAIT Do not wait until resources are available and do not schedule a callback.</td>
</tr>
<tr>
<td>arg</td>
<td>Argument to be passed to the callback function, if specified.</td>
</tr>
<tr>
<td>dmaereqp</td>
<td>A pointer to a DMA engine request structure. See ddi_dmae_req(9S).</td>
</tr>
<tr>
<td>cookiep</td>
<td>A pointer to a ddi_dma_cookie(9S) object, obtained from ddi_dma_segtocookie(9F), which contains the address and count.</td>
</tr>
<tr>
<td>countp</td>
<td>A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.</td>
</tr>
<tr>
<td>limitsp</td>
<td>A pointer to a DMA limit structure. See ddi_dma_lim_x86(9S).</td>
</tr>
<tr>
<td>attrp</td>
<td>A pointer to a DMA attribute structure. See ddi_dma_attr(9S).</td>
</tr>
</tbody>
</table>

DESCRIPTION

There are three possible ways that a device can perform DMA engine functions:

"Bus master DMA" If the device is capable of acting as a true bus master, then the driver should program the device’s DMA registers directly and not...
make use of the DMA engine functions described here. The driver should obtain the DMA address and count from `ddi_dma_segtocookie`(9F). See `ddi_dma_cookie`(9S) for a description of a DMA cookie.

"Third-party DMA" This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

"First-party DMA" Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

**ddi_dmae_alloc()**

The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See `detach`(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of `callback` determines what action will be taken. If the value of `callback` is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument `arg`) from interrupt context.

When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.

**ddi_dmae_prog()**

The `ddi_dmae prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for `dmaereqp` if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and
count are specified by passing `ddi_dmae_prog()` a cookie obtained from
`ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA
engine request structure passed in through `dmaereqp`. The fields of that structure are
documented in `ddi_dmae_req(9S)`. Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA
setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be
used to retrieve a cookie which contains the address and count. Then this cookie is
passed to `ddi_dmae_prog()`.

`ddi_dmae_disable()` The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds
to a device’s DMA service requests.

`ddi_dmae_enable()` The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be
used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is
automatically enabled after successful programming by `ddi_dmae_prog()`.

`ddi_dmae_stop()` The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

`ddi_dmae_getcnt()` The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and
sets *countp to the number of bytes remaining to be transferred. The channel is assumed
to be stopped.

`ddi_dmae_1stparty()` In the case of ISA and EISA buses, `ddi_dmae_1stparty()` configures a channel in the
system’s DMA engine to operate in a "slave" ("cascade") mode.
In the case of the MCA bus, a call to `ddi_dmae_1stparty()` should still be made, regardless
of whether the channel number specifies one of the DMA arbitration levels or a non-DMA
arbitration level.
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)`. The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their
own bus mastering or use first-party DMA must create and initialize their own DMA limit
structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be
passed to the DMA setup routines so that they will know how to break the DMA request
into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If
the device has any particular restrictions on transfer size or granularity (such as the size
of disk sector), the driver should further restrict the values in the structure members
before passing them to the DMA setup routines. The driver must not relax any of the res-
trictions 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.
**ddi_dmae_getattr**

The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by `attrp`, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See `ddi_dma_nextcookie(9F)` and `ddi_dma_getwin(9F).

**RETURN VALUES**

- **DDI_SUCCESS** Upon success, for all of these routines.
- **DDI_FAILURE** May be returned due to invalid arguments.
- **DDI_DMA_NORESOURCES** May be returned by `ddi_dmae_alloc()` if the requested resources are not available and the value of `dmae_waitfp` is not `DDI_DMA_SLEEP`.

**CONTEXT**

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`eisa(4), isa(4), mca(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)`

modified 1 Jan 1997 SunOS 5.6 9F-251
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  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
- `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.

SEE ALSO  
`ddi_dma_addr_setup(9F)`, `ddi_dma_buf_setup(9F)`, `ddi_dma_htoc(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_req(9S)`

Writing Device Drivers
NAME  

ddi_dma_free_handle – free DMA handle

SYNOPSIS

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

void ddi_dma_free_handle(ddi_dma_handle_t *handle);

ARGUMENTS

handle  

A pointer to the DMA handle previously allocated by a call to
ddi_dma_alloc_handle(9F).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_free_handle( ) destroys the DMA handle pointed to by handle. Any further
references to the DMA handle will have undefined results. Note that
ddi_dma_unbind_handle(9F) must be called prior to ddi_dma_free_handle( ) to free
any resources the system may be caching on the handle.

CONTEXT

ddi_dma_free_handle( ) can be called from user, kernel, or interrupt context.

SEE ALSO

ddi_dma_alloc_handle(9F), ddi_dma_unbind_handle(9F)

Writing Device Drivers

modified 26 Sep 1994  
SunOS 5.6  
9F-253
ddi_dma_getwin (9F)

Kernel Functions for Drivers

NAME    ddi_dma_getwin – activate a new DMA window

SYNOPSIS include <sys/ddi.h>
            #include <sys/sunddi.h>
            int ddi_dma_getwin(ddi_dma_handle_t handle, uint_t win,
                                off_t *offp, size_t *lenp, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F).
win Number of the window to activate.
offp Pointer to an offset. Upon a successful return, offp will contain the new offset indicating the beginning of the window within the object.
lenp Upon a successful return, lenp will contain the size, in bytes, of the current window.
cookiep A pointer to the first ddi_dma_cookie(9S) structure.
ccountp Upon a successful return, ccountp will contain the number of cookies for this DMA window.

DESCRIPTION ddi_dma_getwin() activates a new DMA window. If a DMA resource allocation request returns DDI_DMA_PARTIAL_MAP indicating that resources for less than the entire object were allocated, the current DMA window can be changed by a call to ddi_dma_getwin(). The caller must first determine the number of DMA windows, N, using ddi_dma_numwin(9F). ddi_dma_getwin() takes a DMA window number from the range [0..N-1] as the parameter win and makes it the current DMA window.

ddi_dma_getwin() fills in the first DMA cookie pointed to by cookiep with the appropriate address, length, and bus type. *ccountp is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved using ddi_dma_nextcookie(9F).

ddi_dma_getwin() takes care of underlying resource synchronizations required to shift the window. However accessing the data prior to or after moving the window requires further synchronization steps using ddi_dma_sync(9F).

ddi_dma_getwin() is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, the interrupt routine returns without invoking another DMA transfer. Otherwise, it calls ddi_dma_getwin() to shift the current window and start another DMA transfer.

9F-254 SunOS 5.6 modified 15 Nov 1996
Kernel Functions for Drivers

**ddi_dma_getwin()**

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDI_SUCCESS</strong></td>
<td>Resources for the specified DMA window are allocated.</td>
</tr>
<tr>
<td><strong>DDI_FAILURE</strong></td>
<td><code>win</code> is not a valid window index.</td>
</tr>
</tbody>
</table>

**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*

---

Modified 15 Nov 1996

SunOS 5.6

9F-255
**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**
Solaris SPARC DDI specific (Solaris SPARC DDI).

**ARGUMENTS**
- **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_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.

**SEE ALSO**
`ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S)`

*Writing Device Drivers*
Kernel Functions for Drivers

NAME  

ddi_dma_mem_alloc – allocate memory for DMA transfer

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

int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
    ddi_device_acc_attr_t *accattrp, uint_t flags,
    int (*waitfp)(caddr_t), caddr_t arg, caddr_t *kaddrp,
    size_t *real_length, ddi_acc_handle_t *handlep);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle  
The DMA handle previously allocated by a call to
ddi_dma_alloc_handle(9F).

length  
The length in bytes of the desired allocation.

accattrp  
Pointer to a device access attribute structure of this device (see
ddi_device_acc_attr(9S)).

flags  
Data transfer mode flags. Possible values are:

DDI_DMA_STREAMING  
Sequential, unidirectional, block-sized, and block-aligned transfers.

DDI_DMA_CONSISTENT  
Nonsequential transfers of small objects.

waitfp  
The address of a function to call back later if resources are not available
now. The special function addresses DDI_DMA_SLEEP and
DDI_DMA_DONTWAIT are taken to mean, respectively, wait until
resources are available or, do not wait and do not schedule a callback.

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

kaddrp  
On successful return, *kaddrp points to the allocated memory.

*real_length  
The amount of memory, in bytes, allocated. Alignment and padding
requirements may require ddi_dma_mem_alloc() to allocate more
memory than requested in length.

handlep  
Pointer to a data access handle.

DESCRIPTION

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

flags should be set to DDI_DMA_STREAMING if the device is doing sequential, unidirectional,
block-sized, and block-aligned transfers to or from memory. The alignment and
padding constraints specified by the minxfer and burstsizes fields in the DMA attribute
structure, ddi_dma_attr(9S) (see ddi_dma_alloc_handle(9F)) will be used to allocate the
most effective hardware support for large transfers. For example, if an I/O transfer can be

modified 15 Nov 1996  SunOS 5.6  9F-257
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.

RETURN VALUES ddi_dma_mem_alloc() returns:

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

CONTEXT ddi_dma_mem_alloc() can be called from user or interrupt context, except when waitfp is set to DDI_DMA_SLEEP, in which case it can be called from user context only.

SEE ALSO ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F), ddi_dma_mem_free(9F), ddi_dma_sync(9F), ddi_dma_unbind_handle(9F), ddi_get8(9F), ddi_put8(9F), ddi_device_acc_attr(9S), ddi_dma_attr(9S)

Writing Device Drivers
NAME  
ddi_dma_mem_free – free previously allocated memory

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>
void ddi_dma_mem_free(ddi_acc_handle_t *handlep);

ARGUMENTS  
handlep  
Pointer to the data access handle previously allocated by a call to ddi_dma_mem_alloc(9F).

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

DESCRIPTION  
ddi_dma_mem_free() deallocates the memory acquired by ddi_dma_mem_alloc(9F). In addition, it destroys the data access handle handlep associated with the memory.

CONTEXT  
ddi_dma_mem_free() can be called from user, kernel, or interrupt context.

SEE ALSO  
ddi_dma_mem_alloc(9F)
Writing Device Drivers
### NAME
ddi_dma_movwin – shift current DMA window

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

int ddi_dma_movwin(ddi_dma_handle_t handle, off_t *offp, u_int *lenp,
                    ddi_dma_cookie_t *cookiep);
```

### INTERFACE LEVEL
Solaris SPARC DDI specific (Solaris SPARC DDI).

### ARGUMENTS
- **handle**: The DMA handle filled in by a call to `ddi_dma_setup(9F)`.
- **offp**: A pointer to an offset to set the DMA window to. Upon a successful return, it will be filled in with the new offset from the beginning of the object resources are allocated for.
- **lenp**: A pointer to a value which must either be the current size of the DMA window (as known from a call to `ddi_dma_curwin(9F)` or from a previous call to `ddi_dma_movwin()`). Upon a successful return, it will be filled in with the size, in bytes, of the current window.
- **cookiep**: A pointer to a DMA cookie (see `ddi_dma_cookie(9S)`). Upon a successful return, `cookiep` is filled in just as if an implicit `ddi_dma_htoc(9F)` had been made.

### DESCRIPTION
`ddi_dma_movwin()` shifts the current DMA window. If a DMA request allows the system to allocate resources for less than the entire object by setting the DDI_DMA_PARTIAL flag in the `ddi_dma_req(9S)` structure, the current DMA window can be shifted by a call to `ddi_dma_movwin()`.

The caller must first determine the current DMA window size by a call to `ddi_dma_curwin(9F)`. Using the current offset and size of the window thus retrieved, the caller of `ddi_dma_movwin()` may change the window onto the object by changing the offset by a value which is some multiple of the size of the DMA window.

`ddi_dma_movwin()` takes care of underlying resource synchronizations required to shift the window. However if you want to access the data prior or after moving the window, further synchronizations using `ddi_dma_sync(9F)` are required.

This function is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, it returns without invoking another DMA transfer. Otherwise it calls `ddi_dma_movwin()` to shift the current window and starts another DMA transfer.

### RETURN VALUES
`ddi_dma_movwin()` returns:
- **DDI_SUCCESS**: The current length and offset are legal and have been set.
- **DDI_FAILURE**: Otherwise.

---

9F-260 SunOS 5.6 modified 13 Sep 1992
ddi_dma_movwin() can be called from user or interrupt context.

SEE ALSO ddi_dma_curwin(9F), ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S), ddi_dma_req(9S)

Writing Device Drivers

WARNINGS The caller must guarantee that the resources used by the object are inactive prior to calling this function.
NAME  
ddi_dma_nextcookie – retrieve subsequent DMA cookie

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>
void ddi_dma_nextcookie(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep);

ARGUMENTS  
handle       The handle previously allocated by a call to ddi_dma_alloc_handle(9F).
cookiep      A pointer to a ddi_dma_cookie(9S) structure.

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

DESCRIPTION  
ddi_dma_nextcookie() retrieves subsequent DMA cookies for a DMA object.
ddi_dma_nextcookie() fills in the ddi_dma_cookie(9S) structure pointed to by cookiep. The ddi_dma_cookie(9S) structure must be allocated prior to calling ddi_dma_nextcookie().

The DMA cookie count returned by ddi_dma_buf_bind_handle(9F), ddi_dma_addr_bind_handle(9F), or ddi_dma_getwin(9F) indicates the number of DMA cookies a DMA object consists of. If the resulting cookie count, N, is larger than 1, ddi_dma_nextcookie() must be called N-1 times to retrieve all DMA cookies.

CONTEXT  
ddi_dma_nextcookie() can be called from user, kernel, or interrupt context.

EXAMPLES  
This example demonstrates the use of ddi_dma_nextcookie() to process a scatter-gather list of I/O requests.

/* setup scatter-gather list with multiple DMA cookies */

ddi_dma_cookie_t dmacookie;
uint_t ccount;
...

status = ddi_dma_buf_bind_handle(handle, bp, DDI_DMA_READ, NULL, NULL, &dmacookie, &ccount);

if (status == DDI_DMA_MAPPED) {

    /* program DMA engine with first cookie */

    while (--ccount > 0) {
        ddi_dma_nextcookie(handle, &dmacookie);  /* program DMA engine with next cookie */
    }
}

...
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*

modified 26 Sep 1994  SunOS 5.6  9F-263
NAME

ddi_dma_nextseg – get next DMA segment

SYNOPSIS

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

int ddi_dma_nextseg( ddi_dma_win_t win, ddi_dma_seg_t seg, ddi_dma_seg_t *nseg);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

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

DESCRIPTION

ddi_dma_nextseg() gets the next DMA segment within the specified window win. If the current segment is NULL, the first DMA segment within the window is returned.

A DMA segment is always required for a DMA window. A DMA segment is a contiguous portion of a DMA window (see ddi_dma_nextwin(9F)) which is entirely addressable by the device for a data transfer operation.

An example where multiple DMA segments are allocated is where the system does not contain DVMA capabilities and the object may be non-contiguous. In this example the object will be broken into smaller contiguous DMA segments. Another example is where the device has an upper limit on its transfer size (for example an 8-bit address register) and has expressed this in the DMA limit structure (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). In this example the object will be broken into smaller addressable DMA segments.

RETURN VALUES

ddi_dma_nextseg() returns:

DDI_SUCCESS Successfully filled in the next segment pointer.
DDI_DMA_DONE There is no next segment. The current segment is the final segment within the specified window.
DDI_DMA_STALE win does not refer to the currently active window.

CONTEXT

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

EXAMPLES

For an example see ddi_dma_segtocookie(9F).

SEE ALSO

ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_nextwin(9F), ddi_dma_segtocookie(9F), ddi_dma_sync(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
NAME

ddi_dma_nextwin – get next DMA window

SYNOPSIS

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

int ddi_dma_nextwin( ddi_dma_handle_t handle, ddi_dma_win_t win,
    ddi_dma_win_t *nwin);
```

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

- `handle`: A DMA handle.
- `win`: The current DMA window or NULL.
- `nwin`: A pointer to the next DMA window to be filled in. If `win` is NULL, a pointer to the first window within the object is returned.

DESCRIPTION

`ddi_dma_nextwin()` shifts the current DMA window `win` within the object referred to by `handle` to the next DMA window `nwin`. If the current window is NULL, the first window within the object is returned. A DMA window is a portion of a DMA object or might be the entire object. A DMA window has system resources allocated to it and is prepared to accept data transfers. Examples of system resources are DVMA mapping resources and intermediate transfer buffer resources.

All DMA objects require a window. If the DMA window represents the whole DMA object it has system resources allocated for the entire data transfer. However, if the system is unable to setup the entire DMA object due to system resource limitations, the driver writer may allow the system to allocate system resources for less than the entire DMA object. This can be accomplished by specifying the DDI_DMA_PARTIAL flag as a parameter to `ddi_dma_buf_setup(9F)` or `ddi_dma_addr_setup(9F)` or as part of a `ddi_dma_req(9S)` structure in a call to `ddi_dma_setup(9F)`.

Only the window that has resources allocated is valid per object at any one time. The currently valid window is the one that was most recently returned from `ddi_dma_nextwin()`. Furthermore, because a call to `ddi_dma_nextwin()` will reallocate system resources to the new window, the previous window will become invalid. Note: It is a severe error to call `ddi_dma_nextwin()` before any transfers into the current window are complete.

`ddi_dma_nextwin()` takes care of underlying memory synchronizations required to shift the window. However, if you want to access the data before or after moving the window, further synchronizations using `ddi_dma_sync(9F)` are required.

RETURN VALUES

- `DDI_SUCCESS`: Successfully filled in the next window pointer.
- `DDI_DMA_DONE`: There is no next window. The current window is the final window within the specified object.
- `DDI_DMA_STALE`: `win` does not refer to the currently active window.

modified 12 Oct 1992

SunOS 5.6

9F-265
| CONTEXT | ddi_dma_nextwin() can be called from user or interrupt context. |
| EXAMPLES | For an example see ddi_dma_segtocookie(9F). |
| SEE ALSO | ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_nextseg(9F), ddi_dma_segtocookie(9F), ddi_dma_sync(9F), ddi_dma_req(9S) |

Writing Device Drivers
NAME
ddi_dma_numwin – retrieve number of DMA windows

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
int ddi_dma_numwin(ddi_dma_handle_t handle, uint_t *nwinp);

ARGUMENTS
handle The DMA handle previously allocated by a call to
ddi_dma_alloc_handle(9F).
nwinp Upon a successful return, nwinp will contain the number of DMA win-
dows for this object.

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION
ddi_dma_numwin() returns the number of DMA windows for a DMA object if partial
resource allocation was permitted.

RETURN VALUES
ddi_dma_numwin() returns:
DDI_SUCCESS Successfully filled in the number of DMA windows.
DDI_FAILURE DMA windows are not activated.

CONTEXT
ddi_dma_numwin() can be called from user, kernel, or interrupt context.

SEE ALSO
ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F),
ddi_dma_buf_bind_handle(9F), ddi_dma_unbind_handle(9F)

Writing Device Drivers
NAME  ddi_dma_segtocookie – convert a DMA segment to a DMA address cookie

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

int ddi_dma_segtocookie( ddi_dma_seg_t seg, off_t *offp, off_t *lenp,
                        ddi_dma_cookie_t *cookiep);

ARGUMENTS  
seg  A DMA segment.
offp  A pointer to an off_t. Upon a successful return, it is filled in with the
      offset. This segment is addressing within the object.
lenp  The byte length. This segment is addressing within the object.
cookiep  A pointer to a DMA cookie (see ddi_dma_cookie(9S)).

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

DESCRIPTION  
 ddi_dma_segtocookie() takes a DMA segment and fills in the cookie pointed to by cookiep
              with the appropriate address, length, and bus type to be used to program the DMA
              engine.  ddi_dma_segtocookie() also fills in *offp and *lenp, which specify the range
              within the object.

RETURN VALUES  
 ddi_dma_segtocookie() returns:
DDI_SUCCESS  Successfully filled in all values.
DDI_FAILURE  Failed to successfully fill in all values.

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

EXAMPLE  
for (win = NULL; (retw = ddi_dma_nextwin(handle, win, &nwin)) !=
       DDI_DMA_DONE; win = nwin) {
    if (retw != DDI_SUCCESS) {
        /* do error handling */
    } else {
        for (seg = NULL; (rets = ddi_dma_nextseg(nwin, seg, &nseg)) !=
             DDI_DMA_DONE; seg = nseg) {
            if (rets != DDI_SUCCESS) {

                /* do error handling */
            } else {
                ddi_dma_segtocookie(nseg, &off, &len, &cookie);

                /* program DMA engine */
            }
        }
    }
}
Kernel Functions for Drivers

SEE ALSO  

| Writing Device Drivers |

| ddi_dma_nextseg(9F), ddi_dma_nextwin(9F), ddi_dma_sync(9F), ddi_dma_cookie(9S) |

modified 12 Oct 1992

SunOS 5.6

9F-269
NAME  

ddi_dma_set_sbus64 – allow 64 bit transfers on SBus

SYNOPSIS  

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

int ddi_dma_set_sbus64(ddi_dma_handle_t handle, uint_t burstsizes);

INTERFACE LEVEL  

Solaris DDI specific (Solaris DDI).

ARGUMENTS  

handle  

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

burstsizes  

The possible burst sizes the device’s DMA engine can accept in 64 bit mode.

DESCRIPTION  

ddi_dma_set_sbus64( ) informs the system that the device wishes to perform 64 bit data transfers on the SBus. The driver must first allocate a DMA handle using ddi_dma_alloc_handle(9F) with a ddi_dma_attr(9S) structure describing the DMA attributes for a 32 bit transfer mode.

burstsizes describes the possible burst sizes the device’s DMA engine can accept in 64 bit mode. It may be distinct from the burst sizes for 32 bit mode set in the ddi_dma_attr(9S) structure. The system will activate 64 bit SBus transfers if the SBus supports them. Otherwise, the SBus will operate in 32 bit mode.

After DMA resources have been allocated (see ddi_dma_addr_bind_handle(9F) or ddi_dma_buf_bind_handle(9F)), the driver should retrieve the available burst sizes by calling ddi_dma_burstsizes(9F). This function will return the burst sizes in 64 bit mode if the system was able to activate 64 bit transfers. Otherwise burst sizes will be returned in 32 bit mode.

RETURN VALUES  

ddi_dma_set_sbus64( ) returns:

DDI_SUCCESS  

Successfully set the SBus to 64 bit mode.

DDI_FAILURE  

64 bit mode could not be set.

CONTEXT  

ddi_dma_set_sbus64( ) can be called from user, kernel, or interrupt context.

NOTES  

64 bit SBus mode is activated on a per SBus slot basis. If there are multiple SBus cards in one slot, they all must operate in 64 bit mode or they all must operate in 32 bit mode.

ATTRIBUTES  

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>SBus</td>
</tr>
</tbody>
</table>

SEE ALSO  

attributes(5), ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F), ddi_dma_buf_bind_handle(9F), ddi_dma_burstsizes(9F), ddi_dma_attr(9S)
NAME ddi_dma_setup – setup DMA resources

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

int ddi_dma_setup(dev_info_t *dip, ddi_dma_req_t *dmareqp,
                   ddi_dma_handle_t *handlep);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
  dip  A pointer to the device’s dev_info structure.
  dmareqp  A pointer to a DMA request structure (see ddi_dma_req(9S)).
  handlep  A pointer to a DMA handle to be filled in. See below for a discussion of a
           handle. If handlep is NULL, the call to ddi_dma_setup() is considered an
           advisory call, in which case no resources are allocated, but a value indi-
           cating the legality and the feasibility of the request is returned.

DESCRIPTION ddi_dma_setup() allocates resources for a memory object such that a device can perform
DMA to or from that object.
A call to ddi_dma_setup() informs the system that device referred to by dip wishes to
perform DMA to or from a memory object. The memory object, the device’s DMA capabil-
ities, the device driver’s policy on whether to wait for resources, are all specified in the
ddi_dma_req structure pointed to by dmareqp.
A successful call to ddi_dma_setup() fills in the value pointed to by handlep. This is an
opaque object called a DMA handle. This handle is then used in subsequent DMA calls,
until ddi_dma_free(9F) is called.
Again a DMA handle is opaque—drivers may not attempt to interpret its value. When a
driver wants to enable its DMA engine, it must retrieve the appropriate address to supply
to its DMA engine using a call to ddi_dma_htoc(9F), which takes a pointer to a DMA han-
dle and returns the appropriate DMA address.
When DMA transfer completes, the driver should free up the the allocated DMA resources
by calling ddi_dma_free().

RETURN VALUES ddi_dma_setup() returns:

  DDI_DMA_MAPPED  Successfully allocated resources for the object.
                 In the case of an advisory call, this indicates that
                 the request is legal.

  DDI_DMA_PARTIAL_MAP  Successfully allocated resources for a part of
                     the object. This is acceptable when partial
                     transfers are allowed using a flag setting in the
                     ddi_dma_req structure (see ddi_dma_req(9S)
                     and ddi_dma_movwin(9F)).

  DDI_DMA_NORESOURCES  When no resources are available.

modified 7 Jun 1993  SunOS 5.6  9F-271
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.

SEE ALSO: `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.
Kernel Functions for Drivers

NAME

ddi_dma_sync – synchronize CPU and I/O views of memory

SYNOPSIS

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

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

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle

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

offset

The offset into the object described by the handle.

length

The length, in bytes, of the area to synchronize. When length is zero, the entire range starting from offset to the end of the object has the requested operation applied to it.

type

Indicates the caller’s desire about what view of the memory object to synchronize. The possible values are DDI_DMA_SYNC_FORDEV, DDI_DMA_SYNC_FORCPU and DDI_DMA_SYNC_FORKERNEL.

DESCRIPTION

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.

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 didn’t do the modifying, a call to ddi_dma_sync( ) is required. This is true independent of any attributes of the memory object including, but not limited to, whether or not the memory was allocated for consistent mode I/O (see ddi_dma_mem_alloc(9F)) or whether or not DMA resources have been allocated for consistent mode I/O (see ddi_dma_addr_bind_handle(9F) or ddi_dma_buf_bind_handle(9F)).

This cannot be stated too strongly. If a consistent view of the memory object must be ensured between the time DMA resources are allocated for the object and the time they are deallocated, you must call ddi_dma_sync( ) to ensure that either a CPU or a DMA device has such a consistent view.

What to set type to depends on the view you are trying to ensure consistency for. If the memory object is modified by a CPU, and the object is going to be read by the DMA engine of the device, use DDI_DMA_SYNC_FORDEV. This ensures that the device’s DMA engine sees any changes that a CPU has made to the memory object. If the DMA engine for the device has written to the memory object, and you are going to read (with a CPU) the
object (using an extant virtual address mapping that you have to the memory object), use
**DDI_DMA_SYNC_FORCPU**. This ensures that a CPU’s view of the memory object
includes any changes made to the object by the device’s DMA engine. If you are only
interested in the kernel’s view (kernel-space part of the CPU’s view) you may use
**DDI_DMA_SYNC_FORKERNEL**. This gives a hint to the system—that is, if it is more
economical to synchronize the kernel’s view only, then do so; otherwise, synchronize for
CPU.

**RETURN VALUES**  
**ddi_dma_sync()** returns:
- **DDI_SUCCESS**  Caches are successfully flushed.
- **DDI_FAILURE**  The address range to be flushed is out of the address range esta-
blished by **ddi_dma_addr_bind_handle(9F)** or
  **ddi_dma_buf_bind_handle(9F)**.

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

**SEE ALSO**  
**ddi_dma_addr_bind_handle(9F)**, **ddi_dma_alloc_handle(9F)**,
**ddi_dma_buf_bind_handle(9F)**, **ddi_dma_mem_alloc(9F)**,
**ddi_dma_unbind_handle(9F)**

*Writing Device Drivers*
**NAME**

ddi_dma_unbind_handle – unbinds the address in a DMA handle

**SYNOPSIS**

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>
int ddi_dma_unbind_handle(ddi_dma_handle_t handle);
```

**ARGUMENTS**

| handle | The DMA handle previously allocated by a call to ddi_dma_alloc_handle(9F). |

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**DESCRIPTION**

ddi_dma_unbind_handle() frees all DMA resources associated with an existing DMA handle. When a DMA transfer completes, the driver should call ddi_dma_unbind_handle() to free system DMA resources established by a call to ddi_dma_buf_bind_handle(9F) or ddi_dma_addr_bind_handle(9F).

ddi_dma_unbind_handle() does an implicit ddi_dma_sync(9F) making further synchronization steps unnecessary.

**RETURN VALUES**

- **DDI_SUCCESS** on success
- **DDI_FAILURE** on failure

**CONTEXT**

ddi_dma_unbind_handle() can be called from user, kernel, or interrupt context.

**SEE ALSO**

ddi_dma_addr_bind_handle(9F), ddi_dma_alloc_handle(9F), ddi_dma_buf_bind_handle(9F), ddi_dma_free_handle(9F), ddi_dma_sync(9F)

*Writing Device Drivers*
NAME  
ddi_enter_critical, ddi_exit_critical – enter and exit a critical region of control

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

unsigned int ddi_enter_critical(void);
void ddi_exit_critical(unsigned int ddic);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
ddic  The returned value from the call to ddi_enter_critical() must be passed to ddi_exit_critical().

DESCRIPTION  
Nearly all driver operations can be done without any special synchronization and protection mechanisms beyond those provided by, e.g., mutexes (see mutex(9F)). However, for certain devices there can exist a very short critical region of code which must be allowed to run unimpeded. The function ddi_enter_critical() provides a mechanism by which a driver can ask the system to guarantee to the best of its ability that the current thread of execution will neither be preempted nor interrupted. This stays in effect until a bracketing call to ddi_exit_critical() is made (with an argument which was the returned value from ddi_enter_critical()).

The driver may not call any functions external to itself in between the time it calls ddi_enter_critical() and the time it calls ddi_exit_critical().

RETURN VALUES  
ddi_enter_critical() returns an opaque unsigned integer which must be used in the subsequent call to ddi_exit_critical().

CONTEXT  
This function can be called from user or interrupt context.

WARNINGS  
Driver writers should note that in a multiple processor system this function does not temporarily suspend other processors from executing. This function also cannot guarantee to actually block the hardware from doing such things as interrupt acknowledge cycles. What it can do is guarantee that the currently executing thread will not be preempted.

Do not write code bracketed by ddi_enter_critical() and ddi_exit_critical() that can get caught in an infinite loop, as the machine may crash if you do.

SEE ALSO  
mutex(9F)
Writing Device Drivers
**NAME**

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

**SYNOPSIS**

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

int ddi_ffs(long mask);
int ddi_flsl(long mask);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

`mask` A 32-bit argument value to search through.

**DESCRIPTION**

The function `ddi_ffs()` takes its argument and returns the shift count that the first (least significant) bit set in the argument corresponds to. The function `ddi_flsl()` does the same, only it returns the shift count for the last (most significant) bit set in the argument.

**RETURN VALUES**

0 No bits are set in mask.

`N` Bit `N` is the least significant (`ddi_ffs`) or most significant (`ddi_flsl`) bit set in mask. Bits are numbered from 1 to 32, with bit 1 being the least significant bit position and bit 32 the most significant position.

**CONTEXT**

This function can be called from user or interrupt context.

**SEE ALSO**

*Writing Device Drivers*

---

modified 20 Dec 1995

SunOS 5.6

9F-277
NAME ddi_get8, ddi_get16, ddi_get32, ddi_get64, ddi_getb, ddi_getw, ddi_getl, ddi_getll – read data from the mapped memory address, device register or allocated DMA memory address

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

uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).

dev_addr Base device address.

DESCRIPTION The ddi_get8(), ddi_get16(), ddi_get32(), and ddi_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr.

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

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

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

SEE ALSO ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F)

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb</td>
<td>ddi_get8</td>
</tr>
<tr>
<td>ddi_getw</td>
<td>ddi_get16</td>
</tr>
<tr>
<td>ddi_getl</td>
<td>ddi_get32</td>
</tr>
<tr>
<td>ddi_getll</td>
<td>ddi_get64</td>
</tr>
<tr>
<td>NAME</td>
<td>ddi_get_cred – returns a pointer to the credential structure of the caller</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| SYNOPSIS        | #include <sys/types.h>  
                    #include <sys/ddi.h>  
                    #include <sys/sunddi.h>  
                    cred_t *ddi_get_cred(); |
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI).                                    |
| DESCRIPTION     | ddi_get_cred() returns a pointer to the user credential structure of the caller. |
| RETURN VALUES   | ddi_get_cred() returns a pointer to the caller’s credential structure.   |
| CONTEXT         | ddi_get_cred() can be called from user context only.                    |
| SEE ALSO        | Writing Device Drivers                                                  |
NAME

ddi_get_driver_private, ddi_set_driver_private – get or set the address of the device’s private data area

SYNOPSIS

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

void ddi_set_driver_private(dev_info_t *dip, caddr_t data);
caddr_t ddi_get_driver_private(dev_info_t *dip);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_get_driver_private()</td>
<td>dip</td>
<td>Pointer to device information structure to get from.</td>
</tr>
<tr>
<td>ddi_set_driver_private()</td>
<td>dip</td>
<td>Pointer to device information structure to set.</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>Data area address to set.</td>
</tr>
</tbody>
</table>

DESCRIPTION

`ddi_get_driver_private()` returns the address of the device’s private data area from the device information structure pointed to by `dip`.

`ddi_set_driver_private()` sets the address of the device’s private data area in the device information structure pointed to by `dip` with the value of `data`.

RETURN VALUES

`ddi_get_driver_private()` returns the contents of `devi_driver_data`. If `ddi_set_driver_private()` has not been previously called with `dip`, an unpredictable value is returned.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

`Writing Device Drivers`
NAME  

ddi_get_instance – get device instance number

SYNOPSIS

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

int ddi_get_instance(dev_info_t *dip);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip  Pointer to dev_info structure.

DESCRIPTION

ddi_get_instance() returns the instance number of the device corresponding to dip. The system assigns an instance number to every device. Instance numbers for devices attached to the same driver are unique. This provides a way for the system and the driver to uniquely identify one or more devices of the same type. The instance number is derived by the system from different properties for different device types in an implementation specific manner.

Once an instance number has been assigned to a device, it will remain the same even across reconfigurations and reboots. Therefore, instance numbers seen by a driver may not appear to be in consecutive order. For example, if device foo0 has been assigned an instance number of 0 and device foo1 has been assigned an instance number of 1, if foo0 is removed, foo1 will continue to be associated with instance number 1 (even though foo1 is now the only device of its type on the system).

RETURN VALUES

ddi_get_instance() returns the instance number of the device corresponding to dip.

CONTEXT

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

SEE ALSO

path_to_inst(4)

Writing Device Drivers

modified 20 Jul 1994  SunOS 5.6  9F-281
ddi_get_parent (9F)  

NAME  ddi_get_parent – find the parent of a device information structure

SYNOPSIS  

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

dev_info_t *ddi_get_parent(dev_info_t *dip);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  

dip  Pointer to a device information structure.

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

RETURN VALUES  
ddi_get_parent() returns a pointer to a device information structure.

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

SEE ALSO  Writing Device Drivers
NAME

ddi_intr_hilevel – indicate interrupt handler type

SYNOPSIS

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

int ddi_intr_hilevel(dev_info_t *dip, u_int inumber);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to dev_info structure.
inumber Interrupt number.

DESCRIPTION

ddi_intr_hilevel() returns non-zero if the specified interrupt is a "high level" interrupt.

High level interrupts must be handled without using system services that manipulate
thread or process states, because these interrupts are not blocked by the scheduler.

In addition, high level interrupt handlers must take care to do a minimum of work
because they are not preemptable.

A typical high level interrupt handler would put data into a circular buffer and schedule
a soft interrupt by calling ddi_trigger_softintr(). The circular buffer could be protected
by using a mutex that was properly initialized for the interrupt handler.

ddi_intr_hilevel() can be used before calling ddi_add_intr() to decide which type of
interrupt handler should be used. Most device drivers are designed with the knowledge
that the devices they support will always generate low level interrupts, however some
devices, for example those using SBus or VME bus level 6 or 7 interrupts must use this
test because on some machines those interrupts are high level (above the scheduler level)
and on other machines they are not.

RETURN VALUES

non-zero indicates a high-level interrupt.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

ddi_add_intr(9F), mutex(9F)

Writing Device Drivers

modified 7 Jan 1992
SunOS 5.6
ddi_io_get8 (9F) Kernel Functions for Drivers

NAME

ddi_io_get8, ddi_io_get16, ddi_io_get32, ddi_io_getb, ddi_io_getw, ddi_io_getl – read data from the mapped device register in I/O space

SYNOPSIS

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

uint8_t ddi_io_get8(ddi_acc_handle_t handle, int dev_port);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, int dev_port);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, int dev_port);

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_port The device port.

DESCRIPTION

These routines generate a read of various sizes from the device port, dev_port, 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 port, dev_port.

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

CONTEXT

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

SEE ALSO

isa(4), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_io_rep_put8(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as x86 where accesses to the memory and I/O space are different.

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getb</td>
<td>ddi_io_get8</td>
</tr>
<tr>
<td>ddi_io_getw</td>
<td>ddi_io_get16</td>
</tr>
<tr>
<td>ddi_io_getl</td>
<td>ddi_io_get32</td>
</tr>
</tbody>
</table>

9F-284 SunOS 5.6 modified 30 Sep 1996
NAME    ddi_iomin – find minimum alignment and transfer size for DMA

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

int ddi_iomin(dev_info_t *dip, int initial, int streaming);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip            A pointer to the device’s dev_info structure.
initial        The initial minimum DMA transfer size in bytes. This may be zero or an appropriate dlim_minxfer value for device’s ddi_dma_lim structure (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). This value must be a power of two.
streaming      This argument, if non-zero, indicates that the returned value should be modified to account for streaming mode accesses (see ddi_dma_req(9S) for a discussion of streaming versus non-streaming access mode).

DESCRIPTION ddi_iomin(), finds out the minimum DMA transfer size for the device pointed to by dip. This provides a mechanism by which a driver can determine the effects of underlying caches as well as intervening bus adapters on the granularity of a DMA transfer.

RETURN VALUES ddi_iomin() returns the minimum DMA transfer size for the calling device, or it returns zero, which means that you cannot get there from here.

CONTEXT This function can be called from user or interrupt context.

SEE ALSO ddi_dma_devalign(9F), ddi_dma_setup(9F), ddi_dma_sync(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Writing Device Drivers
**NAME**
ddi_iopb_alloc, ddi_iopb_free – allocate and free non-sequentially accessed memory

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

int ddi_iopb_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, u_int length, 
caddr_t *iopbp);

void ddi_iopb_free(caddr_t iopb);
```

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

**ARGUMENTS**

- **ddi_iopb_alloc()**
  - *dip* A pointer to the device’s dev_info structure.
  - *limits* A pointer to a DMA limits structure for this device (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). If this pointer is NULL, a default set of DMA limits is assumed.
  - *length* The length in bytes of the desired allocation.
  - *iopbp* A pointer to a caddr_t. On a successful return, *iopbp* points to the allocated storage.

- **ddi_iopb_free()**
  - *iopb* The iopb returned from a successful call to ddi_iopb_alloc().

**DESCRIPTION**

ddi_iopb_alloc() allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using ddi_dma_sync(9F) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as consistent access. The allocation will obey the alignment and padding constraints as specified in the limits argument and other limits imposed by the system.

Note that you still must use DMA resource allocation functions (see ddi_dma_setup(9F)) to establish DMA resources for the memory allocated using ddi_iopb_alloc().

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using ddi_dma_sync(9F) or ddi_dma_free(9F) are still required. The DMA resources will be allocated so that these synchronization steps are as efficient as possible.

ddi_iopb_free() frees up memory allocated by ddi_iopb_alloc().

**RETURN VALUES**

- **ddi_iopb_alloc()** returns:
  - DDI_SUCCESS Memory successfully allocated.
  - DDI_FAILURE Allocation failed.

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

modified 17 May 1994

SunOS 5.6

9F-287
### NAME
.GetChild data to the mapped device register in I/O space

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

void ddi_io_put8(ddi_acc_handle_t handle, int dev_port, uint8_t value);
void ddi_io_put16(ddi_acc_handle_t handle, int dev_port, uint16_t value);
void ddi_io_put32(ddi_acc_handle_t handle, int dev_port, uint32_t value);
```

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

### ARGUMENTS
- **handle**: The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.
- **dev_port**: The device port.
- **value**: The data to be written to the device.

### DESCRIPTION
These routines generate a write of various sizes to the device port, `dev_port`, 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 port, `dev_port`.

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

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

### SEE ALSO
- isa(4), `ddi_io_get8(9F)`, `ddi_io_rep_get8(9F)`, `ddi_io_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

### NOTES
For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see isa(4)) but memory space only in PCI local bus. This is especially true in instruction set architectures such as x86 where accesses to the memory and I/O space are different.

The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_putb</td>
<td>ddi_io_put8</td>
</tr>
<tr>
<td>ddi_io_putw</td>
<td>ddi_io_put16</td>
</tr>
<tr>
<td>ddi_io_putl</td>
<td>ddi_io_put32</td>
</tr>
</tbody>
</table>

modified 31 Aug 1996

SunOS 5.6

9F-289
**NAME**

ddi_io_rep_get8, ddi_io_rep_get16, ddi_io_rep_get32, ddi_io_rep_getw, ddi_io_rep_getb, ddi_io_rep_getl – read multiple data from the mapped device register in I/O space

**SYNOPSIS**

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

void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, int dev_port, size_t repcount);
void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, int dev_port, size_t repcount);
void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, int dev_port, size_t repcount);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

- `handle` The data access handle returned from setup calls, such as `ddi_regs_map_setup(9F)`.  
- `host_addr` Base host address.  
- `dev_port` The device port.  
- `repcount` Number of data accesses to perform.

**DESCRIPTION**

These routines generate multiple reads from the device port, `dev_port`, in I/O space. `repcount` data is copied from the device port, `dev_port`, to the host address, `host_addr`. For each input datum, the `ddi_io_rep_get8()`, `ddi_io_rep_get16()`, and `ddi_io_rep_get32()` functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device port.  

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

**CONTEXT**

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

**SEE ALSO**

`isa(4)`, `ddi_io_get8(9F)`, `ddi_io_put8(9F)`, `ddi_io_rep_put8(9F)`, `ddi_regs_map_free(9F)`, `ddi_regs_map_setup(9F)`, `ddi_device_acc_attr(9S)`

**NOTES**

For drivers using these functions, it may not be easy to maintain a single source to support devices with multiple bus versions. For example, devices may offer I/O space in ISA bus (see `isa(4)`) but memory space only in PCI local bus. This is especially true in instruction set architectures such as x86 where accesses to the memory and I/O space are different.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_getb</td>
<td>ddi_io_rep_get8</td>
</tr>
<tr>
<td>ddi_io_rep_getw</td>
<td>ddi_io_rep_get16</td>
</tr>
<tr>
<td>ddi_io_rep_getl</td>
<td>ddi_io_rep_get32</td>
</tr>
</tbody>
</table>
ddi_io_rep_put8 (9F)  Kernel Functions for Drivers

NAME
  ddi_io_rep_put8, ddi_io_rep_put16, ddi_io_rep_put32, ddi_io_rep_putw,
  ddi_io_rep_putl, ddi_io_rep_putb – write multiple data to the mapped device register in
  I/O space

SYNOPSIS
  #include <sys/ddi.h>
  #include <sys/sunddi.h>
  void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, int dev_port,
                       size_t repcount);
  void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, int dev_port,
                       size_t repcount);
  void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, int dev_port,
                       size_t repcount);

INTERFACE
  LEVEL
  Solaris DDI specific (Solaris DDI).

ARGUMENTS
  handle       The data access handle returned from setup calls, such as
               ddi_regs_map_setup(9F).
  host_addr    Base host address.
  dev_port     The device port.
  repcount     Number of data accesses to perform.

DESCRIPTION
  These routines generate multiple writes to the device port, dev_port, in I/O space.
  repcount data is copied from the host address, host_addr, to the device port, dev_port. 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 port.
  host_addr must be aligned to the datum boundary described by the function.
  Each individual datum will automatically be translated to maintain a consistent view
  between the host and the device based on the encoded information in the data access
  handle. The translation may involve byte-swapping if the host and the device have
  incompatible endian characteristics.

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

SEE ALSO
  isa(4), ddi_io_get8(9F), ddi_io_put8(9F), ddi_io_rep_get8(9F), ddi_regs_map_setup(9F),
  ddi_device_acc_attr(9S)

NOTES
  For drivers using these functions, it may not be easy to maintain a single source to sup-
  port 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 instruc-
  tion set architectures such as x86 where accesses to the memory and I/O space are dif-
  ferent.
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_rep_putb</td>
<td>ddi_io_rep_put8</td>
</tr>
<tr>
<td>ddi_io_rep_putw</td>
<td>ddi_io_rep_put16</td>
</tr>
<tr>
<td>ddi_io_rep_putl</td>
<td>ddi_io_rep_put32</td>
</tr>
</tbody>
</table>
NAME  
ddi_mapdev – create driver-controlled mapping of device

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

int ddi_mapdev(dev_t dev, off_t offset, struct asp *asp, caddr_t *addrp, off_t len,
               u_int prot, u_int maxprot, u_int flags, cred_t *cred, struct ddi_mapdev_ctl *ctl,
               ddi_mapdev_handle_t *handlep, void *devprivate);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
dev The device whose memory is to be mapped.
offset The offset within device memory at which the mapping begins.
asp An opaque pointer to the user address space into which the device
memory should be mapped.
addrp Pointer to the starting address within the user address space to which
the device memory should be mapped.
len Length (in bytes) of the memory to be mapped.
prot A bit field that specifies the protections.
maxprot Maximum protection flag possible for attempted mapping.
flags Flags indicating type of mapping.
cred Pointer to the user credentials structure.
ctl A pointer to a ddi_mapdev_ctl(9S) structure. The structure contains
pointers to device driver-supplied functions that manage events on the
device mapping.
handlep An opaque pointer to a device mapping handle. A handle to the new
device mapping is generated and placed into the location pointed to by
*handlep. If the call fails, the value of *handlep is undefined.
devprivate Driver private mapping data. This value is passed into each mapping
call back routine.

DESCRIPTION  
Future releases of Solaris will provide this function for binary and source compatibility.
However, for increased functionality, use devmap_setup(9F) instead. See
devmap_setup(9F) for details.

ddi_mapdev() sets up user mappings to device space. The driver is notified of user
events on the mappings via the entry points defined by ctl.
The user events that the driver is notified of are:

access User has accessed an address in the mapping that has no trans-
lations.
duplication User has duplicated the mapping. Mappings are duplicated
when the process calls fork(2).
unmapping User has called munmap(2) on the mapping or is exiting. See mapdev_access(9E), mapdev_dup(9E), and mapdev_free(9E) for details on these entry points.

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

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

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

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

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

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

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

SEE ALSO fork(2), mmap(2), munmap(2), mapdev_access(9E), mapdev_dup(9E), mapdev_free(9E), mmap(9E), segmap(9E), 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

NOTES Only mappings of type MAP_PRIVATE should be used with ddi_mapdev().
NAME

ddi_mapdev_intercept, ddi_mapdev_nointercept – control driver notification of user accesses

SYNOPSIS

#include <sys/sunddi.h>

int ddi_mapdev_intercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);
int ddi_mapdev_nointercept(ddi_mapdev_handle_t handle, off_t offset, off_t len);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle An opaque pointer to a device mapping handle.
offset An offset in bytes within device memory.
len Length in bytes.

DESCRIPTION

Future releases of Solaris will provide these functions for binary and source compatibility. However, for increased functionality, use devmap_load(9F) or devmap_unload(9F) instead. See devmap_load(9F) and devmap_unload(9F) for details.

The ddi_mapdev_intercept() and ddi_mapdev_nointercept() functions control whether or not user accesses to device mappings created by ddi_mapdev(9F) in the specified range will generate calls to the mapdev_access(9E) entry point. ddi_mapdev_intercept() tells the system to intercept the user access and notify the driver to invalidate the mapping translations. ddi_mapdev_nointercept() tells the system to not intercept the user access and allow it to proceed by validating the mapping translations.

For both routines, the range to be affected is defined by the offset and len arguments. Requests affect the entire page containing the offset and all pages up to and including the page containing the last byte as indicated by offset + len.

Supplying a value of 0 for the len argument affects all addresses from the offset to the end of the mapping. Supplying a value of 0 for the offset argument and a value of 0 for len argument affect all addresses in the mapping.

To manage a device context, a device driver would call ddi_mapdev_intercept() on the context about to be switched out, switch contexts, and then call ddi_mapdev_nointercept() on the context switched in.

RETURN VALUES

ddi_mapdev_intercept() and ddi_mapdev_nointercept() return the following values:

zero Successful completion.
Non-zero An error occurred.

EXAMPLES

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

ddi_mapdev_handle_t cur_hdl;
static int
xmapdev_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.

SEE ALSO mapdev_access(9E), ddi_mapdev(9F)

Writing Device Drivers
NAME
ddi_mapdev_set_device_acc_attr – set the device attributes for the mapping

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

int ddi_mapdev_set_device_acc_attr(ddi_mapdev_handle_t mapping_handle,
    off_t offset, off_t len, ddi_device_acc_attr_t *accattrp, uint_t rnumber);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
  mapping_handle A pointer to a device mapping handle.
  offset    The offset within device memory to which the device access attributes structure applies.
  len       Length (in bytes) of the memory to which the device access attributes structure applies.
  *accattrp Pointer to a ddi_device_acc_attr(9S) structure. Contains the device access attributes to be applied to this range of memory.
  rnumber   Index number to the register address space set.

DESCRIPTION
Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use devmap(9E) instead. See devmap(9E) for details.

The ddi_mapdev_set_device_acc_attr() function assigns device access attributes to a range of device memory in the register set given by rnumber.

*accattrp defines the device access attributes. See ddi_device_acc_attr(9S) for more details.

mapping_handle is a mapping handle returned from a call to ddi_mapdev(9F).

The range to be affected is defined by the offset and len arguments. Requests affect the entire page containing the offset and all pages up to and including the page containing the last byte as indicated by offset+len. Supplying a value of 0 for the len argument affects all addresses from the offset to the end of the mapping. Supplying a value of 0 for the offset argument and a value of 0 for the len argument affect all addresses in the mapping.

RETURN VALUES
The ddi_mapdev_set_device_acc_attr() function returns the following values:

  DDI_SUCCESS     The attributes were successfully set.
  DDI_FAILURE     It is not possible to set these attributes for this mapping handle.

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

SEE ALSO
segmap(9E), ddi_mapdev(9F), ddi_segmap_setup(9F), ddi_device_acc_attr(9S)
NAME

ddi_map_regs, ddi_unmap_regs – map or unmap registers

SYNOPSIS

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

int ddi_map_regs(dev_info_t *dip, u_int rnumber, caddr_t *kaddrp,
off_t offset, off_t len);

void ddi_unmap_regs(dev_info_t *dip, u_int rnumber, caddr_t *kaddrp,
off_t offset, off_t len);

ARGUMENTS

ddi_map_regs()

dip Pointer to the device’s dev_info structure.

rnumber Register set number.

kaddrp Pointer to the base kernel address of the mapped region (set on return).

offset Offset into register space.

len Length to be mapped.

ddi_unmap_regs()

dip Pointer to the device’s dev_info structure.

rnumber Register set number.

kaddrp Pointer to the base kernel address of the region to be unmapped.

offset Offset into register space.

len Length to be unmapped.

INTERFACE

LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_map_regs() maps in the register set given by rnumber. The register number determines which register set will be mapped if more than one exists. The base kernel virtual address of the mapped register set is returned in kaddrp. offset specifies an offset into the register space to start from and len indicates the size of the area to be mapped. If len is non-zero, it overrides the length given in the register set description. See the discussion of the reg property in sbus(4) and vme(4) for more information on register set descriptions. If len and offset are 0, the entire space is mapped.

ddi_unmap_regs() undoes mappings set up by ddi_map_regs(). This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to ddi_unmap_regs() must correspond to a previous call to ddi_map_regs()). Releasing portions of previous mappings is not allowed. rnumber determines which register set will be unmapped if more than one exists. The kaddrp, offset and len specify the area to be unmapped. kaddrp is a pointer to the address returned from ddi_map_regs(); offset and len should match what ddi_map_regs() was called with.
<table>
<thead>
<tr>
<th>RETURN VALUES</th>
<th>ddi_map_regs() returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DDI_SUCCESS on success.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SEE ALSO</th>
<th>sbus(4), vme(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
</tbody>
</table>

modified 27 Jan 1993 | SunOS 5.6 | 9F-301
### NAME

ddi_mem_alloc, ddi_mem_free – allocate and free sequentially accessed memory

### SYNOPSIS

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

int ddi_mem_alloc(dev_info_t *dip, ddi_dma_lim_t *limits, u_int length, u_int flags, caddr_t *kaddrp, u_int *real_length);

void ddi_mem_free(caddr_t kaddr);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

### ARGUMENTS

**ddi_mem_alloc()**

- `dip` : A pointer to the device’s `dev_info` structure.
- `limits` : A pointer to a DMA limits structure for this device (see `ddi_dma_lim_sparc(9S)` or `ddi_dma_lim_x86(9S)`). If this pointer is `NULL`, a default set of DMA limits is assumed.
- `length` : The length in bytes of the desired allocation.
- `flags` : The possible flags `1` and `0` are taken to mean, respectively, wait until memory is available, or do not wait.
- `kaddrp` : On a successful return, `*kaddrp` points to the allocated memory.
- `real_length` : The length in bytes that was allocated. Alignment and padding requirements may cause `ddi_mem_alloc()` to allocate more memory than requested in `length`.

**ddi_mem_free()**

- `kaddr` : The memory returned from a successful call to `ddi_mem_alloc()`.

### DESCRIPTION

`ddi_mem_alloc()` allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as streaming access. The allocation will obey the alignment and padding constraints as specified by the `limits` argument and other limits imposed by the system.

Note that you must still use DMA resource allocation functions (see `ddi_dma_setup(9F)`) to establish DMA resources for the memory allocated using `ddi_mem_alloc()`.

`ddi_mem_alloc()` returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. `ddi_dma_setup(9F)` requires the actual length.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using `ddi_dma_sync(9F)` or `ddi_dma_free(9F)` are required.

`ddi_mem_free()` frees up memory allocated by `ddi_mem_alloc()`.
Kernel Functions for Drivers

**RETURN VALUES**

- **ddi_mem_alloc()** returns:
  - **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.

**SEE ALSO**

- ddi_dma_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*

---

modified 4 Apr 1996

SunOS 5.6

9F-303
NAME

ddi_mem_get8, ddi_mem_get16, ddi_mem_get32, ddi_mem_get64, ddi_mem_getw, ddi_mem_getl, ddi_mem_getll, ddi_mem_getb - read data from mapped device in the memory space or allocated DMA memory.

SYNOPSIS

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

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

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
dev_addr Base device address.

DESCRIPTION

These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi_mem_get8(), ddi_mem_get16(), ddi_mem_get32(), and ddi_mem_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr, in memory space.

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

CONTEXT

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

SEE ALSO

ddi_mem_put8(9F), ddi_mem_rep_get8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_getb</td>
<td>ddi_mem_get8</td>
</tr>
<tr>
<td>ddi_mem_getw</td>
<td>ddi_mem_get16</td>
</tr>
<tr>
<td>ddi_mem_getl</td>
<td>ddi_mem_get32</td>
</tr>
<tr>
<td>ddi_mem_getll</td>
<td>ddi_mem_get64</td>
</tr>
</tbody>
</table>
NAME
ddi_mem_put8, ddi_mem_put16, ddi_mem_put32, ddi_mem_put64, ddi_mem_putb,
ddi_mem_putw, ddi_mem_putl, ddi_mem_putll – write data to mapped device in the
memory space or allocated DMA memory

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

void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

ARGUMENTS
handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
dev_addr Base device address.
value The data to be written to the device.

INTERFACE
LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION
These routines generate a write of various sizes to memory space or allocated DMA
memory. The ddi_mem_put8( ), ddi_mem_put16( ), ddi_mem_put32( ), and
ddi_mem_put64( ) functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively,
to the device address, dev_addr, in memory space.

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

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

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

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_putb</td>
<td>ddi_mem_put8</td>
</tr>
<tr>
<td>ddi_mem_putw</td>
<td>ddi_mem_put16</td>
</tr>
<tr>
<td>ddi_mem_putl</td>
<td>ddi_mem_put32</td>
</tr>
<tr>
<td>ddi_mem_putll</td>
<td>ddi_mem_put64</td>
</tr>
</tbody>
</table>

modified 28 Sep 1996 SunOS 5.6 9F-305
NAME | ddi_mem_rep_get8, ddi_mem_rep_get16, ddi_mem_rep_get32, ddi_mem_rep_get64, ddi_mem_rep_getw, ddi_mem_rep_getl, ddi_mem_rep_getll, ddi_mem_rep_getb – read multiple data from mapped device in the memory space or allocated DMA memory

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

void ddi_mem_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, size_t repcount, uint_t flags);
void ddi_mem_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, size_t repcount, uint_t flags);

INTERFACE LEVEL ARGUMENTS | Solaris DDI specific (Solaris DDI).

handle | The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
host_addr | Base host address.
dev_addr | Base device address.
repcount | Number of data accesses to perform.
flags | Device address flags:

DDI_DEV_AUTOINCR
   Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
   Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION | These routines generate multiple reads from memory space or allocated DMA memory. repcount data is copied from the device address, dev_addr, in memory space to the host address, host_addr. For each input datum, the ddi_mem_rep_get8( ), ddi_mem_rep_get16( ), ddi_mem_rep_get32( ), and ddi_mem_rep_get64( ) functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.

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

9F-306 SunOS 5.6 modified 28 Aug 1996
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

CONTEXT

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

SEE ALSO

ddi_mem_get8(9F), ddi_mem_put8(9F), ddi_mem_rep_put8(9F), ddi_regs_map_setup(9F), ddi_device_acc_attr(9S)

NOTES

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_getb</td>
<td>ddi_mem_rep_get8</td>
</tr>
<tr>
<td>ddi_mem_rep_getw</td>
<td>ddi_mem_rep_get16</td>
</tr>
<tr>
<td>ddi_mem_rep_getl</td>
<td>ddi_mem_rep_get32</td>
</tr>
<tr>
<td>ddi_mem_rep_getll</td>
<td>ddi_mem_rep_get64</td>
</tr>
</tbody>
</table>

modified 28 Aug 1996
NAME

ddi_mem_rep_put8, ddi_mem_rep_put16, ddi_mem_rep_put32, ddi_mem_rep_put64,

ddi_mem_rep_putw, ddi_mem_rep_putl, ddi_mem_rep_putll, ddi_mem_rep_putb

− write multiple data to mapped device in the memory space or allocated DMA memory

SYNOPSIS

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

void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
uint8_t *dev_addr, size_t repcount, uint_t *flags);

void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
uint16_t *dev_addr, size_t repcount, uint_t *flags);

void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
uint32_t *dev_addr, size_t repcount, uint_t *flags);

void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
uint64_t *dev_addr, size_t repcount, uint_t *flags);

INTERFACE LEVEL

ARGUMENTS

handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).

host_addr Base host address.

dev_addr Base device address.

repcount Number of data accesses to perform.

flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION

These routines generate multiple writes to memory space or allocated DMA memory. rep-
count data is copied from the host address, host_addr, to the device address, dev_addr, in
memory space. For each input datum, the ddi_mem_rep_put8(),
ddi_mem_rep_put16(), ddi_mem_rep_put32(), and ddi_mem_rep_put64() functions
write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address.
dev_addr and host_addr must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions will treat the device address, dev_addr, as a memory buffer location on the device and increments its address on the next input datum. However, when the flags argument is set to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

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

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

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mem_rep_putb</td>
<td>ddi_mem_rep_put8</td>
</tr>
<tr>
<td>ddi_mem_rep_putw</td>
<td>ddi_mem_rep_put16</td>
</tr>
<tr>
<td>ddi_mem_rep_putl</td>
<td>ddi_mem_rep_put32</td>
</tr>
<tr>
<td>ddi_mem_rep_putll</td>
<td>ddi_mem_rep_put64</td>
</tr>
</tbody>
</table>
NAME

ddi_mmap_get_model – return data model type of current thread

SYNOPSIS

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

uint_t ddi_mmap_get_model(void);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_mmap_get_model() returns the C Language Type Model which the current thread
expects. ddi_mmap_get_model() is used in combination with
ddi_model_convert_from(9F) in the mmap(9E) driver entry point to determine whether
there is a data model mismatch between the current thread and the device driver. The
device driver might have to adjust the shape of data structures before exporting them to a
user thread which supports a different data model.

RETURN VALUES

DDI_MODEL_ILP32 Current thread expects 32-bit (ILP32) semantics.
DDI_MODEL_LP64 Current thread expects 64-bit (LP64) semantics.
DDI_FAILURE The ddi_mmap_get_model() function was not called from the
mmap(9E) entry point.

CONTEXT

The ddi_mmap_get_model() function can only be called from the mmap(9E) driver entry
point.

EXAMPLES

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.

```c
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;

    #ifdef _MULTI_DATAMODEL
    switch (ddi_model_convert_from(ddi_mmap_get_model())) {
    case DDI_MODEL_ILP32:
        struct data32 *da32p;
```

9F-310 SunOS 5.6 modified 20 Oct 1996
da32p = (struct data32 *)shared_area;
dp = &dtc;
dp->len = da32p->len;
dp->address = da32->address;
break;
}
case DDI_MODEL_NONE:
    break;
}
#endif /* _MULTI_DATAMODEL */
/* continues along using dp */
....

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

Writing Device Drivers
ddi_model_convert_from -- determine data model type mismatch

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

uint_t ddi_model_convert_from(uint_t model);

Solaris DDI specific (Solaris DDI).

The data model type of the current thread.

ddi_model_convert_from() is used to determine if the current thread uses a different C
Language Type Model than the device driver. The 64-bit version of Solaris will require a
64-bit kernel to support both 64-bit and 32-bit user mode programs. The difference
between a 32-bit program and a 64-bit program is in its C Language Type Model: a 32-bit
program is ILP32 (integer, longs, and pointers are 32-bit) and a 64-bit program is LP64
(longs and pointers are 64-bit). There are a number of driver entry points such as
ioctl(9E) and mmap(9E) where it is necessary to identify the C Language Type Model of
the user-mode originator of an kernel event. For example any data which flows between
programs and the device driver or vice versa need to be identical in format. A 64-bit dev-

device driver may need to modify the format of the data before sending it to a 32-bit applica-
tion. 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.

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.

ddi_model_convert_from() can be called from any context.

The following is an example how to use ddi_model_convert_from() in the ioctl() entry
point to support both 32-bit and 64-bit applications.

struct passargs32 {
    int len;
    caddr32_t addr;
};

struct passargs {
    int len;
    caddr_t addr;
};

xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
cred_t *credp, int *rvalp) {
    struct passargs pa;
```c
#ifdef _MULTI_DATAMODEL
switch (ddi_model_convert_from(mode & FMODELS)) {
    case DDI_MODEL_ILP32:
    {
        struct passargs32 pa32;
        ddi_copyin(arg, &pa32, sizeof (struct passargs32), mode);
        pa.len = pa32.len;
        pa.address = pa32.address;
        break;
    }
    case DDI_MODEL_NONE:
    ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
    break;
}
#else /*_MULTI_DATAMODEL */
    ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
#endif /*_MULTI_DATAMODEL */

    do_ioctl(&pa);
    ....
}

SEE ALSO ioctl(9E), mmap(9E), ddi_mmap_get_model(9F)

Writing Device Drivers
```
NAME

ddi_node_name – return the devinfo node name

SYNOPSIS

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

char *ddi_node_name(dev_info_t *dip);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip A pointer the device’s dev_info structure.

DESCRIPTION

ddi_node_name() returns the device node name contained in the dev_info node pointed to by dip.

RETURN VALUES

ddi_node_name() returns the device node name contained in the dev_info structure.

CONTEXT

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

SEE ALSO

ddi_binding_name(9F)

Writing Device Drivers
NAME

ddi_peek, ddi_peek8, ddi_peek16, ddi_peek32, ddi_peek64, ddi_peekc, ddi_peeks,
ddi_peekl, ddi_peekd – read a value from a location

SYNOPSIS

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

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip A pointer to the device’s dev_info structure.
addr Virtual address of the location to be examined.
valuep Pointer to a location to hold the result. If a null pointer is specified, then the
value read from the location will simply be discarded.

DESCRIPTION

These routines cautiously attempt to read a value from a specified virtual address, and
return the value to the caller, using the parent nexus driver to assist in the process where
necessary.

If the address is not valid, or the value cannot be read without an error occurring, an
error code is returned.

The routines are most useful when first trying to establish the presence of a device on the
system in a driver’s probe(9E) or attach(9E) routines.

RETURN VALUES

DDI_SUCCESS The value at the given virtual address was successfully read, and if
valuep is non-null, *valuep will have been updated.
DDI_FAILURE An error occurred while trying to read the location. *valuep is
unchanged.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLES

Check to see that the status register of a device is mapped into the kernel address space:

if (ddi_peek8(dip, csr, (int8_t *)&0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");

    return (DDI_FAILURE);
}

modified 20 Nov 1996

SunOS 5.6

9F-315
Read and log the device type of a particular device:

```c
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "%s%d: device type 0x%x\n",
                ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}
```

SEE ALSO attach(9E), probe(9E), ddi_poke(9F)

Writing Device Drivers

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_peekc</td>
<td>ddi_peek8</td>
</tr>
<tr>
<td>ddi_peeks</td>
<td>ddi_peek16</td>
</tr>
<tr>
<td>ddi_peekl</td>
<td>ddi_peek32</td>
</tr>
<tr>
<td>ddi_peekd</td>
<td>ddi_peek64</td>
</tr>
</tbody>
</table>
NAME
ddi_poke, ddi_poke8, ddi_poke16, ddi_poke32, ddi_poke64, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

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

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
dip A pointer to the device's dev_info structure.
addr Virtual address of the location to be written to.
value Value to be written to the location.

DESCRIPTION
These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's probe(9E) or attach(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the ddi_peek(9F) routines instead if possible.

RETURN VALUES
DDI_SUCCESS The value was successfully written to the given virtual address.
DDI_FAILURE An error occurred while trying to write to the location.

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

SEE ALSO
attach(9E), probe(9E), ddi_peek(9F)

Writing Device Drivers

NOTES
The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:
<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_pokec</td>
<td>ddi_poke8</td>
</tr>
<tr>
<td>ddi_pokes</td>
<td>ddi_poke16</td>
</tr>
<tr>
<td>ddi_poke1l</td>
<td>ddi_poke32</td>
</tr>
<tr>
<td>ddi_poked</td>
<td>ddi_poke64</td>
</tr>
</tbody>
</table>
**NAME**
ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all, ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

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

int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);
int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags, char *name);
int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep, int length);
int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);
void ddi_prop_remove_all(dev_info_t *dip);
```

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

**ARGUMENTS**
<table>
<thead>
<tr>
<th>Function</th>
<th>dev</th>
<th>dip</th>
<th>flags</th>
<th>name</th>
<th>valuep</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_create()</td>
<td>dev_t of the device.</td>
<td>dev_info_t pointer of the device.</td>
<td>flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.</td>
<td>name of property.</td>
<td>pointer to property value.</td>
<td>property length.</td>
</tr>
<tr>
<td>ddi_prop_undefine()</td>
<td>dev_t of the device.</td>
<td>dev_info_t pointer of the device.</td>
<td>flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.</td>
<td>name of property.</td>
<td>pointer to property value.</td>
<td>property length.</td>
</tr>
<tr>
<td>ddi_prop_modify()</td>
<td>dev_t of the device.</td>
<td>dev_info_t pointer of the device.</td>
<td>flag modifiers. The only possible flag value is DDI_PROP_CANSLEEP: Memory allocation may sleep.</td>
<td>name of property.</td>
<td>pointer to property value.</td>
<td>property length.</td>
</tr>
</tbody>
</table>
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.

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_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.

`ddi_prop_undefine()` is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. See `ddi_prop_op(9F)`.

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.

`ddi_prop_modify()` modifies the length and the value of a property. If `ddi_prop_modify()` finds the property in the driver’s property list, allocates memory for the property value and returns `DDI_PROP_SUCCESS`. If the property was not found, the function returns `DDI_PROP_NOT_FOUND`.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.
ddi_prop_remove() unlinks a property from the device’s property list. If ddi_prop_remove() finds the property (an exact match of both name and dev), it unlinks the property, frees its memory, and returns DDI_PROP_SUCCESS, otherwise, it returns DDI_PROP_NOT_FOUND.

ddi_prop_remove_all() removes the properties of all the dev_t’s associated with the dip. It is called before unloading a driver.

RETURN VALUES

- **ddi_prop_create()**:
  - DDI_PROP_SUCCESS on success.
  - DDI_PROP_NO_MEMORY on memory allocation failure.
  - DDI_PROP_INVAL_ARG if an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.

- **ddi_prop_undefined()**: DDI_PROP_SUCCESS on success.
  - DDI_PROP_NO_MEMORY on memory allocation failure.
  - DDI_PROP_INVAL_ARG if an attempt is made to create a property with dev DDI_DEV_T_ANY or if name is NULL or name is the NULL string.

- **ddi_prop_modify()**: DDI_PROP_SUCCESS on success.
  - DDI_PROP_NO_MEMORY on memory allocation failure.
  - DDI_PROP_INVAL_ARG if an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
  - DDI_PROP_NOT_FOUND on property search failure.

- **ddi_prop_remove()**: DDI_PROP_SUCCESS on success.
  - DDI_PROP_INVAL_ARG if an attempt is made to create a property with dev equal to DDI_DEV_T_ANY or if name is NULL or name is the NULL string.
  - DDI_PROP_NOT_FOUND on property search failure.

CONTEXT

If DDI_PROP_CANSLEEP is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

modified 18 Sep 1992 SunOS 5.6 9F-321
EXAMPLES

Create a property called \textit{nblocks} for each partition on a disk.

\begin{verbatim}
for (minor = 0; minor < 8; minor++) {
    (void) ddi_prop_create(makedevice/DDI_MAJOR_T_UNKNOWN, minor),
    dev, DDI_PROP_CANSLEEP, "nblocks", 8192, sizeof (int));
...
}\end{verbatim}

SEE ALSO

\texttt{driver.conf(4), attach(9E), ddi_getproplen(9F), ddi_prop_op(9F), makedevice(9F)}

\textit{Writing Device Drivers}
**NAME**

`ddi_prop_exists` – check for the existence of a property

**SYNOPSIS**

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

int ddi_prop_exists(dev_t match_dev, dev_info_t *dip, u_int flags, char *name);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

- `match_dev`  
  Device number associated with property or `DDI_DEV_T_ANY`.

- `dip`  
  Pointer to the device info node of device whose property list should be searched.

- `flags`  
  Possible flag values are some combination of:
  - `DDI_PROP_DONTPASS`  
    Do not pass request to parent device information node if the property is not found.
  - `DDI_PROP_NOTPROM`  
    Do not look at PROM properties (ignored on platforms that do not support PROM properties).

- `name`  
  String containing the name of the property.

**DESCRIPTION**

`ddi_prop_exists()` checks for the existence of a property regardless of the property value data type. Properties are searched for based on the `dip`, `name`, and `match_dev`. The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If `DDI_PROP_NOTPROM` is not set, search the PROM properties (if they exist).
5. If `DDI_PROP_DONTPASS` is not set, pass this request to the parent device information node.
6. Return 0 if not found and 1 if found.

Usually, the `match_dev` argument should be set to the actual device number that this property is associated with. However, if the `match_dev` argument is `DDI_DEV_T_ANY`, then `ddi_prop_exists()` will match the request regardless of the `match_dev` the property was created with. That is the first property whose name matches `name` will be returned. If a property was created with `match_dev` set to `DDI_DEV_T_NONE` then the only way to look up this property is with a `match_dev` set to `DDI_DEV_T_ANY`. PROM properties are always created with `match_dev` set to `DDI_DEV_T_NONE`.

**modified 22 May 1995**

SunOS 5.6

9F-323
name must always be set to the name of the property being looked up.

RETURN VALUES  

ddi_prop_exists() returns 1 if the property exists and 0 otherwise.

CONTEXT  

These functions can be called from user or kernel context.

EXAMPLES  

The following example demonstrates the use of ddi_prop_exists().

```c
/*
   * Enable "whizzy" mode if the "whizzy-mode" property exists
   */
if (ddi_prop_exists(xx_dev, xx_dip, DDI_PROP_NOTPROM, "whizzy-mode") == 1) {
    xx_enable_whizzy_mode(xx_dip);
} else {
    xx_disable_whizzy_mode(xx_dip);
}
```

SEE ALSO  

ddi_prop_get_int(9F), ddi_prop_lookup(9F), ddi_prop_remove(9F), ddi_prop_update(9F)

Writing Device Drivers
# NAME
ddi_prop_get_int – lookup integer property

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

int ddi_prop_get_int(dev_t match_dev, dev_info_t *dip,
                     u_int flags, char *name, int defvalue);

# ARGUMENTS

match_dev  Device number associated with property or DDI_DEV_T_ANY.
dip        Pointer to the device info node of device whose property list should be searched.
flags      Possible flag values are some combination of:
            DDI_PROP_DONTPASS
                      Do not pass request to parent device information node if property not found.
            DDI_PROP_NOTPROM
                      Do not look at PROM properties (ignored on platforms that do not support PROM properties).
name       String containing the name of the property.
defvalue   An integer value that is returned if the property cannot be found.

# INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

# DESCRIPTION
ddi_prop_get_int() searches for an integer property and, if found, returns the value of the property.
Properties are searched for based on the dip, name, match_dev, and the type of the data (integer). The property search order is as follows:
1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI_PROP_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI_PROP_DONTPASS is not set, pass this request to the parent device information node.
6. Return 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, then ddi_prop_get_int() will match the request regardless of the match_dev the property was created with. If a property was created with match_dev set to DDI_DEV_T_NONE, then the only way to look up this property is with a match_dev set to DDI_DEV_T_ANY. PROM properties are always created with match_dev set to DDI_DEV_T_NONE.
name must always be set to the name of the property being looked up. The return value of the routine is the value of the property. If the property is not found, the argument defvalue is returned as the value of the property.

**RETURN VALUES**  
`ddi_prop_get_int()` returns the value of the property. If the property is not found, the argument defvalue is returned.

**CONTEXT**  
`ddi_prop_get_int()` can be called from user or kernel context.

**EXAMPLES**  
The following example demonstrates the use of `ddi_prop_get_int()`.

```c
/*
 * Get the value of the integer "width" property, using
 * our own default if no such property exists
 */
width = ddi_prop_get_int(xx_dev, xx_dip, 0, "width",
    XX_DEFAULT_WIDTH);
```

**SEE ALSO**  
`ddi_prop_exists(9F)`, `ddi_prop_lookup(9F)`, `ddi_prop_remove(9F)`, `ddi_prop_update(9F)`  
*Writing Device Drivers*
### NAME

ddi_prop_lookup, ddi_prop_lookup_int_array, ddi_prop_lookup_string_array, ddi_prop_lookup_string, ddi_prop_lookup_byte_array, ddi_prop_free – look up property information

### SYNOPSIS

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

int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip, u_int flags, char *name, int **datap, u_int *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip, u_int flags, char *name, char ***datap, u_int *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, u_int flags, char *name, char **datap);

int ddi_prop_lookup_byte_array(dev_t match_dev, dev_info_t *dip, u_int flags, char *name, u_char **datap, u_int *nelementsp);

void ddi_prop_free(void *data);
```

### ARGUMENTS

- **match_dev**
  Device number associated with property or DDI_DEV_T_ANY.

- **dip**
  Pointer to the device info node of device whose property list should be searched.

- **flags**
  Possible flag values are some combination of:

  - **DDI_PROP_DONTPASS**
    Do not pass request to parent device information node if the property is not found.

  - **DDI_PROP_NOTPROM**
    Do not look at PROM properties (ignored on platforms that do not support PROM properties).

- **name**
  String containing the name of the property.

- **nelementsp**
  The address of an unsigned integer which, upon successful return, will contain the number of elements accounted for in the memory pointed at by `datap`. The elements are either integers, strings or bytes depending on the interface used.

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

- **ddi_prop_lookup_string_array()**
  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)`.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_lookup_string()</td>
<td>The address of a pointer to a string which, upon successful return, will</td>
</tr>
<tr>
<td></td>
<td>point to memory containing the NULL terminated string value of the property.</td>
</tr>
<tr>
<td>ddi_prop_lookup_byte_array()</td>
<td>The address of pointer to an array of bytes which, upon successful</td>
</tr>
<tr>
<td></td>
<td>return, will point to memory containing the byte array value of the property.</td>
</tr>
</tbody>
</table>

**INTERFACE LEVEL DESCRIPTION**

Solaris DDI specific (Solaris DDI).

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

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

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

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

For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_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.

`n_elements` 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
Kernel Functions for Drivers

**ddi_prop_lookup_int_array()**

This routine searches for and returns an array of integer property values. An array of integers is defined to *nelements* 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_string_array()**

This routine searches for and returns a property that is an array of strings. *datap* should be set to 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()**

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_string_array()**, **ddi_prop_lookup_string()**, or **ddi_prop_lookup_byte_array()**.

**RETURN VALUES**

- **DDI_PROP_SUCCESS**
  - On success.
- **DDI_PROP_INVAL_ARG**
  - If an attempt is made to look up a property with *match_dev* equal to **DDI_DEV_T_NONE**, *name* is NULL or *name* is the null string.
- **DDI_PROP_NOT_FOUND**
  - Property not found.
- **DDI_PROP_UNDEFINED**
  - Property explicitly not defined (see **ddi_prop_undefine**(9F)).
- **DDI_PROP_CANNOT_DECODE**
  - The value of the property cannot be decoded.

**CONTEXT**

These functions can be called from user or kernel context.

**EXAMPLES**

The following example demonstrates the use of **ddi_prop_lookup()**.

```c
int  *options;
int  noptions;
```

modified 17 Nov 1994 SunOS 5.6 9F-329
* 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);
}
NAME

ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen – get
property information for leaf device drivers

SYNOPSIS

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

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op, int flags,
char *name, caddr_t valuep, int *lengthp);
int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name, int defvalue);
int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char *name, caddr_t valuep,
int *lengthp);
int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char *name,
caddr_t valuep, int *lengthp);
int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *name, int *lengthp);

INTERFACE

LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

dev
Device number associated with property or DDI_DEV_T_ANY as the
wildcard device number.
dip
Pointer to a device info node.
prop_op
Property operator.
flags
Possible flag values are some combination of:

DDI_PROP_DONTPASS
do not pass request to parent device information node if pro-
PERTY not found

DDI_PROP_CANSLEEP
the routine may sleep while allocating memory

DDI_PROP_NOTPROM
do not look at PROM properties (ignored on architectures
that do not support PROM properties).

name
String containing the name of the property.
valuep
If prop_op is PROP_LEN_AND_VAL_BUF, this should be a pointer to the
users buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be
the address of a pointer.
lengthp
On exit, *lengthp will contain the property length. If prop_op is
PROP_LEN_AND_VAL_BUF then before calling ddi_prop_op(), lengthp
should point to an int that contains the length of callers buffer.
defvalue
The value that ddi_getprop() returns if the property is not found.

modified 7 Jun 1993

SunOS 5.6

9F-331
**DESCRIPTION**

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

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

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

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

If `prop_op` is `PROP_LEN_AND_VAL_ALLOC`, and the property is found, `ddi_prop_op()` sets `*lengthp` to the property length. It then attempts to allocate a buffer to return to the caller using the `kmem_alloc(9F)` routine, so that memory can be later recycled using `kmem_free(9F)`. The driver is expected to call `kmem_free()` with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets `*valuep` to point to the allocated buffer, copies the property value into the buffer and returns `DDI_PROP_SUCCESS`. Otherwise, it returns `DDI_PROP_NO_MEMORY`. Note that the `flags` argument may affect the behavior of memory allocation in `ddi_prop_op()`. In particular, if `DDI_PROP_CANSLEEP` is set, then the routine will wait until memory is available to copy the requested property.

`ddi_getprop()` returns boolean and integer-size properties. It is a convenience wrapper for `ddi_prop_op()` with `prop_op` set to `PROP_LEN_AND_VAL_BUF`, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.
ddi_getlongprop() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_ALLOC, so that the routine will allocate space to hold the buffer that will be returned to the caller via *valuep.

ddi_getlongprop_buf() returns arbitrary-size properties. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN_AND_VAL_BUF so the user must supply a buffer.

ddi_getproplen() returns the length of a given property. It is a convenience wrapper for ddi_prop_op() with prop_op set to PROP_LEN.

RETURN VALUES

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_prop_op()</td>
<td>Property found and returned.</td>
</tr>
<tr>
<td>ddi_getlongprop()</td>
<td>Property not found.</td>
</tr>
<tr>
<td>ddi_getlongprop_buf()</td>
<td>Property already explicitly undefined.</td>
</tr>
<tr>
<td>ddi_getproplen()</td>
<td>Property found, but unable to allocate memory.</td>
</tr>
<tr>
<td></td>
<td>lengthp points to the correct property length.</td>
</tr>
<tr>
<td>ddi_getprop()</td>
<td>Property found, but the supplied buffer is too</td>
</tr>
<tr>
<td></td>
<td>small. lengthp points to the correct property</td>
</tr>
<tr>
<td></td>
<td>length.</td>
</tr>
</tbody>
</table>

ddi_getprop() returns:

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

CONTEXT

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

SEE ALSO

ddi_prop_create(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
# NAME

ddi_prop_update, ddi_prop_update_int_array, ddi_prop_update_int,
ddi_prop_update_string_array, ddi_prop_update_string, ddi_prop_update_byte_array – update properties

# SYNOPSIS

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

int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip, 
    char *name, int *data, u_int nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, 
    char *name, int data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, 
    char *name, char **data, u_int 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, u_char *data, u_int nelements);
```

# ARGUMENTS

- **dev**
  - Device number associated with the device.
- **dip**
  - Pointer to the device info node of device whose property list should be updated.
- **name**
  - String containing the name of the property to be updated.
- **nelements**
  - The number of elements contained in the memory pointed at by `data`.

- **data**
  - For `ddi_prop_update_int_array()`: A pointer an integer array with which to update the property.
  - For `ddi_prop_update_int()`: An integer value with which to update the property.
  - For `ddi_prop_update_string_array()`: 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)`.
  - For `ddi_prop_update_string()`: A pointer to a string value with which to update the property.
  - For `ddi_prop_update_byte_array()`: A pointer to a byte array with which to update the property.

# INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

# DESCRIPTION

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

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either ddi_prop_update_int_array() or ddi_prop_update_int(). Attempts to update a property with a function that does correspond to the property value data type will result in the creation of another property with the same name. However, the data type of the new property value will correspond to the data type called out in the function name.

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

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

For the routines ddi_prop_update_int_array(), ddi_prop_update_string_array(), ddi_prop_update_string(), and ddi_prop_update_byte_array() data is a pointer which points to memory containing the value of the property. In each case *data points to a different type of property value. See the individual descriptions of the routines below for details concerning the different values. nelements is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *data.

For the routine ddi_prop_update_int(), 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_string_array() Updates or creates a property that is an array of strings. data must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the argv argument to execve(2).

ddi_prop_update_string() Updates or creates a property that is a single string value. data must be a pointer to a string with which to update the property.
ddi_prop_update_byte_array() Updates or creates a property that is an array of bytes. data should be a pointer to a byte array with which to update the property.

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

RETURN VALUES All of the property update routines return:

- DDI_PROP_SUCCESS on success.
- DDI_PROP_INVAL_ARG if an attempt is made to update a property with name set to NULL or name set to the null string.
- DDI_PROP_CANNOT_ENCODE If the bytes of the property cannot be encoded.

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

EXAMPLES The following example demonstrates the use of ddi_prop_update().

```c
int options[4];

/* Create the "options" integer array with our default values for these parameters */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;

i = ddi_prop_update_int_array(xx_dev, xx_dip, "options", &options, sizeof (options) / sizeof (int));
```

SEE ALSO execve(2), attach(9E), ddi_prop_lookup(9F), ddi_prop_remove(9F), makedevice(9F)

Writing Device Drivers
NAME

ddi_put8, ddi_put16, ddi_put32, ddi_put64, ddi_putb, ddi_putl, ddi_putll, ddi_putw – write data to the mapped memory address, device register or allocated DMA memory address

SYNOPSIS

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

void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr, uint8_t value);
void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr, uint16_t value);
void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr, uint32_t value);
void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr, uint64_t value);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle The data access handle returned from setup calls, such as ddi_regs_map_setup(9F).
value The data to be written to the device.
dev_addr Base device address.

DESCRIPTION

These routines generate a write of various sizes to the mapped memory or device register. The ddi_put8(), ddi_put16(), ddi_put32(), and ddi_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, dev_addr.

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

CONTEXT

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

SEE ALSO

ddi_get8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_rep_put8(9F), ddi_device_acc_attr(9S)

NOTES

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_putb</td>
<td>ddi_put8</td>
</tr>
<tr>
<td>ddi_putw</td>
<td>ddi_put16</td>
</tr>
<tr>
<td>ddi_putl</td>
<td>ddi_put32</td>
</tr>
<tr>
<td>ddi_putll</td>
<td>ddi_put64</td>
</tr>
</tbody>
</table>

modified 30 Sep 1996

SunOS 5.6

9F-337
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).

ARGUMENTS
handle Pointer to a data access handle previously allocated by a call to a setup
routine such as ddi_regs_map_setup(9F).

DESCRIPTION ddi_regs_map_free() frees the mapping represented by the data access handle handle.
This function is provided for drivers preparing to detach themselves from the system,
allowing them to release allocated system resources represented in the handle.

CONTEXT ddi_regs_map_free() must be called from user or kernel context.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus, SBus, ISA, EISA, MCA</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), ddi_regs_map_setup(9F)

Writing Device Drivers
NAME ddi_regs_map_setup – set up a mapping for a register address space

SYNOPSIS

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

int ddi_regs_map_setup(dev_info_t *dip, uint_t rnumber, caddr_t *addrp,
                       offset_t offset, offset_t len, ddi_device_acc_attr_t *accattrp,
                       ddi_acc_handle_t *handlep);
```

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

- **dip**: Pointer to the device’s dev_info structure.
- **rnumber**: Index number to the register address space set.
- **addrp**: Pointer to the mapping address base.
- **offset**: Offset into the register address space.
- **len**: Length to be mapped.
- **accattrp**: Pointer to a device access attribute structure of this device (see ddi_device_acc_attr(9S)).
- **handlep**: Pointer to a data access handle.

DESCRIPTION ddi_regs_map_setup() maps in the register set given by rnumber. The register number determines which register set is mapped if more than one exists.

offset specifies the starting location within the register space and len indicates the size of the area to be mapped. If len is non-zero, it overrides the length given in the register set description. If both len and offset are 0, the entire space is mapped. The base of the mapped register space is returned in addrp.

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

The data access handle is returned in handlep. handlep is opaque – drivers should not attempt to interpret its value. The handle is used by the system to encode information for subsequent data access function calls to maintain a consistent view between the host and the device.

RETURN VALUES ddi_regs_map_setup() returns:

- **DDI_SUCCESS**: Successfully set up the mapping for data access.
- **DDI_FAILURE**: Invalid register number rnumber, offset offset, or length len.
- **DDI_REGS_ACC_CONFLICT**: Cannot enable the register mapping due to access conflicts with other enabled mappings.

modified 1 Jan 1997 SunOS 5.6 9F-339
CONTEXT  
  ddi_regs_map_setup() must be called from user or kernel context.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus, SBus, ISA, EISA, MCA</td>
</tr>
</tbody>
</table>

SEE ALSO  
  attributes(5), ddi_regs_map_free(9F), ddi_device_acc_attr(9S)

Writing Device Drivers
ddi_remove_minor_node — remove a minor node for this dev_info

**SYNOPSIS**

```c
void ddi_remove_minor_node(dev_info_t *dip, char *name);
```

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

**ARGUMENTS**

- `dip` A pointer to the device’s dev_info structure.
- `name` The name of this minor device. If `name` is `NULL` then remove all minor data structures from this dev_info.

**DESCRIPTION**

`ddi_remove_minor_node()` removes a data structure from the linked list of minor data structures that is pointed to by the dev_info structure for this driver.

**EXAMPLES**

This will remove a data structure describing a minor device called foo which is linked into the dev_info structure pointed to by dip.

```c
ddi_remove_minor_node(dip, "foo");
```

**SEE ALSO**

attached(9E), detach(9E), ddi_create_minor_node(9F)

*Writing Device Drivers*

modified 10 Mar 1992

SunOS 5.6

9F-341
NAME  
ddi_rep_get8, ddi_rep_get16, ddi_rep_get32, ddi_rep_get64, ddi_rep_getw, ddi_rep_getl, 
ddi_rep_getll, ddi_rep_getb – read data from the mapped memory address, device regis-
ter or allocated DMA memory address

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

void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr, 
size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr, 
size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr, 
size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr, 
size_t repcount, uint_t flags);

INTERFACE  
LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
handle  The data access handle returned from setup calls, such as 
ddi_regs_map_setup(9F).

host_addr  Base host address.

dev_addr  Base device address.

repcount  Number of data accesses to perform.

flags  Device address flags:

DDI_DEV_AUTOINCR  
Automatically increment the device address, dev_addr, dur-
ing data accesses.

DDI_DEV_NO_AUTOINCR  
Do not advance the device address, dev_addr, during data 
accesses.

DESCRIPTION  
These routines generate multiple reads from the mapped memory or device register. rep-
count data is copied from the device address, dev_addr, to the host address, host_addr. For 
each input datum, the ddi_rep_get8(), ddi_rep_get16(), ddi_rep_get32(), and 
ddi_rep_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, 
from the device address, dev_addr. dev_addr and host_addr must be aligned to the datum 
boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view 
between the host and the device based on the encoded information in the data access 
handle. The translation may involve byte-swapping if the host and the device have 
incompatible endian characteristics.
When the flags argument is set to DDI_DEV_AUTOINCR, these functions treat the device address, dev_addr, as a memory buffer location on the device and increment its address on the next input datum. However, when the flags argument is to DDI_DEV_NO_AUTOINCR, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

RETURN VALUES

These functions return the value read from the mapped address.

CONTEXT

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

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

NOTES

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_getb</td>
<td>ddi_rep_get8</td>
</tr>
<tr>
<td>ddi_rep_getw</td>
<td>ddi_rep_get16</td>
</tr>
<tr>
<td>ddi_rep_getl</td>
<td>ddi_rep_get32</td>
</tr>
<tr>
<td>ddi_rep_getll</td>
<td>ddi_rep_get64</td>
</tr>
<tr>
<td>NAME</td>
<td>ddi_report_dev – announce a device</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>SYNOPSIS</td>
<td></td>
</tr>
</tbody>
</table>
|            | #include <sys/conf.h>
|            | #include <sys/ddi.h>
|            | #include <sys/sunddi.h>
|            | void ddi_report_dev(dev_info_t *dip); |
| INTERFACE  | Solaris DDI specific (Solaris DDI). |
| LEVEL      |                                       |
| ARGUMENTS  | dip a pointer the device’s dev_info structure. |
| DESCRIPTION| ddi_report_dev() prints a banner at boot time, announcing the device pointed to by dip. The banner is always placed in the system logfile (displayed by dmesg(1M)), but is only displayed on the console if the system was booted with the verbose (−v) argument. |
| CONTEXT    | ddi_report_dev() can be called from user or interrupt context. |
| SEE ALSO   | dmesg(1M), kernel(1M) |
|            | Writing Device Drivers |

Writing Device Drivers
NAME
ddi_rep_put8, ddi_rep_put16, ddi_rep_put32, ddi_rep_put64, ddi_rep_putb,
ddi_rep_putw, ddi_rep_putl, ddi_rep_putll – write data to the mapped memory address,
device register or allocated DMA memory address

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

void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr, uint8_t *dev_addr,
                  size_t repcount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr, uint16_t *dev_addr,
                   size_t repcount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr, uint32_t *dev_addr,
                   size_t repcount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr, uint64_t *dev_addr,
                   size_t repcount, uint_t flags);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

ARGUMENTS
handle The data access handle returned from setup calls, such as
ddi_regs_map_setup(9F).
host_addr Base host address.
dev_addr Base device address.
repcount Number of data accesses to perform.
flags Device address flags:

DDI_DEV_AUTOINCR
Automatically increment the device address, dev_addr, during data accesses.

DDI_DEV_NO_AUTOINCR
Do not advance the device address, dev_addr, during data accesses.

DESCRIPTION
These routines generate multiple writes to the mapped memory or device register.
repcount data is copied from the host address, host_addr, to the device address, dev_addr.
For each input datum, the ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), and
ddi_rep_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to
the device address, dev_addr. dev_addr and host_addr must be aligned to the datum boundary described by the function.
Each individual datum will automatically be translated to maintain a consistent view
between the host and the device based on the encoded information in the data access
handle. The translation may involve byte-swapping if the host and the device have
incompatible endian characteristics.

modified 10 Sep 1996
SunOS 5.6
9F-345
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 writing to a data register.

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

SEE ALSO
ddi_get8(9F), ddi_put8(9F), ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_rep_get8(9F), ddi_device_acc_attr(9S)

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_rep_putb</td>
<td>ddi_rep_put8</td>
</tr>
<tr>
<td>ddi_rep_putw</td>
<td>ddi_rep_put16</td>
</tr>
<tr>
<td>ddi_rep_putl</td>
<td>ddi_rep_put32</td>
</tr>
<tr>
<td>ddi_rep_putll</td>
<td>ddi_rep_put64</td>
</tr>
</tbody>
</table>
NAME
ddi_root_node – get the root of the dev_info tree

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

dev_info_t *ddi_root_node(void);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION
ddi_root_node() returns a pointer to the root node of the device information tree.

RETURN VALUES
ddi_root_node() returns a pointer to a device information structure.

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

SEE ALSO
Writing Device Drivers

modified 19 Nov 1992

SunOS 5.6

9F-347
**NAME**
ddi_segmap, ddi_segmap_setup – set up a user mapping using seg_dev

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

int ddi_segmap(dev_t dev, off_t offset, struct asp *asp, caddr_t *addrp, off_t len,
               u_int prot, u_int maxprot, u_int flags, cred_t *credp);

int ddi_segmap_setup(dev_t dev, off_t offset, struct asp *asp, caddr_t *addrp, off_t len,
                      u_int prot, u_int maxprot, u_int flags, cred_t *credp,
                      ddi_device_acc_attr_t *accattrp, u_int rnumber);
```

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

**ARGUMENTS**
- **dev**
  - The device whose memory is to be mapped.
- **offset**
  - The offset within device memory at which the mapping begins.
- **asp**
  - An opaque pointer to the user address space into which the device memory should be mapped.
- **addrp**
  - Pointer to the starting address within the user address space to which the device memory should be mapped.
- **len**
  - Length (in bytes) of the memory to be mapped.
- **prot**
  - A bit field that specifies the protections. Some combinations of possible settings are:
    - PROT_READ: Read access is desired.
    - PROT_WRITE: Write access is desired.
    - PROT_EXEC: Execute access is desired.
    - PROT_USER: User-level access is desired (the mapping is being done as a result of a `mmap(2)` system call).
    - PROT_ALL: All access is desired.
- **maxprot**
  - `maxprot` Maximum protection flag possible for attempted mapping (the PROT_WRITE bit may be masked out if the user opened the special file read-only). If (maxprot & prot) != prot then there is an access violation.
- **flags**
  - Flags indicating type of mapping. Possible values are (other bits may be set):
    - MAP_PRIVATE: Changes are private.
    - MAP_SHARED: Changes should be shared.
    - MAP_FIXED: The user specified an address in *addrp rather than letting the system pick and address.
- **credp**
  - Pointer to user credential structure.
ddi_segmap_setup()  

<table>
<thead>
<tr>
<th>dev_acc_attr</th>
<th>Pointer to a ddi_device_acc_attr(9S) structure which contains the device access attributes to apply to this mapping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rnumber</td>
<td>Index number to the register address space set.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**  
Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use ddi_devmap_segmap(9F) instead. See ddi_devmap_segmap(9F) for details.

ddi_segmap() and ddi_segmap_setup() set up user mappings to device space. When setting up the mapping, the ddi_segmap() and ddi_segmap_setup() routines call the mmap(9E) entry point to validate the range to be mapped. When a user process accesses the mapping, the drivers mmap(9E) entry point is again called to retrieve the page frame number that needs to be loaded. The mapping translations for that page are then loaded on behalf of the driver by the DDI framework.

ddi_segmap() is typically used as the segmap(9E) entry in the cb_ops(9S) structure for those devices that do not choose to provide their own segmap(9E) entry point. However, some drivers may have their own segmap(9E) entry point to do some initial processing on the parameters and then call ddi_segmap() to establish the default memory mapping.

ddi_segmap_setup() is used in the drivers segmap(9E) entry point to set up the mapping and assign device access attributes to that mapping. rnumber specifies the register set representing the range of device memory being mapped. See ddi_device_acc_attr(9S) for details regarding what device access attributes are available.

ddi_segmap_setup() cannot be used directly in the cb_ops(9S) structure and requires a driver to have a segmap(9E) entry point.

**RETURN VALUES**  

ddi_segmap() and ddi_segmap_setup() return the following values:

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Non-zero  
An error occurred. In particular, they return ENXIO if the range to be mapped is invalid.

**CONTEXT**  

ddi_segmap() and ddi_segmap_setup() can be called from user or kernel context only.

**SEE ALSO**  
mmap(2), mmap(9E), segmap(9E), 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.
NAME

ddi_slaveonly – tell if a device is installed in a slave access only location

SYNOPSIS

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

int ddi_slaveonly(dev_info_t *dip);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip A pointer to the device’s dev_info structure.

DESCRIPTION

ddi_slaveonly() tells the caller if the bus, or part of the bus that the device is installed on, does not permit the device to become a DMA master, that is, whether the device has been installed in a slave access only slot.

RETURN VALUES

DDI_SUCCESS The device has been installed in a slave access only location.
DDI_FAILURE The device has not been installed in a slave access only location.

CONTEXT

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

SEE ALSO

Writing Device Drivers
Kernel Functions for Drivers

NAME
ddi_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free,
ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

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

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state_p);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

state_p Address of the opaque state pointer which will be initialized by
ddi_soft_state_init() to point to implementation dependent data.

size Size of the item which will be allocated by subsequent calls to
ddi_soft_state_zalloc().

n_items A hint of the number of items which will be preallocated; zero is
allowed.

state An opaque pointer to implementation-dependent data that describes the
soft state.

item The item number for the state structure; usually the instance number of
the associated devinfo node.

DESCRIPTION
Most device drivers maintain state information with each instance of the device they control; for example, a soft copy of a device control register, a mutex that must be held while accessing a piece of hardware, a partition table, or a unit structure. These utility routines are intended to help device drivers manage the space used by the driver to hold such state information.

For example, if the driver holds the state of each instance in a single state structure, these routines can be used to dynamically allocate and deallocate a separate structure for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, state_p, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.

The routine ddi_soft_state_init() is usually called in the drivers _init(9E) routine to initialize the state pointer, set the size of the soft state structure, and to allow the driver to pre-allocate a given number of such structures if required.

The routine ddi_soft_state_zalloc() is usually called in the drivers attach(9E) routine. The routine is passed an item number which is used to refer to the structure in subsequent calls to ddi_get_soft_state() and ddi_soft_state_free(). The item number is
usually just the instance number of the devinfo node, obtained with
**ddi_get_instance**(9F). The routine attempts to allocate space for the new structure, and if
the space allocation was successful, **DDI_SUCCESS** is returned to the caller.

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 drivers **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 drivers **_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 addi-
tional 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** param-
eter 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_zalloc**( ) can be called from user context only,
since they may internally call **kmem_zalloc**(9F) with the **KM_SLEEP** flag.

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

**EXAMPLE**

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 por-
tions of the code that deal with creating and removing the drivers data structures.

```c
typedef struct {
    volatile caddr_t *csr; /* device registers */
    kmutex_t csr_mutex;   /* protects ‘csr’ field */
    unsigned int state;
    dev_info_t *dip;      /* back pointer to devinfo */
} devstate_t;
```

**9F-352 SunOS 5.6 modified 7 Jun 1993**
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_SUCCESS);
    }
    return (DDI_FAILURE);
}
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {

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

    default:
        return (DDI_FAILURE);
    }
}

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

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

SEE ALSO _®ni(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.

9F-354 SunOS 5.6 modified 7 Jun 1993
DIAGNOSTICS

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

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

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

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

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

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

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.
NAME  ddi_umem_alloc, ddi_umem_free – allocate and free page-aligned kernel memory

SYNOPSIS  

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

void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t *cookiep);

void ddi_umem_free(ddi_umem_cookie_t cookie);
```

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  

**ddi_umem_alloc()**

- `size`  Number of bytes to allocate.
- `flag`  Used to determine the sleep and pageable conditions.
  Possible sleep flags are `DDI_UMEM_SLEEP` which allows sleeping until memory is available, and `DDI_UMEM_NOSLEEP` which returns NULL immediately if memory is not available.
  The default condition is to allocate locked memory; this can be changed to allocate pageable memory using the `DDI_UMEM_PAGEABLE` flag.
- `cookiep`  Pointer to a kernel memory cookie.

**ddi_umem_free()**

- `cookie`  A kernel memory cookie allocated in `ddi_umem_alloc()`.

DESCRIPTION  

`ddi_umem_alloc()` allocates page-aligned kernel memory and returns a pointer to the allocated memory. The number of bytes allocated is a multiple of the system page size (roundup of `size`). The allocated memory can be used in the kernel and can be exported to user space. See `devmap(9E)` and `devmap_umem_setup(9F)` for further information.

**flag** determines whether the caller can sleep for memory and whether the allocated memory is locked or not. `DDI_UMEM_SLEEP` allocations may sleep but are guaranteed to succeed. `DDI_UMEM_NOSLEEP` allocations do not sleep but may fail (return NULL) if memory is currently unavailable. If `DDI_UMEM_PAGEABLE` is set, pageable memory will be allocated. These pages can be swapped out to secondary memory devices. The initial contents of memory allocated using `ddi_umem_alloc()` is zero-filled.

*cookiep* is a pointer to the kernel memory cookie that describes the kernel memory being allocated. A typical use of `cookiep` is in `devmap_umem_setup(9F)` when the drivers want to export the kernel memory to a user application.

To free the allocated memory, a driver calls `ddi_umem_free()` with the cookie obtained from `ddi_umem_alloc()`. `ddi_umem_free()` releases the entire buffer.

RETURN VALUES  

- **Non-null**  Successful completion. `ddi_umem_alloc()` returns a pointer to the allocated memory.
- **NULL**  Memory cannot be allocated by `ddi_umem_alloc()` because `DDI_UMEM_NOSLEEP` is set and the system is out of resources.

9F-356  SunOS 5.6  modified 14 Jan 1997
ddi_umem_alloc() can be called from any context if flag is set to DDI_UMEM_NOSLEEP. If DDI_UMEM_SLEEP is set, ddi_umem_alloc() can be called from user and kernel context only. ddi_umem_free() can be called from any context.

SEE ALSO ddi_umem_alloc(), ddi_umem_free()

Writing Device Drivers

Warning

Setting the DDI_UMEM_PAGEABLE flag in ddi_umem_alloc() will result in an allocation of pageable memory. Because these pages can be swapped out to secondary memory devices, drivers should use this flag with care. This memory should not be used for synchronization objects such as locks and condition variables. See mutex(9F), semaphore(9F), rwlock(9F), and condvar(9F). This memory also should not be accessed in the driver interrupt routines.

Memory allocated using ddi_umem_alloc() without setting DDI_UMEM_PAGEABLE flag cannot be paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.

Excessive use of kernel memory is likely to effect overall system performance. Over-commitment of kernel memory may cause unpredictable consequences.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing an invalid pointer, will cause the system to corrupt data or panic.

ddi_umem_alloc(0, flag, cookie) always returns NULL. ddi_umem_free(NULL) has no effects on system.
**NAME**
delay – delay execution for a specified number of clock ticks

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

void delay(clock_t ticks);
```

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

**ARGUMENTS**
ticks The number of clock cycles to delay.

**DESCRIPTION**
delay() provides a mechanism for a driver to delay its execution for a given period of time. Since the speed of the clock varies among systems, drivers should base their time values on microseconds and use `drv_usecnthz` to convert microseconds into clock ticks.

delay() uses `timeout` to schedule an internal function to be called after the specified amount of time has elapsed. delay() then waits until the function is called.

delay() does not busy-wait. If busy-waiting is required, use `drv_usecwait`.

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

**EXAMPLES**
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.

```c
struct device {
  /* layout of physical device registers */
  int control; /* physical device control word */
  int status; /* physical device status word */
  short xmit_char; /* transmit character to device */
};

... /* get device registers */

register struct device *rp = ...

while (rp->status & NOPAPER) {
  /* while printer is out of paper */
  cmn_err(CE_WARN, "xx_write: NO PAPER in printer %d\007",
          (getminor(dev) & 0xf));
  /* wait one minute and try again */
  delay(60 * drv_usecnthz(1000000));
}
```

SunOS 5.6 modified 20 Sep 1996
SEE ALSO  biodone(9F), biowait(9F), drv_hztousec(9F), drv_usectohz(9F), drv_usecwait(9F), timeout(9F), untimeout(9F)

Writing Device Drivers
devmap_default_access – default driver memory access function

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

int devmap_default_access(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len, u_int type, u_int rw)

An opaque mapping handle that the system uses to describe the mapping.

Driver private mapping data.

User offset within the logical device memory at which the access begins.

Length (in bytes) of the memory being accessed.

Type of access operation.

Type of access.

devmap_default_access() is a function providing the semantics of devmap_access(9E). The drivers call devmap_default_access() to handle the mappings that do not support context switching. The drivers should call devmap_do_ctxmgt(9F) for the mappings that support context management.

devmap_default_access() can either be called from devmap_access(9E) or be used as the devmap_access(9E) entry point. The arguments dhp, pvtp, off, len, type, and rw are provided by the devmap_access(9E) entry point and must not be modified.

0 Successful completion.

Non-zero An error occurred.

devmap_default_access() must be called from the driver’s devmap_access(9E) entry point.

The following shows an example of using devmap_default_access() in the devmap_access(9E) entry point.

...#define OFF_DO_CTXMGT 0x40000000
#define OFF_NORMAL 0x40100000
#define CTXMG_SIZE 0x100000
#define NORMAL_0x100000

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

static int xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len, u_int type, u_int 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);
    }
devmap_default_access (9F)
NAME
devmap_devmem_setup, devmap_umem_setup – set driver memory mapping parameters

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

int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip,
                        struct devmap_callback_ctl *callbackops, u_int rnumber, offset_t roff,
                        size_t len, u_int maxprot, u_int 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, u_int maxprot, u_int flags, ddi_device_acc_attr_t *accattrp)

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

devmap_devmem_setup()
dhp
An opaque mapping handle that the system uses to describe the mapping.
dip
Pointer to the device’s dev_info structure.
callbackops
Pointer to a devmap_callback_ctl(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping. The framework will copy the structure to the system private memory.
rnumber
Index number to the register address space set.
roff
Offset into the register address space.
len
Length (in bytes) of the mapping to be mapped.
maxprot
Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:

  PROT_READ     Read access is allowed.
  PROT_WRITE    Write access is allowed.
  PROT_EXEC     Execute access is allowed.
  PROT_USER     User-level access is allowed (the mapping is being done as a result of a mmap(2) system call).
  PROT_ALL      All access is allowed.

flags
Must be set to 0.
accattrp
Pointer to a ddi_device_acc_attr(9S) structure. The structure contains the device access attributes to be applied to this range of memory.
**devmap_devmem_setup()**

```
int devmap_devmem_setup(dhp, dip, callbacks, cookie, koff, len, maxprot, flags, accattrp);
```

- **dhp**: An opaque data structure that the system uses to describe the mapping.
- **dip**: Pointer to the device’s `dev_info` structure.
- **callbacks**: Pointer to a `devmap_callback_ctl(9S)` structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.
- **cookie**: A kernel memory cookie (see `ddi_umem_alloc(9F)`).
- **koff**: Offset into the kernel memory defined by `cookie`.
- **len**: Length (in bytes) of the mapping to be mapped.
- **maxprot**: Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:
  - `PROT_READ`: Read access is allowed.
  - `PROT_WRITE`: Write access is allowed.
  - `PROT_EXEC`: Execute access is allowed.
  - `PROT_USER`: User-level access is allowed (the mapping is being done as a result of a `mmap(2)` system call).
  - `PROT_ALL`: All access is allowed.
- **flags**: Must be set to 0.
- **accattrp**: Pointer to a `ddi_device_acc_attr(9S)` structure. The structure contains the device access attributes to be applied to this range of memory.

**DESCRIPTION**

`devmap_devmem_setup()` and `devmap_umem_setup()` are used in the `devmap(9E)` entry point to pass mapping parameters from the driver to the system.

`dhp` is a device mapping handle that the system uses to store all mapping parameters of a physical contiguous memory. The system copies the data pointed to by `callbacks` to a system private memory. This allows the driver to free the data after returning from either `devmap_devmem_setup()` or `devmap_umem_setup()`. The driver is notified of user events on the mappings via the entry points defined by `devmap_callback_ctl(9S)`.

The driver is notified of the following user events:

- **Mapping Setup**: User has called `mmap(2)` to create a mapping to the device memory.
- **Access**: User has accessed an address in the mapping that has no translations.
- **Duplication**: User has duplicated the mapping. Mappings are duplicated when the process calls `fork(2)`.
- **Unmapping**: User has called `munmap(2)` on the mapping or is exiting, `exit(2)`.
See **devmap_map**(9E), **devmap_access**(9E), **devmap_dup**(9E), and **devmap_unmap**(9E) for details on these entry points.

By specifying a valid *callbackops* to the system, device drivers can manage events on a device mapping. For example, the **devmap_access**(9E) entry point allows the drivers to perform context switching by unloading the mappings of other processes and to load the mapping of the calling process. Device drivers may specify **NULL** to *callbackops* which means the drivers do not want to be notified by the system.

The maximum protection allowed for the mapping is specified in *maxprot*. *accattrp* defines the device access attributes. See **ddi_device_acc_attr**(9S) for more details.

**devmap_devmem_setup()** is used for device memory to map in the register set given by *rnumber* and the offset into the register address space given by *roff*. The system uses *rnumber* and *roff* to go up the device tree to get the physical address that corresponds to *roff*. The range to be affected is defined by *len* and *roff*. The range from *roff* to *roff + len* must be a physical contiguous memory and page aligned.

Drivers use **devmap_umem_setup()** for kernel memory to map in the kernel memory described by *cookie* and the offset into the kernel memory space given by *koff*. *cookie* is a kernel memory pointer obtained from **ddi_umem_alloc**(9F). If *cookie* is **NULL**, **devmap_umem_setup()** returns -1. The range to be affected is defined by *len* and *koff*. The range from *koff* to *koff + len* must be within the limits of the kernel memory described by *koff + len* and must be page aligned.

Drivers use **devmap_umem_setup()** to export the kernel memory allocated by **ddi_umem_alloc**(9F) to user space. The system selects a user virtual address that is aligned with the kernel virtual address being mapped to avoid cache incoherence if the mapping is not **MAP_FIXED**.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred.</td>
</tr>
</tbody>
</table>

**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

---

modified 22 Jan 1997

SunOS 5.6
NAME

devmap_do_ctxmgt – perform device context switching on a mapping

SYNOPSIS

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

int devmap_do_ctxmgt(devmap_cookie_t, dhp, void *pvtp, offset_t off, size_t len,
                      u_int type, u_int rw, int (*devmap_contextmgt)(devmap_cookie_t,
                      void *, offset_t, size_t, u_int, u_int ));
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dhp An opaque mapping handle that the system uses to describe the mapping.
pvtp Driver private mapping data.
off User offset within the logical device memory at which the access begins.
len Length (in bytes) of the memory being accessed.
devmap_contextmgt The address of driver function that the system will call to perform context switching on a mapping. See devmap_contextmgt(9E) for details.
type Type of access operation. Provided by devmap_access(9E). Should not be modified.
rw Direction of access. Provided by devmap_access(9E). Should not be modified.

DESCRIPTION

Device drivers call devmap_do_ctxmgt() in the devmap_access(9E) entry point to perform device context switching on a mapping. devmap_do_ctxmgt() passes a pointer to a driver supplied callback function, devmap_contextmgt(9E), to the system that will perform the actual device context switching. If devmap_contextmgt(9E) is not a valid driver callback function, the system will fail the memory access operation which will result in a SIGSEGV or SIGBUS signal being delivered to the process.

devmap_do_ctxmgt() performs context switching on the mapping object identified by dhp and pvtp in the range specified by off and len. The arguments dhp, pvtp, type, and rw are provided by the devmap_access(9E) entry point and must not be modified. The range from off to off+len must support context switching.

The system will pass through dhp, pvtp, off, len, type, and rw to devmap_contextmgt(9E) in order to perform the actual device context switching. The return value from devmap_contextmgt(9E) will be returned directly to devmap_do_ctxmgt().

RETURN VALUES

0 Successful completion.
Non-zero An error occurred.
**CONTEXT**

`devmap_do_ctxmgt()` must be called from the driver’s `devmap_access(9E)` entry point.

**EXAMPLES**

The following shows an example of using `devmap_do_ctxmgt()` in the `devmap_access(9E)` entry point.

```c
... 
#define OFF_DO_CTXMGT 0x40000000
#define OFF_NORMAL 0x40100000
#define CTXMGT_SIZE 0x100000
#define NORMAL_SIZE 0x100000
/
/*
 * Driver devmap_contextmgt(9E) callback function.
 */
static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset, 
       size_t length, u_int type, u_int 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, u_int type, u_int 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);
    }
```

modified 22 Jan 1997  SunOS 5.6  9F-367
/= 
* perform context switching 
*/
err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
   rw, xx_context_mgt);

/=
* check if off is within the range that does normal
* memory mapping. 
*/
} else if ((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) { 
   if ((len + off) > (OFF_NORMAL + NORMAL_SIZE))
      return (-1);
   err = devmap_default_access(dhp, pvtp, off, len, type, rw);
} else
   return (-1);

return (err);
}

SEE ALSO devmap_access(9E), devmap_contextmgt(9E), devmap_default_access(9F)

Writing Device Drivers
NAME
devmap_set_ctx_timeout – set the timeout value for the context management callback

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

void devmap_set_ctx_timeout(devmap_cookie_t dhp, clock_t ticks)

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
dhp An opaque mapping handle that the system uses to describe the mapping.
ticks Number of clock ticks to wait between successive calls to the context management callback function.

DESCRIPTION
devmap_set_ctx_timeout() specifies the time interval for the system to wait between successive calls to the driver’s context management callback function, devmap_contextmgnt(9E).
Device drivers typically call devmap_set_ctx_timeout() in the devmap_map(9E) routine. If the drivers do not call devmap_set_ctx_timeout() to set the timeout value, the default timeout value of 0 will result in no delay between successive calls to the driver’s devmap_contextmgnt(9E) callback function.

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

SEE ALSO
devmap_contextmgnt(9E), devmap_map(9E), timeout(9F)
devmap_setup (9F) Kernel Functions for Drivers

NAME

devmap_setup, ddi_devmap_segmap – set up a user mapping to device memory using the devmap framework

SYNOPSIS

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

int devmap_setup(dev_t dev, offset_t off, ddi_as_handle_t as, caddr_t *addrp,
size_t len, u_int prot, u_int maxprot, u_int 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, u_int prot, u_int maxprot, u_int flags, cred_t *cred)

INTERFACE

LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

dev Device whose memory is to be mapped.
off User offset within the logical device memory at which the mapping begins.
as An opaque data structure that describes the address space into which the device memory should be mapped.
addrp Pointer to the starting address in the address space into which the device memory should be mapped.
len Length (in bytes) of the memory to be mapped.
prot A bit field that specifies the protections. Some possible settings combinations are:

    PROT_READ  Read access is desired.
    PROT_WRITE Write access is desired.
    PROT_EXEC  Execute access is desired.
    PROT_USER  User-level access is desired (the mapping is being done as a result of a mmap(2) system call).
    PROT_ALL   All access is desired.

maxprot Maximum protection flag possible for attempted mapping; the PROT_WRITE bit may be masked out if the user opened the special file read-only.
flags Flags indicating type of mapping. The following flags can be specified:

    MAP_PRIVATE Changes are private.
    MAP_SHARED  Changes should be shared.
    MAP_FIXED   The user specified an address in *addrp rather than letting the system choose an address.

cred Pointer to the user credential structure.
**DESCRIPTION**

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

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

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

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

**RETURN VALUES**

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

**CONTEXT**

`devmap_setup()` and `ddi_devmap_segmap()` can be called from user or kernel context only.

**SEE ALSO**

`mmap(2)`, `devmap(9E)`, `segmap(9E)`, `ddi_segmap(9F)`, `ddi_segmap_setup(9F)`, `cb_ops(9S)`

*Writing Device Drivers*
NAME
devmap_load, devmap_unload – control validation of memory address translations

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
int devmap_load(devmap_cookie_t *dhp, offset_t off, size_t len, u_int type, u_int rw)
int devmap_unload(devmap_cookie_t *dhp, offset_t off, size_t len)

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
dhp
An opaque mapping handle that the system uses to describe the mapping.
off
User offset within the logical device memory at which the loading or unloading of the address translations begins.
len
Length (in bytes) of the range being affected.

devmap_load() only

<table>
<thead>
<tr>
<th>type</th>
<th>Type of access operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td>Direction of access.</td>
</tr>
</tbody>
</table>

DESCRIPTION

devmap_unload() and devmap_load() are used to control the validation of the memory mapping described by dhp in the specified range. devmap_unload() invalidates the mapping translations and will generate calls to the devmap_access(9E) entry point next time the mapping is accessed. The drivers use devmap_load() to validate the mapping translations during memory access.

A typical use of devmap_unload() and devmap_load() is in the driver’s context management callback function, devmap_contextmgt(9E). To manage a device context, a device driver calls devmap_unload() on the context about to be switched out. It switches contexts, and then calls devmap_load() on the context switched in. devmap_unload() can be used to unload the mappings of other processes as well as the mappings of the calling process, but devmap_load() can only be used to load the mappings of the calling process. Attempting to load another process’s mappings with devmap_load() will result in a system panic.

For both routines, the range to be affected is defined by the off and len arguments. Requests affect the entire page containing the off and all pages up to and including the page containing the last byte as indicated by off + len. The arguments type and rw are provided by the system to the calling function (for example, devmap_contextmgt(9E)) and should not be modified.

Supplying a value of 0 for the len argument affects all addresses from the off to the end of the mapping. Supplying a value of 0 for the off argument and a value of 0 for len argument affect all addresses in the mapping.

A non-zero return value from either devmap_unload() or devmap_load() will cause the corresponding operation to fail. The failure may result in a SIGSEGV or SIGBUS signal being delivered to the process.
Kernel Functions for Drivers

**RETURN VALUES**

0  Successful completion.
Non-zero  An error occurred.

**CONTEXT**

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

**EXAMPLES**

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

```c
struct xx_context cur_ctx;

static int
xxdevmap_contextmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
 size_t len, u_int type, u_int 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

modified 22 Jan 1997  SunOS 5.6  9F-373
NAME    disksort – single direction elevator seek sort for buffers

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

void disksort(struct diskhd *dp, struct buf *bp);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

dp    A pointer to a diskhd structure. A diskhd structure is essentially identical to head of a buffer structure (see buf(9S)). The only defined items of interest for this structure are the av_forw and av_back structure elements which are used to maintain the front and tail pointers of the forward linked I/O request queue.

bp    A pointer to a buffer structure. Typically this is the I/O request that the driver receives in its strategy routine (see strategy(9E)). The driver is responsible for initializing the b_resid structure element to a meaningful sort key value prior to calling disksort().

DESCRIPTION The function disksort() sorts a pointer to a buffer into a single forward linked list headed by the av_forw element of the argument *dp.

It uses a one-way elevator algorithm that sorts buffers into the queue in ascending order based upon a key value held in the argument buffer structure element b_resid.

This value can either be the driver calculated cylinder number for the I/O request described by the buffer argument, or simply the absolute logical block for the I/O request, depending on how fine grained the sort is desired to be or how applicable either quantity is to the device in question.

The head of the linked list is found by use of the av_forw structure element of the argument *dp. The tail of the linked list is found by use of the av_back structure element of the argument *dp. The av_forw element of the *bp argument is used by disksort() to maintain the forward linkage. The value at the head of the list presumably indicates the currently active disk area.

CONTEXT This function can be called from user or interrupt context.

SEE ALSO strategy(9E), buf(9S)

Writing Device Drivers

WARNINGS disksort() does no locking. Therefore, any locking is completely the responsibility of the caller.
NAME  drv_getparm – retrieve kernel state information

SYNOPSIS  
#include <sys/ddi.h>

int drv_getparm(unsigned int parm, void *value_p);

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

ARGUMENTS  
parm  The kernel parameter to be obtained. Possible values are:

   LBOLT  Read the value of lbolt. (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.)

   PPGRP  Read the process group identification number. This number determines which processes should receive a HANGUP or BREAK signal when detected by a driver.

   UPROCP  Read the process table token value.

   PPID  Read process identification number.

   PSID  Read process session identification number.

   TIME  Read time in seconds.

   UCRED  Return a pointer to the caller’s credential structure.

value_p  A pointer to the data space in which the value of the parameter is to be copied.

DESCRIPTION  

drv_getparm() function verifies that parm corresponds to a kernel parameter that may be read. If the value of parm does not correspond to a parameter or corresponds to a parameter that may not be read, -1 is returned. Otherwise, the value of the parameter is stored in the data space pointed to by value_p.


drv_getparm() does not explicitly check to see whether the device has the appropriate context when the function is called and the function does not check for correct alignment in the data space pointed to by value_p. It is the responsibility of the driver writer to use this function only when it is appropriate to do so and to correctly declare the data space needed by the driver.

RETURN VALUES  

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.

modified 30 Aug 1996  SunOS 5.6  9F-375
drv_getparm(9F)  Kernel Functions for Drivers

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
buf(9S)
Writing Device Drivers
NAME
drv_hztousec – convert clock ticks to microseconds

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
clock_t drv_hztousec(clock_t hertz);

INTERFACE
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
hertz The number of clock ticks to convert.

DESCRIPTION
drv_hztousec() converts into microseconds the time expressed by hertz, which is in system clock ticks.

The kernel variable lbolt, which is (only) readable through drv_getparm(9F), is the length of time the system has been up since boot and is expressed in clock ticks. Drivers often use the value of lbolt before and after an I/O request to measure the amount of time it took the device to process the request. drv_hztousec() can be used by the driver to convert the reading from clock ticks to a known unit of time.

RETURN VALUES
The number of microseconds equivalent to the hertz argument.
No error value is returned. If the microsecond equivalent to hertz is too large to be represented as a clock_t, then the maximum clock_t value will be returned.

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

SEE ALSO
drv_getparm(9F), drv_usectohz(9F), drv_usecwait(9F)
Writing Device Drivers
NAME       drv_priv – determine driver privilege

SYNOPSIS   #include <sys/types.h>
           #include <sys/cred.h>
           #include <sys/ddi.h>
           int drv_priv(cred_t *cr);

INTERFACE LEVEL Architecture independent level 1 (DDI/DKI).
ARGUMENTS   cr     Pointer to the user credential structure.

DESCRIPTION drv_priv() provides a general interface to the system privilege policy. It determines
whether the credentials supplied by the user credential structure pointed to by cr identify
a privileged process. This function should only be used when file access modes and spe-
cial minor device numbers are insufficient to provide protection for the requested driver
function. It is intended to replace all calls to suser() and any explicit checks for effective
user ID = 0 in driver code.

RETURN VALUES This routine returns 0 if it succeeds, EPERM if it fails.

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

SEE ALSO   Writing Device Drivers
NAME
drv_usectohz – convert microseconds to clock ticks

SYNOPSIS
#include <sys/types.h>
#include <sys/ddi.h>
clock_t drv_usectohz(clock_t microsecs);

INTERFACE
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
microsecs    The number of microseconds to convert.

DESCRIPTION
drv_usectohz() converts a length of time expressed in microseconds to a number of system clock ticks. The time arguments to timeout(9F) and delay(9F) are expressed in clock ticks.

drv_usectohz() is a portable interface for drivers to make calls to timeout(9F) and delay(9F) and remain binary compatible should the driver object file be used on a system with a different clock speed (a different number of ticks in a second).

RETURN VALUES
The value returned is the number of system clock ticks equivalent to the microsecs argument. No error value is returned. If the clock tick equivalent to microsecs is too large to be represented as a clock_t, then the maximum clock_t value will be returned.

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

SEE ALSO
delay(9F), drv_hztousec(9F), timeout(9F)

Writing Device Drivers
### NAME
drv_usecwait – busy-wait for specified interval

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

void drv_usecwait(clock_t microsecs);
```

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

### ARGUMENTS
- **microsecs**
The number of microseconds to busy-wait.

### DESCRIPTION
drv_usecwait() gives drivers a means of busy-waiting for a specified microsecond count. The amount of time spent busy-waiting may be greater than the microsecond count but will minimally be the number of microseconds specified.

delay(9F) can be used by a driver to delay for a specified number of system ticks, but it has two limitations. First, the granularity of the wait time is limited to one clock tick, which may be more time than is needed for the delay. Second, delay(9F) may only be invoked from user context and hence cannot be used at interrupt time or system initialization.

Often, drivers need to delay for only a few microseconds, waiting for a write to a device register to be picked up by the device. In this case, even in user context, delay(9F) produces too long a wait period.

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

### SEE ALSO
delay(9F), timeout(9F), untimeout(9F)

*Writing Device Drivers*

### NOTES
The driver wastes processor time by making this call since drv_usecwait() does not block but simply busy-waits. The driver should only make calls to drv_usecwait() as needed, and only for as much time as needed. drv_usecwait() does not mask out interrupts.
NAME dupb – duplicate a message block descriptor

SYNOPSIS

#include <sys/stream.h>

mblk_t *dupb(mblk_t *bp);

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

bp Pointer to the message block to be duplicated. mblk_t is an instance of the msgb(9S) structure.

DESCRIPTION dupb() creates a new mblk_t structure (see msgb(9S)) to reference the message block pointed to by bp.

Unlike copyb(9F), dupb() does not copy the information in the dblk_t structure (see datab(9S)), but creates a new mblk_t structure to point to it. The reference count in the dblk_t structure (db_ref) is incremented. The new mblk_t structure contains the same information as the original. Note that b_rptr and b_wptr are copied from the bp.

RETURN VALUES

If successful, dupb() returns a pointer to the new message block. A NULL pointer is returned if dupb() cannot allocate a new message block descriptor or if the db_ref field of the data block structure (see datab(9S)) has reached a maximum value (255).

CONTEXT dupb() can be called from user, kernel, or interrupt context.

EXAMPLES This srv(9E) (service) routine adds a header to all M_DATA messages before passing them along. dupb is used instead of copyb(9F) because the contents of the header block are not changed.
For each message on the queue, if it is a priority message, pass it along immediately (lines 10–11). Otherwise, if it is anything other than an M_DATA message (line 12), and if it can be sent along (line 13), then do so (line 14). Otherwise, put the message back on the queue and return (lines 16–17). For all M_DATA messages, first check to see if the stream is flow-controlled (line 20). If it is, put the message back on the queue and return (lines 37–38). If it is not, the header block is duplicated (line 21).

dupb() can fail either due to lack of resources or because the message block has already been duplicated 255 times. In order to handle the latter case, the example calls copyb(9F) (line 22). If copyb(9F) fails, it is due to buffer allocation failure. In this case, qbufcall(9F) is used to initiate a callback (lines 30–31) if one is not already pending (lines 26–27).

The callback function, xxxcallback(), clears the recorded qbufcall(9F) callback id and schedules the service procedure (lines 49–50). Note that the close routine, xxxclose(), must cancel any outstanding qbufcall(9F) callback requests (lines 58–59).

If dupb() or copyb(9F) succeed, link the M_DATA message to the new message block (line 34) and pass it along (line 35).

```c
1 .xxxsrv(q)
2   queue_t *q;
3 {
4     struct xx *xx = (struct xx *)q->q_ptr;
5     mblk_t *mp;
6     mblk_t *bp;
7     extern mblk_t *hdr;
8
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) {
27                        /* qbufcall pending */
28                        return;
29                    }
30            }
```
void xxxcallback(q)
{
    struct xx *xx = (struct xx *)q->q_ptr;
    xx->xx_qbufcall_id = 0;
    qenable(q);
}

xxxclose(q, cflag, crp)
{
    struct xx *xx = (struct xx *)q->q_ptr;
    ... if (xx->xx_qbufcall_id)
    qunbufcall(q, xx->xx_qbufcall_id);
    ... }

SEE ALSO srv(9E), copyb(9F), qbufcall(9F), datab(9S), msgb(9S)

Writing Device Drivers
STREAMS Programming Guide
NAME  dupmsg – duplicate a message

SYNOPSIS  
#include <sys/stream.h>

 mblk_t *dupmsg(mblk_t *mp);

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

ARGUMENTS  mp  Pointer to the message.

DESCRIPTION  dupmsg() forms a new message by copying the message block descriptors pointed to by mp and linking them. dupb(9F) is called for each message block. The data blocks themselves are not duplicated.

RETURN VALUES  If successful, dupmsg() returns a pointer to the new message block. Otherwise, it returns a NULL pointer. A return value of NULL indicates either memory depletion or the data block reference count, db_ref (see datab(9S)), has reached a limit (255). See dupb(9F).

CONTEXT  dupmsg() can be called from user, kernel, or interrupt context.

EXAMPLES  See copyb(9F) for an example using dupmsg().

SEE ALSO  copyb(9F), copymsg(9F), dupb(9F), datab(9S)
Writing Device Drivers
STREAMS Programming Guide
enableok (9F)

<table>
<thead>
<tr>
<th>NAME</th>
<th>enableok – reschedule a queue for service</th>
</tr>
</thead>
</table>
| SYNOPSIS | #include <sys/stream.h>  
#include <sys/ddi.h>  
void enableok(queue_t *q); |
| INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI). |
| ARGUMENTS | q A pointer to the queue to be rescheduled. |
| DESCRIPTION | enableok() enables queue q to be rescheduled for service. It reverses the effect of a previous call to noenable(9F) on q by turning off the QNOENB flag in the queue. |
| CONTEXT | enableok() can be called from user or interrupt context. |
| EXAMPLES | 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. |
| SEE ALSO | noenable(9F), qenable(9F) |

Writing Device Drivers  
STREAMS Programming Guide

modified 11 Apr 1991 SunOS 5.6 9F-385
esballoc – allocate a message block using a caller-supplied buffer

#include <sys/stream.h>
mblk_t *esballoc(uchar *base, size_t size, uint pri, frtn_t *fr_rtnp);

Architecture independent level 1 (DDI/DKI).

base Address of user supplied data buffer.
size Number of bytes in data buffer.
pri Priority of allocation request (to be used by allocb(9F) function, called by esballoc()).
fr_rtnp Free routine data structure.

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 user-supplied data buffer, pointed to by base, is used as the data buffer for the message.

When freeb(9F) is called to free the message, the driver’s message freeing routine (referred to through the free_rtn structure) is called, with appropriate arguments, to free the data buffer.

The free_rtn structure includes the following members:

void (*free_func)(); /* user’s freeing routine */
char *free_arg; /* arguments to free_func */

Instead of requiring a specific number of arguments, the free_arg field is defined of type char *. This way, the driver can pass a pointer to a structure if more than one argument is needed.

The method by which free_func is called is implementation-specific. The module writer must not assume that free_func will or will not be called directly from STREAMS utility routines like freeb(9F) which free a message block.

free_func must not call another module's put procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

free_func must not access any dynamically allocated data structure that might no longer exist when it runs.

On success, a pointer to the newly allocated message block is returned. On failure, NULL is returned.

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.
esbbcall – call function when buffer is available

SYNOPSIS
#include <sys/stream.h>
int esbbcall(uint pri, void (*func)(intptr_t arg), intptr_t arg);

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

ARGUMENTS
pri Priority of allocation request (to be used by allocb(9F) function, called by esbbcall())
func Function to be called when buffer becomes available.
arg Argument to func.

DESCRIPTION esbbcall(), like bufcall(9F), serves as a timeout(9F) call of indeterminate length. If esballocc(9F) is unable to allocate a message and data block header to go with its externally supplied data buffer, esbbcall() can be used to schedule the routine func, to be called with the argument arg when a buffer becomes available. func may be a routine that calls esballocc (9F) or it may be another kernel function.

RETURN VALUES
On success, a non-zero integer is returned. On failure, 0 is returned. The value returned from a successful call should be saved for possible future use with unbufcall() should it become necessary to cancel the esbbcall() request (as at driver close time).

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

SEE ALSO allocb(9F), bufcall(9F), esballocc(9F), timeout(9F), datab(9S), unbufcall(9F)
Writing Device Drivers
STREAMS Programming Guide
NAME
flushband – flush messages for a specified priority band

SYNOPSIS
#include <sys/stream.h>
void flushband(queue_t *q, unsigned char pri, int flag);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
ARGUMENTS
q Pointer to the queue.
pri Priority of messages to be flushed.
flag Valid flag values are:

- FLUSHDATA Flush only data messages (types M_DATA, M_DELAY,
  MPROTO, and M_PCPROTO).
- FLUSHALL Flush all messages.

DESCRIPTION
flushband() flushes messages associated with the priority band specified by pri. If pri is
0, only normal and high priority messages are flushed. Otherwise, messages are flushed
from the band pri according to the value of flag.

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

SEE ALSO
flushq(9F)
Writing Device Drivers
STREAMS Programming Guide

modified 11 Apr 1991 SunOS 5.6 9F-389
NAME  
flushq – remove messages from a queue

SYNOPSIS  
```c
#include <sys/stream.h>
void flushq(queue_t *q, int flag);
```

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

ARGUMENTS  
- `q`: Pointer to the queue to be flushed.
- `flag`: Valid flag values are:
  - FLUSHDATA: Flush only data messages (types M_DATA M_DELAY M_PROTO and M_PCPROTO).
  - FLUSHALL: Flush all messages.

DESCRIPTION  
flushq() frees messages and their associated data structures by calling `freemsg(9F)`. If the queue’s count falls below the low water mark and the queue was blocking an upstream service procedure, the nearest upstream service procedure is enabled.

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

EXAMPLES  
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.

```c
1 /*
2 * Module write-side put procedure.
3 */
4 xxxwput(q, mp)
5 queue_t *q;
6 mblk_t *mp;
7 {
8     switch(mp->b_datap->db_type) {
9     case M_FLUSH:
```
if (*mp->b_rptr & FLUSHR)
    flushq(RD(q), FLUSHALL);
if (*mp->b_rptr & FLUSHW)
    flushq(q, FLUSHALL);
    putnext(q, mp);
    break;
...
## NAME
freeb – free a message block

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

void freeb(mblk_t *bp);
```

## ARGUMENTS
- `bp` Pointer to the message block to be deallocated. `mblk_t` is an instance of the `msgb(9S)` structure.

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

## DESCRIPTION
`freeb()` deallocates a message block. If the reference count of the `db_ref` member of the `datab(9S)` structure is greater than 1, `freeb()` decrements the count. If `db_ref` equals 1, it deallocates the message block and the corresponding data block and buffer.

If the data buffer to be freed was allocated with the `esballoc(9F)`, the buffer may be a non-STREAMS resource. In that case, the driver must be notified that the attached data buffer needs to be freed, and run its own freeing routine. To make this process independent of the driver used in the stream, `freeb()` finds the `free_rtn(9S)` structure associated with the buffer. The `free_rtn` structure contains a pointer to the driver-dependent routine, which releases the buffer. Once this is accomplished, `freeb()` releases the STREAMS resources associated with the buffer.

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

## EXAMPLE
See `copyb(9F)` for an example of using `freeb()`.

## SEE ALSO
- `allocb(9F)`, `copyb(9F)`, `dupb(9F)`, `esballoc(9F)`, `free_rtn(9S)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
NAME  freemsg – free all message blocks in a message

SYNOPSIS  #include <sys/stream.h>
           void freemsg(mblk_t *mp);

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

ARGUMENTS  mp  Pointer to the message blocks to be deallocated. mblk_t is an instance of the msgb(9S) structure.

DESCRIPTION  freemsg() calls freeb(9F) to free all message and data blocks associated with the message pointed to by mp.

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

EXAMPLE  See copymsg(9F).

SEE ALSO  copymsg(9F), freeb(9F), msgb(9S)
           Writing Device Drivers
           STREAMS Programming Guide
NAME
freerbuf – free a raw buffer header

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

void freerbuf(struct buf *bp);

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

ARGUMENTS
bp    Pointer to a previously allocated buffer header structure.

DESCRIPTION
freerbuf() frees a raw buffer header previously allocated by getrbuf(9F). This function
does not sleep and so may be called from an interrupt routine.

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

SEE ALSO
getrbuf(9F), kmem_alloc(9F), kmem_free(9F), kmem_zalloc(9F)
**NAME**  
freezestr, unfreezestr – freeze, thaw the state of a stream

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

void freezestr(queue_t *q);
void unfreezestr(queue_t *q);
```

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

**ARGUMENTS**  
$q$  
Pointer to the message queue to freeze/unfreeze.

**DESCRIPTION**  
freezestr() freezes the state of the entire stream containing the queue pair $q$. A frozen stream blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the stream, and blocks any thread currently within the stream if it attempts to put messages onto or take messages off of any queue within the stream (with the sole exception of the caller). Threads blocked by this mechanism remain so until the stream is thawed by a call to unfreezestr().

Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as getq(9F), putq(9F), putbq(9F), etc.)

**CONTEXT**  
These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.

**SEE ALSO**  
getq(9F), insq(9F), putbq(9F), putq(9F), rmvq(9F), strqget(9F), strqset(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**  
Calling freezestr() to freeze a stream that is already frozen by the caller will result in a single-party deadlock.

The caller of unfreezestr() must be the thread who called freezestr().

There are usually better ways to accomplish things than by freezing the stream.

STREAMS utility functions such as getq(9F), putq(9F), putbq(9F), etc. may not be called by the caller of freezestr() while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.
| **NAME**  | geterror – return I/O error |
| **SYNOPSIS** | ```
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>

int geterror(struct buf *bp);
``` |
| **INTERFACE LEVEL** | Architecture independent level 1 (DDI/DKI). |
| **ARGUMENTS** | `bp` Pointer to a `buf(9S)` structure. |
| **DESCRIPTION** | `geterror()` returns the error number from the error field of the buffer header structure. |
| **RETURN VALUES** | An error number indicating the error condition of the I/O request is returned. If the I/O request completes successfully, 0 is returned. |
| **CONTEXT** | `geterror()` can be called from user or interrupt context. |
| **SEE ALSO** | `buf(9S)`

*Writing Device Drivers*
<table>
<thead>
<tr>
<th>NAME</th>
<th>getmajor – get major device number</th>
</tr>
</thead>
</table>
| SYNOPSIS   | #include <sys/types.h>  
|           | #include <sys/mkdev.h>  
|           | #include <sys/ddi.h>  
|           | major_t getmajor(dev_t dev); |
| INTERFACE  | Architecture independent level 1 (DDI/DKI). |
| LEVEL      |                                   |
| ARGUMENTS  | dev Device number.               |
| DESCRIPTION| getmajor() extracts the major number from a device number. |
| RETURN VALUES | The major number.               |
| CONTEXT    | getmajor() can be called from user or interrupt context. |
| EXAMPLE    | The following example shows both the getmajor() and getminor(9F) functions used in a debug cmn_err(9F) statement to return the major and minor numbers for the device supported by the driver. |
|            | dev_t dev;                        |
|            | #ifdef DEBUG                       |
|            | cmn_err(CE_NOTE,"Driver Started. Major# = %d, Minor# = %d", getmajor(dev), getminor(dev)); |
|            | #endif|
| SEE ALSO   | cmn_err(9F), getminor(9F), makedevice(9F)  
|            | Writing Device Drivers            |
| WARNINGS   | No validity checking is performed. If dev is invalid, an invalid number is returned. |
NAME getminor – get minor device number

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

minor_t getminor(dev_t dev);

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

ARGUMENTS
dev Device number.

DESCRIPTION getminor() extracts the minor number from a device number.

RETURN VALUES The minor number.

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

EXAMPLE See the getmajor(9F) manual page for an example of how to use getminor.

SEE ALSO getmajor(9F), makedevice(9F)
Writing Device Drivers

WARNINGS No validity checking is performed. If dev is invalid, an invalid number is returned.
NAME
get_pktiopb, free_pktiopb – allocate/free a SCSI packet in the iopb map

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *get_pktiopb(struct scsi_address *ap, caddr_t *datap, int cdblen, int statuslen, int datalen, int readflag, int (*callback)(void));

void free_pktiopb(struct scsi_pkt *pkt, caddr_t datap, int datalen);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
ap Pointer to the target’s scsi_address structure.
datap Pointer to the address of the packet, set by this function.
cdblen Number of bytes required for the SCSI command descriptor block (CDB).
statuslen Number of bytes required for the SCSI status area.
datalen Number of bytes required for the data area of the SCSI command.
readflag If non-zero, data will be transferred from the SCSI target.
callback Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC
pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION get_pktiopb() allocates a scsi_pkt structure that has a small data area allocated. It is used by some SCSI commands such as REQUEST_SENSE, which involve a small amount of data and require cache-consistent memory for proper operation. It uses ddi_iopb_alloc(9F) for allocating the data area and scsi_resalloc(9F) to allocate the packet and DMA resources.

callback indicates what get_pktiopb() should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC Wait indefinitely for resources.
Other Values callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

free_pktiopb() is used for freeing the packet and its associated resources.

RETURN VALUES get_pktiopb() returns a pointer to the newly allocated scsi_pkt or a NULL pointer.
**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.

**SEE ALSO**

`ddi_iopb_alloc(9F), scsi_alloc_consistent_buf(9F), scsi_free_consistent_buf(9F),
scsi_pktalloc(9F), scsi_resalloc(9F), scsi_pkt(9S)`

**Writing Device Drivers**

**NOTES**

`get_pktiopb()` and `free_pktiopb()` are old functions and should be replaced with `scsi_alloc_consistent_buf(9F)` and `scsi_free_consistent_buf(9F)`. `get_pktiopb()` uses scarce resources. Use it selectively.
**NAME**
getq – get the next message from a queue

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

mblk_t *getq(queue_t *q);
```

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

**ARGUMENTS**
- `q`  
  Pointer to the queue from which the message is to be retrieved.

**DESCRIPTION**
`getq()` is used by a service (`srv(9E)`) routine to retrieve its enqueued messages. A module or driver may include a service routine to process enqueued messages. Once the STREAMS scheduler calls `srv()` it must process all enqueued messages, unless prevented by flow control. `getq()` obtains the next available message from the top of the queue pointed to by `q`. It should be called in a `while` loop that is exited only when there are no more messages or flow control prevents further processing.

If an attempt was made to write to the queue while it was blocked by flow control, `getq()` back-enables (restarts) the service routine once it falls below the low water mark.

**RETURN VALUES**
If there is a message to retrieve, `getq()` returns a pointer to it. If no message is queued, `getq()` returns a NULL pointer.

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

**EXAMPLE**
See `dupb(9F)`.

**SEE ALSO**
- `srv(9E)`, `bcanput(9F)`, `canput(9F)`, `dupb(9F)`, `putbq(9F)`, `putq(9F)`, `qenable(9F)`
  - Writing Device Drivers
  - STREAMS Programming Guide
NAME  
getrbuf – get a raw buffer header

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

struct buf *getrbuf(int sleepflag);

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

ARGUMENTS  
sleepflag  Indicates whether driver should sleep for free space.

DESCRIPTION  
getrbuf() allocates the space for a buffer header to the caller. It is used in cases where a block driver is performing raw (character interface) I/O and needs to set up a buffer header that is not associated with the buffer cache.

getrbuf() calls kmem_alloc(9F) to perform the memory allocation. kmem_alloc() requires the information included in the sleepflag argument. If sleepflag is set to KM_SLEEP, the driver may sleep until the space is freed up. If sleepflag is set to KM_NOSLEEP, the driver will not sleep. In either case, a pointer to the allocated space is returned or NULL to indicate that no space was available.

RETURN VALUES  
getrbuf() returns a pointer to the allocated buffer header, or NULL if no space is available.

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

SEE ALSO  
bioint(9F), freerbuf(9F), kmem_alloc(9F), kmem_free(9F)

Writing Device Drivers
NAME hat_getkpfnum – get page frame number for kernel address

SYNOPSIS

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

u_int hat_getkpfnum(caddr_t addr);
```

INTERFACE LEVEL
Architecture independent level 2 (DKI only).

ARGUMENTS

`addr` The kernel virtual address for which the page frame number is to be returned.

DESCRIPTION

hat_getkpfnum() returns the page frame number corresponding to the kernel virtual address, `addr`.

`addr` must be a kernel virtual address which maps to device memory. `ddi_map_regs(9F)` can be used to obtain this address. For example, `ddi_map_regs(9F)` can be called in the driver’s `attach(9E)` routine. The resulting kernel virtual address can be saved by the driver (see `ddi_soft_state(9F)`) and used in `mmap(9E)`. The corresponding `ddi_unmap_regs(9F)` call can be made in the driver’s `detach(9E)` routine. Refer to `mmap(9E)` for more information.

RETURN VALUES

The page frame number corresponding to the valid virtual address `addr`. Otherwise the return value is undefined.

CONTEXT

hat_getkpfnum() can be called only from user or kernel context.

SEE ALSO

`attach(9E), detach(9E), mmap(9E), ddi_map_regs(9F), ddi_soft_state(9F), ddi_unmap_regs(9F)`

Writing Device Drivers

NOTES

For some devices, mapping device memory in the driver’s `attach(9E)` routine and unmapping device memory in the driver’s `detach(9E)` routine is a sizeable drain on system resources. This is especially true for devices with a large amount of physical address space. Refer to `mmap(9E)` for alternative methods.

modified 02 Sep 1994
SunOS 5.6
9F-403
NAME

inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port

SYNOPSIS

```c
#include <sys/ddi.h>
#include <sys/sunddi.h>
unsigned char inb(int port);
unsigned short inw(int port);
unsigned long inl(int port);
void repinsb(int port, unsigned char *addr, int count);
void repinsw(int port, unsigned short *addr, int count);
void repinsd(int port, unsigned long *addr, int count);
```

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

ARGUMENTS

- `port` A valid I/O port address.
- `addr` The address of a buffer where the values will be stored.
- `count` The number of values to be read from the I/O port.

DESCRIPTION

These routines read data of various sizes from the I/O port with the address specified by `port`.

The `inb()`, `inw()`, and `inl()` functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The `repinsb()`, `repinsw()`, and `repinsd()` functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. `count` specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold `count` values of the requested size.

RETURN VALUES

`inb()`, `inw()`, and `inl()` return the value that was read from the I/O port.

CONTEXT

These functions may be called from user or interrupt context.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86</td>
</tr>
</tbody>
</table>

SEE ALSO

eisa(4), isa(4), mca(4), attributes(5), outb(9F)

Writing Device Drivers

9F-404

SunOS 5.6

modified 1 Jan 1997
NAME

insq – insert a message into a queue

SYNOPSIS

#include <sys/stream.h>

int insq(queue_t *q, mblk_t *emp, mblk_t *nmp);

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Pointer to the queue containing message emp.</td>
</tr>
<tr>
<td>emp</td>
<td>Enqueued message before which the new message is to be inserted. mblk_t is an instance of the msgb(9S) structure.</td>
</tr>
<tr>
<td>nmp</td>
<td>Message to be inserted.</td>
</tr>
</tbody>
</table>

DESCRIPTION

insq() inserts a message into a queue. The message to be inserted, nmp, is placed in q immediately before the message emp. If emp is NULL, the new message is placed at the end of the queue. The queue class of the new message is ignored. All flow control parameters are updated. The service procedure is enabled unless QNOENB is set.

RETURN VALUES

insq() returns 1 on success, and 0 on failure.

CONTEXT

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

EXAMPLE

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.

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

static int xxxwput(queue_t *q, mblk_t *mp)
{
    union T_primitives *tp;
    mblk_t *bp;
    union T_primitives *ntp;

    switch (mp->b_datap->db_type) {
    case M_PROTO:
        tp = (union T_primitives *)mp->b_rptr;
        switch (tp->type) {
        case T_DATA_REQ:
            putq(q, mp);
        case T_EXDATA_REQ:
            // Insert expedited message before normal messages
            // Here is the implementation ofxxxwput for expedited messages
        default:
            // Handle other message types
            break;
    }
    }
case T_EXDATA_REQ:
    freezestr(q);
    for (bp = q->q_first; bp; bp = bp->b_next) {
        if (bp->b_datap->db_type == M_PROTO) {
            ntp = (union T_primitives *)bp->b_rptr;
            if (ntp->type != T_EXDATA_REQ)
                break;
        }
    }
    (void) insq(q, bp, mp);
    unfreezestr(q);
    break;

SEE ALSO: freezestr(9F), putq(9F), rmvq(9F), unfreezestr(9F), msgb(9S)

WARNINGS: If emp is non-NULL, it must point to a message on q or a system panic could result.

NOTES: The stream must be frozen using freezestr(9F) before calling insq().
NAME  IOC_CONVERT_FROM – determine if there is a need to translate M_IOCTL contents.

SYNOPSIS  

```
#include <sys/stream.h>
uint IOC_CONVERT_FROM(struct iocblk *iopc);
```

INTERFACE  

LEVEL  Solaris DDI Specific (Solaris DDI)

ARGUMENTS  

*iopc  A pointer to the M_IOCTL control structure.

DESCRIPTION  

The IOC_CONVERT_FROM() macro is used to see if the contents of the current M_IOCTL message had its origin in a different C Language Type Model.

RETURN VALUES  

IOC_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.

CONTEXT  

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

SEE ALSO  

ddi_model_convert_from(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*
NAME  kmem_alloc, kmem_zalloc, kmem_free – allocate kernel memory

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

void *kmem_alloc(size_t size, int flag);  
void *kmem_zalloc(size_t size, int flag);  
void kmem_free(void *buf, size_t size);

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

ARGUMENTS  

size  Number of bytes to allocate.

flag  Determines whether caller can sleep for memory. Possible flags are KM_SLEEP to allow sleeping until memory is available, or KM_NOSLEEP to return NULL immediately if memory is not available.

buf  Pointer to allocated memory.

DESCRIPTION  kmem_alloc() allocates size bytes of kernel memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can hold any C data structure. No greater alignment can be assumed. flag determines whether the caller can sleep for memory. KM_SLEEP allocations may sleep but are guaranteed to succeed. KM_NOSLEEP allocations are guaranteed not to sleep but may fail (return NULL) if no memory is currently available. The initial contents of memory allocated using kmem_alloc() are random garbage.

kmem_zalloc() is like kmem_alloc() but returns zero-filled memory.

kmem_free() frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.

RETURN VALUES  
If successful, kmem_alloc() and kmem_zalloc() return a pointer to the allocated memory. If KM_NOSLEEP is set and memory cannot be allocated without sleeping, kmem_alloc() and kmem_zalloc() return NULL.

CONTEXT  kmem_alloc() and kmem_zalloc() can be called from interrupt context only if the KM_NOSLEEP flag is set. They can be called from user context with any valid flag. kmem_free() can be called from user or interrupt context.

SEE ALSO  copyout(9F), freerbuf(9F), getrbuf(9F)  
Writing Device Drivers

WARNINGS  Memory allocated using kmem_alloc() is not paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.
Excessive use of kernel memory is likely to affect overall system performance. Overcommitment of kernel memory will cause the system to hang or panic.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing a null or invalid pointer, will corrupt the kernel heap and may cause the system to corrupt data and/or panic.

The initial contents of memory allocated using `kmem_alloc()` are random garbage. This random garbage may include secure kernel data. Therefore, uninitialized kernel memory should be handled carefully. For example, never `copyout(9F)` a potentially uninitialized buffer.

**NOTES**

`kmem_alloc(0, flag)` always returns `NULL`. `kmem_free(NULL, 0)` is legal.
NAME: kstat_create – create and initialize a new kstat

SYNOPSIS:
```
#include <sys/types.h>
#include <sys/kstat.h>
kstat_t *kstat_create(char *module, int instance, char *name, char *class, uchar_t type,
ulong_t ndata, uchar_t ks_flag);
```

INTERFACE LEVEL: Solaris DDI specific (Solaris DDI)

ARGUMENTS:
- **module**: The name of the provider’s module (such as "sd", "esp", ...). The "core" kernel uses the name "unix".
- **instance**: The provider’s instance number, as from ddi_get_instance(9F). Modules which don’t have a meaningful instance number should use 0.
- **name**: A pointer to a string that uniquely identifies this structure. Only KSTAT_STRLEN - 1 characters are significant.
- **class**: The general class that this kstat belongs to. The following classes are currently in use: disk, tape, net, controller, vm, kvm, hat, streams, kstat, and misc.
- **type**: The type of kstat to allocate. Valid types are:
  - KSTAT_TYPE_NAMED: named - allows more than one data record per kstat
  - KSTAT_TYPE_INTR: interrupt - only one data record per kstat
  - KSTAT_TYPE_IO: I/O - only one data record per kstat
- **ndata**: The number of type-specific data records to allocate.
- **flag**: A bit-field of various flags for this kstat. flag is some combination of:
  - KSTAT_FLAG_VIRTUAL: Tells kstat_create() not to allocate memory for the kstat data section; instead, the driver will set the ks_data field to point to the data it wishes to export. This provides a convenient way to export existing data structures.
  - KSTAT_FLAG_WRITABLE: Makes the kstat’s data section writable by root.
  - KSTAT_FLAG_PERSISTENT: Indicates that this kstat is to be persistent over time. For persistent kstats, kstat_delete(9F) simply marks the kstat as dormant; a subsequent kstat_create() reactivates the kstat. This feature is provided so that statistics are not lost across driver close/open (such as raw disk I/O on a disk with no mounted partitions.).

Note: Persistent kstats cannot be virtual, since ks_data points to garbage as soon as the driver goes away.
DESCRIPTION  

`kstat_create()` is used in conjunction with `kstat_install(9F)` to allocate and initialize a `kstat(9S)` structure. The method is generally as follows:

```c
kstat_t *ksp;

ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}
```

`kstat_create()` allocates and performs necessary system initialization of a `kstat(9S)` structure. `kstat_create()` allocates memory for the entire kstat (header plus data), initializes all header fields, initializes the data section to all zeroes, assigns a unique kstat ID (KID), and puts the kstat onto the system’s kstat chain. The returned kstat is marked invalid because the provider (caller) has not yet had a chance to initialize the data section.

After a successful call to `kstat_create()` the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type `KSTAT_TYPE_NAMED`). Virtual kstats must have the `ks_data` field set at this time. The provider may also set the `ks_update`, `ks_private`, and `ks_lock` fields if necessary.

Once the kstat is completely initialized, `kstat_install(9F)` is used to make the kstat accessible to the outside world.

RETURN VALUES  

If successful, `kstat_create()` returns a pointer to the allocated kstat. NULL is returned on failure.

CONTEXT  

`kstat_create()` can be called from user or kernel context.

SEE ALSO  

`kstat(3K), ddi_get_instance(9F), kstat_delete(9F), kstat_install(9F), kstat_named_init(9F), kstat(9S), kstat_named(9S)`

Writing Device Drivers
**NAME**

kstat_delete – remove a kstat from the system

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/kstat.h>
void kstat_delete(kstat_t *ksp);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI)

**ARGUMENTS**

- `ksp` Pointer to a currently installed kstat(9S) structure.

**DESCRIPTION**

kstat_delete() removes ksp from the kstat chain and frees all associated system resources.

**RETURN VALUES**

None.

**CONTEXT**

kstat_delete() can be called from any context.

**SEE ALSO**

kstat_create(9F), kstat_install(9F), kstat_named_init(9F), kstat(9S)

Writing Device Drivers

**NOTES**

When calling kstat_delete(), the driver must not be holding that kstat’s ks_lock. Otherwise, it may deadlock with a kstat reader.
## NAME
kstat_install – add a fully initialized kstat to the system

## SYNOPSIS
```c
#include <sys/types.h>
#include <sys/kstat.h>
void kstat_install(kstat_t *ksp);
```

## INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

## ARGUMENTS
- `ksp` Pointer to a fully initialized kstat(9S) structure.

## DESCRIPTION
`kstat_install()` is used in conjunction with `kstat_create(9F)` to allocate and initialize a kstat(9S) structure. The method is generally as follows:

```c
kstat_t *ksp;
ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}
```

After a successful call to `kstat_create()` the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type `KSTAT_TYPE_NAMED`). Virtual kstats must have the `ks_data` field set at this time. The provider may also set the `ks_update`, `ks_private`, and `ks_lock` fields if necessary.

Once the kstat is completely initialized, `kstat_install` is used to make the kstat accessible to the outside world.

## RETURN VALUES
None.

## CONTEXT
`kstat_install()` can be called from user or kernel context.

## SEE ALSO
- `kstat_create(9F)`, `kstat_delete(9F)`, `kstat_named_init(9F)`, `kstat(9S)`

*Writing Device Drivers*
NAME kstat_named_init – initialize a named kstat

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

void kstat_named_init(kstat_named_t *knp, char *name, uchar_t data_type);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI)

ARGUMENTS
knp Pointer to a kstat_named(9S) structure.
name The name of the statistic.
data_type The type of value. This indicates which field of the kstat_named(9S) structure should be used. Valid values are:

KSTAT_DATA_CHAR the "char" field.
KSTAT_DATA_LONG the "long" field.
KSTAT_DATA_ULONG the "unsigned long" field.
KSTAT_DATA_LONGLONG the "long long" field.
KSTAT_DATA_ULONGLONG the "unsigned long long" field.

DESCRIPTION kstat_named_init( ) associates a name and a type with a kstat_named(9S) structure.

RETURN VALUES None.

CONTEXT kstat_named_init( ) can be called from user or kernel context.

SEE ALSO kstat_create(9F), kstat_install(9F), kstat(9S), kstat_named(9S)

Writing Device Drivers
Kernel Functions for Drivers

NAME

kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit,
kstat_waitq_to_runq, kstat_runq_back_to_waitq — update I/O kstat statistics

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI)

ARGUMENTS

kiop Pointer to a kstat_io(9S) structure.

DESCRIPTION

A large number of I/O subsystems have at least two basic "lists" (or queues) of transactions they manage: one for transactions that have been accepted for processing but for which processing has yet to begin, and one for transactions which are actively being processed (but not done). For this reason, two cumulative time statistics are kept: wait (pre-service) time, and run (service) time.

The kstat_queue() family of functions manage these times based on the transitions between the driver wait queue and run queue.

kstat_waitq_enter() kstat_waitq_enter() should be called when a request arrives and is placed into a pre-service state (such as just prior to calling disksort(9F)).

kstat_waitq_exit() kstat_waitq_exit() should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's start routine).

kstat_runq_enter() kstat_runq_enter() is also called when a request is placed in its service state (just prior to calling the driver's start routine, but after kstat_waitq_exit()).

kstat_runq_exit() kstat_runq_exit() is used when a request is removed from its service state (just prior to calling biodone(9F)).

kstat_waitq_to_runq() kstat_waitq_to_runq() transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a kstat_waitq_exit() followed by a call to kstat_runq_enter().

kstat_runq_back_to_waitq() kstat_runq_back_to_waitq() transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an example).

modified 4 Apr 1994
SunOS 5.6
9F-415
RETURN VALUES
None.

CONTEXT
kstat_create() can be called from user or kernel context.

WARNINGS
These transitions must be protected by holding the kstat’s ks_lock, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

SEE ALSO
biodone(9F), disksort(9F), kstat_create(9F), kstat_delete(9F), kstat_named_init(9F), kstat(9S), kstat_io(9S)

Writing Device Drivers
NAME  linkb – concatenate two message blocks

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

void linkb(mblk_t *mp1, mblk_t *mp2);
```

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

ARGUMENTS  
- `mp1`  The message to which `mp2` is to be added. `mblk_t` is an instance of the `msgb(9S)` structure.
- `mp2`  The message to be added.

DESCRIPTION  linkb() creates a new message by adding `mp2` to the tail of `mp1`. The continuation pointer, `b_cont`, of `mp1` is set to point to `mp2`.

```
mp1  | b_cont  | db_base  | data buffer |
     | b_datap |          |

mp2  | b_cont (0) | db_base  | data buffer |
     | b_datap   |          |
```

```
linkb(mp1, mp2);
```

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

EXAMPLE  See `dupb(9F)` for an example of using linkb().

SEE ALSO  `dupb(9F), unlinkb(9F), msgb(9S)`

Writing Device Drivers
STREAMS Programming Guide
## NAME
makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 – make a packet for SCSI commands

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

void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd,
                int addr, int cnt);
void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd,
                   int cnt, int fixbit);
void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd,
                int addr, int cnt);
void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp, int flag, int cmd,
                int addr, int cnt);
```

## INTERFACE LEVEL ARGUMENTS
- **pkt**: Pointer to an allocated `scsi_pkt(9S)` structure.
- **devp**: Pointer to the target’s `scsi_device(9S)` structure.
- **flag**: Flags for the `pkt_flags` member.
- **cmd**: First byte of a group 0 or 1 or 5 SCSI CDB.
- **addr**: Pointer to the location of the data.
- **cnt**: Number of bytes to transfer.
- **fixbit**: Fixed bit in sequential access device commands.

## DESCRIPTION
`makecom` functions initialize a packet with the specified command descriptor block, `devp` and transport flags. The `pkt_address`, `pkt_flags`, and the command descriptor block pointed to by `pkt_cdbp` are initialized using the remaining arguments. Target drivers may use `makecom_g0()` for Group 0 commands (except for sequential access devices), or `makecom_g0_s()` for Group 0 commands for sequential access devices, or `makecom_g1()` for Group 1 commands, or `makecom_g5()` for Group 5 commands. `fixbit` is used by sequential access devices for accessing fixed block sizes and sets the the tag portion of the SCSI CDB.

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

## EXAMPLE
```c
if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEV, pflag, SCMD_WRITE_G1,
               (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEV, pflag, SCMD_WRITE,
               (int) blkno, nblk);
}
```
SEE ALSO  

scsi_device(9S), scsi_pkt(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers
<table>
<thead>
<tr>
<th>NAME</th>
<th>makedevice – make device number from major and minor numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/types.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/mkdev.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>dev_t makedevice(major_t majnum, minor_t minnum);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td>majnum           Major device number.</td>
</tr>
<tr>
<td></td>
<td>minnum           Minor device number.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>makedevice() creates a device number from a major and minor device number. makedevice() should be used to create device numbers so the driver will port easily to releases that treat device numbers differently.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The device number, containing both the major number and the minor number, is returned. No validation of the major or minor numbers is performed.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>makedevice() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>getmajor(9F), getminor(9F)</td>
</tr>
</tbody>
</table>
NAME
max – return the larger of two integers

SYNOPSIS
#include <sys/ddi.h>
int max(int int1, int int2);

INTERFACE
Architecture independent level 1 (DDI/DKI).
LEVEL
ARGUMENTS
int1 The first integer.
int2 The second integer.

DESCRIPTION
max() compares two signed integers and returns the larger of the two.

RETURN VALUES
The larger of the two numbers.

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

SEE ALSO
min(9F)
Writing Device Drivers

modified 11 Apr 1991
SunOS 5.6

9F-421
<table>
<thead>
<tr>
<th>NAME</th>
<th>min – return the lesser of two integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/ddi.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int min(int int1, int int2);</code></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td><code>int1</code> The first integer.</td>
</tr>
<tr>
<td></td>
<td><code>int2</code> The second integer.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>min() compares two signed integers and returns the lesser of the two.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>The lesser of the two integers.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>min() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>max(9F)</td>
</tr>
<tr>
<td></td>
<td><em>Writing Device Drivers</em></td>
</tr>
</tbody>
</table>
NAME  mkiocb – allocates a STREAMS ioctl block for M_IOCTL messages in the kernel.

SYNOPSIS  
#include <sys/stream.h>  
mblk_t *mkiocb ( uint command );

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
command  The ioctl command for the ioc_cmd field.

DESCRIPTION  
STREAMS modules or drivers might need to issue an ioctl to a lower module or driver. The \texttt{mkiocb()} function tries to allocate (using \texttt{allocb}(9F)) a STREAMS M_IOCTL message block (\texttt{iocblk}(9S)). Buffer allocation fails only when the system is out of memory. If no buffer is available, the \texttt{qbufcall}(9F) function can help a module recover from an allocation failure.

The \texttt{mkiocb} function returns a \texttt{mblk_t} structure which is large enough to hold any of the ioctl messages (\texttt{iocblk}(9S), \texttt{copyreq}(9S) or \texttt{copyresp}(9S)), and has the following special properties:

- \texttt{b_wptr} set to \texttt{b_rptr + sizeof (struct iocblk)}
- \texttt{b_cont} set to NULL
- \texttt{b_datap->db_type} set to M_IOCTL

The fields in the iocblk structure are initialized as follows:

- \texttt{ioc_cmd} set to the command value passed in
- \texttt{ioc_id} set to a unique identifier
- \texttt{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
- \texttt{ioc_count} set to 0
- \texttt{ioc_rval} set to 0
- \texttt{ioc_error} set to 0
- \texttt{ioc_flags} is set to IOC_NATIVE to reflect that this is native to the running kernel

RETURN VALUES  
Upon success, the \texttt{mkiocb} function returns a pointer to the allocated \texttt{mblk_t} of type M_IOCTL. On failure, it returns a null pointer.

CONTEXT  
The \texttt{mkiocb()} function can be called from user or interrupt context.

modified 13 Nov 1996  
SunOS 5.6  
9F-423
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 }
```

During the read service routine, the ioctl ID value for `M_IOCACK` or `M_IOCNACK` should equal the ioctl that was previously sent by this module before processing.

```
1 test_lrsrv(queue_t *q)
2 {
3    ...
4
5    switch (DB_TYPE(mp)) {
6        case M_IOCACK:
7        case M_IOCNACK:
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 }
```

The next example shows a `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
```
id = qbufcall(q, sizeof (union ioctypes), BPRI_HI,
    dummy_callback, 0);

/* Handle interrupts */
if (!qwait_sig(q)) {
    qunbufcall(q, id);
    return (EINTR);
}
putnext(q, mp);

SEE ALSO allocb(9F), putnext(9F), qbufcall(9F), qwait_sig(9F), copyreq(9S), copyresp(9S), iocblk(9S)

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_IOCNACK is received.
mod_install (9F)  Kernel Functions for Drivers

NAME  mod_install, mod_remove, mod_info – add, remove or query a loadable module

SYNOPSIS  
#include <sys/modctl.h>

int mod_install(struct modlinkage *modlinkage);
int mod_remove(struct modlinkage *modlinkage);
int mod_info(struct modlinkage *modlinkage, struct modinfo *modinfo);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
modlinkage  Pointer to the loadable module’s modlinkage structure which describes what type(s) of module elements are included in this loadable module.

modinfo  Pointer to the modinfo structure passed to _info(9E).

DESCRIPTION  mod_install() must be called from a module’s _init(9E) routine.
mod_remove() must be called from a module’s _fini(9E) routine.
mod_info() must be called from a module’s _info(9E) routine.

RETURN VALUES  mod_install() and mod_remove() return 0 on success and non-zero on failure.
mod_info() returns a non-zero value on success and 0 on failure.

EXAMPLES  See _init(9E) for an example describing the usage of these functions.

SEE ALSO  _fini(9E), _info(9E), _init(9E), modldrv(9S), modlinkage(9S), modlstrmod(9S)
Writing Device Drivers
msgdsize – return the number of bytes in a message

#include <sys/stream.h>

size_t msgdsize(mblk_t *mp);

Architecture independent level 1 (DDI/DKI).

Message to be evaluated.

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.

The number of data bytes in a message, expressed as an integer.

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

See bufcall(9F) for an example of using msgdsize().

bufcall(9F)

Writing Device Drivers
STREAMS Programming Guide
**NAME**

msgpullup – concatenate bytes in a message

**SYNOPSIS**

```c
#include <sys/stream.h>
mblk_t *msgpullup (mblk_t *mp, ssize_t len);
```

**INTERFACE LEVEL**

Architecture independent level 1 (DDI/DKI).

**ARGUMENTS**

- `mp` Pointer to the message whose blocks are to be concatenated.
- `len` Number of bytes to concatenate.

**DESCRIPTION**

`msgpullup()` concatenates and aligns the first `len` data bytes of the message pointed to by `mp`, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered. If `len` equals −1, all data are concatenated. If `len` bytes of the same message type cannot be found, `msgpullup()` fails and returns `NULL`.

**RETURN VALUES**

`msgpullup` returns the following values:

- Non-null  Successful completion. A pointer to the new message is returned.
- `NULL`  An error occurred.

**CONTEXT**

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

**SEE ALSO**

`srv(9E), allocb(9F), pullupmsg(9F), msgb(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**NOTES**

`msgpullup()` is a DKI-complaint replacement for the older `pullupmsg(9F)` routine. Users are strongly encouraged to use `msgpullup()` instead of `pullupmsg(9F)`.
NAME
mt-streams – STREAMS multithreading

SYNOPSIS
#include <sys/conf.h>

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

DESCRIPTION
STREAMS drivers configures the degree of concurrency using the cb_flag field in the
cb_ops structure (see cb_ops(9S)). The corresponding field for STREAMS modules is the
f_flag in the fmodsw structure.

For the purpose of restricting and controlling the concurrency in drivers/modules, we
define the concepts of inner and outer perimeters. A driver/module can be configured
either to have no perimeters, to have only an inner or an outer perimeter, or to have both
an inner and an outer perimeter. Each perimeter acts as a readers-writers lock, that is,
there can be multiple concurrent readers or a single writer. Thus, each perimeter can be
entered in two modes: shared (reader) or exclusive (writer). The mode depends on the
perimeter configuration and can be different for the different STREAMS entry points
(open(9E), close(9E), put(9E), or srv(9E)).

The concurrency for the different entry points is (unless specified otherwise) to enter with
exclusive access at the inner perimeter (if present) and shared access at the outer perime-
ter (if present).

The perimeter configuration consists of flags that define the presence and scope of the
inner perimeter, the presence of the outer perimeter (which can only have one scope),
and flags that modify the default concurrency for the different entry points.
All MT safe modules/drivers specify the D_MP flag.

Inner Perimeter Flags
The inner perimeter presence and scope are controlled by the mutually exclusive flags:

D_MTPERQ The module/driver has an inner perimeter around each queue.
D_MTQPAIR The module/driver has an inner perimeter around each
read/write pair of queues.
D_MTPERMOD The module/driver has an inner perimeter that encloses all the
module’s/driver’s queues.
None of the above The module/driver has no inner perimeter.

Outer Perimeter Flags
The outer perimeter presence is configured using:

D_MTOUTPERIM In addition to any inner perimeter, the module/driver has an outer
perimeter that encloses all the module’s/driver’s queues. This can
be combined with all the inner perimeter options except
D_MTPERMOD.

The default concurrency can be modified using:

D_MTPUTSHARED This flag modifies the default behavior when put(9E) procedure
are invoked so that the inner perimeter is entered shared instead of
exclusively.

modified 2 Mar 1993
SunOS 5.6
9F-429
D_MTOEXCL This flag modifies the default behavior when open(9E) and close(9E) procedures are invoked so the outer perimeter is entered exclusively instead of shared.

The module/driver can use qwait(9F) or qwait_sig() in the open(9E) and close(9E) procedures if it needs to wait "outside" the perimeters.

The module/driver can use qwriter(9F) to upgrade the access at the inner or outer perimeter from shared to exclusive.

The use and semantics of qprocson() and qprocsoff(9F) is independent of the inner and outer perimeters.

SEE ALSO close(9E), open(9E), put(9E), srv(9E), qprocsoff(9F), qprocson(9F), qwait(9F), qwriter(9F), cb_ops(9S)

STREAMS Programming Guide

Writing Device Drivers
NAME
mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

SYNOPSIS

#include <sys/ksynch.h>

void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
void mutex_destroy(kmutex_t *mp);
void mutex_enter(kmutex_t *mp);
void mutex_exit(kmutex_t *mp);
int mutex_owned(kmutex_t *mp);
int mutex_tryenter(kmutex_t *mp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

mp Pointer to a kernel mutex lock (kmutex_t).
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)
type Type of mutex lock.
arg Type-specific argument for initialization routine.

DESCRIPTION

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() initializes a mutex. It is an error to initialize a mutex more than once. The type argument should be set to MUTEX_DRIVER.

arg provides type-specific information for a given variant type of mutex. When mutex_init() is called for driver mutexes, if the mutex is used by the interrupt handler, the arg should be the ddi_iblock_cookie returned from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the mutex is never used inside an interrupt handler, the argument should be NULL.

mutex_enter() is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic will ensue.

mutex_owned() should only be used in ASSERT()s, and may be enforced by not being defined unless the preprocessor symbol DEBUG is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by mp.
mutex_tryenter() is very similar to mutex_enter() except that it doesn’t block when the mutex is already held. mutex_tryenter() returns non-zero when it acquired the mutex and 0 when the mutex is already held.

mutex_exit() releases a mutex and will unblock another thread if any are blocked on the mutex.

mutex_destroy() releases any resources that might have been allocated by mutex_init(). mutex_destroy() must be called before freeing the memory containing the mutex, and should be called with the mutex unheld (not owned by any thread). The caller must somehow be sure that no other thread will attempt to use the mutex.

RETURN VALUES

mutex_tryenter() returns non-zero on success and zero of failure.

mutex_owned() returns non-zero if the calling thread currently holds the mutex pointed to by mp, or when that cannot be determined, if any thread holds the mutex.

mutex_owned() returns zero otherwise.

CONTEXT

These functions can be called from user, kernel, or high-level interrupt context, except for mutex_init() and mutex_destroy(), which can be called from user or kernel context only.

EXAMPLES

Initialization

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

```c
    ddi_get_iblock_cookie(dip, 0, &iblock);
    mutex_init(&un->un_lock, NULL, MUTEX_DRIVER,
               (void *)iblock);
    ddi_add_intr(dip, 0, NULL, &dev_cookie, xxintr,
                 (caddr_t)un);
```

Also, a routine that expects to be called with a certain lock held might have the following ASSERT:

```c
    xxstart(struct xxunit *un)
    {
        ASSERT(mutex_owned(&un->un_lock));
        ...
```

SEE ALSO

condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), rwlock(9F), semaphore(9F)

Writing Device Drivers

NOTES

Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).

9F-432 SunOS 5.6 modified 7 May 1997
### NAME
nochpoll – error return function for non-pollable devices

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

int nochpoll(dev_t dev, short events, int anyyet, short *reventsp,
              struct pollhead **pollhdrp);
```

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

### ARGUMENTS
- **dev**
  Device number.
- **events**
  Event flags.
- **anyyet**
  Check current events only.
- **reventsp**
  Event flag pointer.
- **pollhdrp**
  Poll head pointer.

### DESCRIPTION
`nochpoll()` is a routine that simply returns the value `ENXIO`. It is intended to be used in the `cb_ops(9S)` structure of a device driver for devices that do not support the `poll(2)` system call.

### RETURN VALUES
`nochpoll()` returns `ENXIO`.

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

### SEE ALSO
- `poll(2)`, `chpoll(9E)`, `cb_ops(9S)`
- *Writing Device Drivers*

---

modified 11 Oct 1995

SunOS 5.6

9F-433
<table>
<thead>
<tr>
<th><strong>NAME</strong></th>
<th>nodev – error return function</th>
</tr>
</thead>
</table>
| **SYNOPSIS** | `#include <sys/conf.h>`  
`#include <sys/ddi.h>`  
`int nodev();` |
| **INTERFACE LEVEL** | Architecture independent level 1 (DDI/DKI). |
| **DESCRIPTION** | `nodev()` returns `ENXIO`. It is intended to be used in the `cb_ops(9S)` data structure of a device driver for device entry points which are not supported by the driver. That is, it is an error to attempt to call such an entry point. |
| **RETURN VALUES** | `nodev()` returns `ENXIO`. |
| **CONTEXT** | `nodev()` can be only called from user context. |
| **SEE ALSO** | `nulldev(9F), cb_ops(9S)`  
*Writing Device Drivers* |
<table>
<thead>
<tr>
<th>NAME</th>
<th>noenable – prevent a queue from being scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/stream.h&gt;</code>&lt;br&gt;<code>#include &lt;sys/ddi.h&gt;</code>&lt;br&gt;<code>void noenable(queue_t *q);</code></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td>ARGUMENTS</td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td><code>q</code></td>
</tr>
<tr>
<td>CONTEXT</td>
<td>noenable() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>enableok(9F), insq(9F), putbq(9F), putq(9F), qenable(9F)</code></td>
</tr>
</tbody>
</table>

Writing Device Drivers
STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>nulldev – zero return function</th>
</tr>
</thead>
</table>
| SYNOPSIS     | `#include <sys/conf.h>`
|              | `#include <sys/ddi.h>`
|              | `int nulldev();`           |
| INTERFACE    | Architecture independent level 1 (DDI/DKI). |
| LEVEL        | DESCRIPTION nulldev() returns 0. It is intended to be used in the cb_ops(9S) data structure of a device driver for device entry points that do nothing. |
| RETURN VALUES| nulldev() returns a 0. |
| CONTEXT      | nulldev() can be called from any context. |
| SEE ALSO     | nodev(9F), cb_ops(9S) |
|              | Writing Device Drivers |
NAME OTHERQ, otherq – get pointer to queue’s partner queue

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>
queue_t *OTHERQ(queue_t *q);

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

ARGUMENTS
q Pointer to the queue.

DESCRIPTION
The OTHERQ() function returns a pointer to the other of the two queue() structures that make up a STREAMS module or driver. If q points to the read queue the write queue will be returned, and vice versa.

RETURN VALUES
OTHERQ returns a pointer to a queue’s partner.

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

EXAMPLES
This routine sets the minimum packet size, the maximum packet size, the high water mark, and the low water mark for the read and write queues of a given module or driver. It is passed either one of the queues. This could be used if a module or driver wished to update its queue parameters dynamically.

```c
void set_q_params(q, min, max, hi, lo)
queue_t *q;
short min;
short max;
ushort hi;
ushort lo;
{
    q->q_minpsz = min;
    q->q_maxpsz = max;
    q->q_hiwat = hi;
    q->q_lowat = lo;
    OTHERQ(q)->q_minpsz = min;
    OTHERQ(q)->q_maxpsz = max;
    OTHERQ(q)->q_hiwat = hi;
    OTHERQ(q)->q_lowat = lo;
}
```

SEE ALSO
Writing Device Drivers
STREAMS Programming Guide

modified 11 Apr 1991 SunOS 5.6 9F-437
NAME  outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

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

void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);

INTERFACE LEVEL  Solaris x86 DDI specific (Solaris x86 DDI).

ARGUMENTS  

port  A valid I/O port address.
value  The data to be written to the I/O port.
addr  The address of a buffer from which the values will be fetched.
count  The number of values to be written to the I/O port.

DESCRIPTION  These routines write data of various sizes to the I/O port with the address specified by port.

The outb(), outw(), and outl() functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by value.

The repoutsb(), repoutsw(), and repoutsd() functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. count specifies the number of values to be written. addr is a pointer to a buffer from which the output values are fetched.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86</td>
</tr>
</tbody>
</table>

SEE ALSO  eisa(4), isa(4), mca(4), attributes(5), inb(9F)

Writing Device Drivers
Kernel Functions for Drivers

NAME
pci_config_get8, pci_config_get16, pci_config_get32, pci_config_get64, pci_config_put8,
pci_config_put16, pci_config_put32, pci_config_put64, pci_config_getb, pci_config_getl,
pci_config_getll, pci_config_getw, pci_config_putb, pci_config_putl, pci_config_putll,
pci_config_putw – read or write single datum of various sizes to the PCI Local Bus
Configuration space

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

uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset, uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset, uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset, uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset, uint64_t value);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
handle The data access handle returned from pci_config_setup(9F).
offset Byte offset from the beginning of the PCI Configuration space.
value Output data.

DESCRIPTION
These routines read or write a single datum of various sizes from or to the PCI Local Bus
Configuration space. The pci_config_get8(), pci_config_get16(), pci_config_get32(), and
pci_config_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively.
The pci_config_put8(), pci_config_put16(), pci_config_put32(), and pci_config_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The offset argu-
ment must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data format,
these functions translate the data from or to native host format to or from little endian
format.

pci_config_setup(9F) must be called before invoking these functions.

RETURN VALUES
pci_config_get8(), pci_config_get16(), pci_config_get32(), and pci_config_get64() return
the value read from the PCI Local Bus Configuration space.

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

modified 1 Jan 1997
SunOS 5.6
9F-439
ATTRIBUTE

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO

attributes(5), pci_config_setup(9F), pci_config_teardown(9F)

NOTES

These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

NOTES

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

<table>
<thead>
<tr>
<th>Previous Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_config_getb</td>
<td>pci_config_get8</td>
</tr>
<tr>
<td>pci_config_getw</td>
<td>pci_config_get16</td>
</tr>
<tr>
<td>pci_config_getl</td>
<td>pci_config_get32</td>
</tr>
<tr>
<td>pci_config_getll</td>
<td>pci_config_get64</td>
</tr>
<tr>
<td>pci_config_putb</td>
<td>pci_config_put8</td>
</tr>
<tr>
<td>pci_config_putw</td>
<td>pci_config_put16</td>
</tr>
<tr>
<td>pci_config_putl</td>
<td>pci_config_put32</td>
</tr>
<tr>
<td>pci_config_putll</td>
<td>pci_config_put64</td>
</tr>
</tbody>
</table>
NAME pci_config_setup, pci_config_teardown – setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space

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

int pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle);
void pci_config_teardown(ddi_acc_handle_t *handle);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to the device's dev_info structure.

handle Pointer to a data access handle.

DESCRIPTION
pci_config_setup() sets up the necessary resources for enabling subsequent data accesses to the PCI Local Bus Configuration space. pci_config_teardown() reclaims and removes those resources represented by the data access handle returned from pci_config_setup().

RETURN VALUES
pci_config_setup() returns:

  DDI_SUCCESS Successfully setup the resources.
  DDI_FAILURE Unable to allocate resources for setup.

CONTEXT
pci_config_setup() must be called from user or kernel context. pci_config_teardown() can be called from any context.

NOTES
These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>PCI Local Bus</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5)
IEEE 1275 PCI Bus Binding

modified 1 Jan 1997 SunOS 5.6 9F-441
**NAME**
physio, minphys – perform physical I/O

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

int physio(int (*strat)(struct buf *), struct buf *bp, dev_t dev, int rw,
void (*mincnt)(struct buf *), struct uio *uio);

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

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

**ARGUMENTS**

**physio()**
- `strat` Pointer to device strategy routine.
- `bp` Pointer to a `buf(9S)` structure describing the transfer. If `bp` is set to NULL then `physio()` allocates one which is automatically released upon completion.
- `dev` The device number.
- `rw` Read/write flag. This is either `B_READ` when reading from the device, or `B_WRITE` when writing to the device.
- `mincnt` Routine which bounds the maximum transfer unit size.
- `uio` Pointer to the `uio` structure which describes the user I/O request.

**minphys()**
- `bp` Pointer to a `buf` structure.

**DESCRIPTION**

`physio()` performs unbuffered I/O operations between the device `dev` and the address space described in the `uio` structure.

Prior to the start of the transfer `physio()` verifies the requested operation is valid by checking the protection of the address space specified in the `uio` structure. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, `strat()`, is then called one or more times to perform the physical I/O operations. `physio()` uses `biowait(9F)` to block until `strat()` has completed each transfer. Upon completion, or detection of an error, `physio()` unlocks the pages and returns the error status.

`physio()` uses `mincnt()` to bound the maximum transfer unit size to the system, or device, maximum length. `minphys()` is the system `mincnt()` routine for use with `physio()` operations. Drivers which do not provide their own local `mincnt()` routines should call `physio()` with `minphys()`.

`minphys()` limits the value of `bp->b_bcount` to a sensible default for the capabilities of the system. Drivers that provide their own `mincnt()` routine should also call `minphys()` to make sure they do not exceed the system limit.
RETURN VALUES  
physio() returns:
0  on success.
non-zero  on failure.

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

SEE ALSO  
strategy(9E), biodone(9F), biowait(9F), buf(9S), uio(9S)
Writing Device Drivers

WARNINGS  
Since physio() calls biowait() to block until each buf transfer is complete, it is the drivers responsibility to call biodone(9F) when the transfer is complete, or physio() will block forever.
### NAME
pm_busy_component, pm_idle_component – control device components’ availability for power management

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

int pm_busy_component(dev_info_t *dip, int component);
int pm_idle_component(dev_info_t *dip, int component);
```

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

### ARGUMENTS
**pm_busy_component( )**
- **dip** Pointer to the device’s `dev_info` structure.
- **component** The number of the component to be power-managed.

**pm_idle_component( )**
- **dip** Pointer to the device’s `dev_info` structure.
- **component** The number of the component to be power-managed.

### DESCRIPTION
The `pm_busy_component()` function sets `component` of `dip` to be busy. Calls to `pm_busy_component()` are stacked, requiring a corresponding number of calls to `pm_idle_component()` to make the component idle again. When a device is busy it will not be power-managed by the system.

The `pm_idle_component()` function marks `component` idle, recording the time that `component` went idle. This function must be called once for each call to `pm_busy_component()`. A component which is idle is available to be power-managed by the system. The `pm_idle_component()` function has no effect if the component is already idle, except to update the system’s notion of when the device went idle.

### RETURN VALUES
The `pm_busy_component()` and `pm_idle_component()` functions return:
- **DDI_SUCCESS** Successfully set the indicated component busy or idle.
- **DDI_FAILURE** Invalid component number `component` or the device has no components.

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

### SEE ALSO
- `power.conf(4)`, `pm(7D)`, `pm(9E)`, `pm_create_components(9F)`, `pm_destroy_components(9F)`
- *Writing Device Drivers*
NAME
pm_create_components, pm_destroy_components – create or destroy power-manageable components

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
int pm_create_components(dev_info_t *dip, int components);
void pm_destroy_components(dev_info_t *dip);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
pm_create_components()
  dip Pointer to the device’s dev_info structure.
  components The number of components to create.

pm_destroy_components()
  dip Pointer to the device’s dev_info structure.

DESCRIPTION
The pm_create_components() function creates power-manageable components for a device. It should be called from the driver’s attach(9E) entry point if the device has power-manageable components. The correspondence of components to parts of the physical device controlled by the driver are the responsibility of the driver. Component 0 must represent the entire device. Components 1-n are driver-defined.

The pm_destroy_components() function removes all components from the device. It should be called from the driver’s detach(9E) entry point.

RETURN VALUES
The pm_create_components() function returns:
DDI_SUCCESS Components are successfully created.
DDI_FAILURE The device already has components.

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

SEE ALSO
power.conf(4), pm(7D), attach(9E), detach(9E), pm(9E), pm_busy_component(9F), pm_idle_component(9F)
Writing Device Drivers

modified 28 Oct 1996
SunOS 5.6
9F-445
NAME pm_get_normal_power, pm_set_normal_power – get or set a device component’s normal power level

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

int pm_get_normal_power(dev_info_t *dip, int component);
void pm_set_normal_power(dev_info_t *dip, int component, int level);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS
pm_get_normal_power()
   dip Pointer to the device’s dev_info structure.
   component Number of component to get normal power level of.

pm_set_normal_power()
   dip Pointer to the device’s dev_info structure.
   component Number of component to set normal power level for.
   level Power level to become the component’s new normal power level.

DESCRIPTION The pm_get_normal_power() function returns the normal power level of component of the device dip.

The pm_set_normal_power() function sets the normal power level of component of the device dip to level.

When a device has been power-managed by pm(7D) 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”, or a power level in the range 1-15, which are reserved, the interpretation of the meaning of the power level is entirely up to the driver.

RETURN VALUES The pm_get_normal_power() function returns:

   level The normal power level of the specified component (a positive integer).

   DDI_FAILURE Invalid component number component or the device has no components.

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

SEE ALSO power.conf(4), pm(7D), pm(9E), power(9E), pm_busy_component(9F), pm_create_components(9F), pm_destroy_components(9F), pm_idle_component(9F)

Writing Device Drivers
### pollwakeup

**NAME**
pollwakeup – inform a process that an event has occurred

**SYNOPSIS**
```c
#include <sys/poll.h>
void pollwakeup(struct pollhead *php, short event);
```

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

**ARGUMENTS**
- `php` Pointer to a `pollhead` structure.
- `event` Event to notify the process about.

**DESCRIPTION**
pollwakeup() wakes a process waiting on the occurrence of an event. It should be called from a driver for each occurrence of an event. The `pollhead` structure will usually be associated with the driver’s private data structure associated with the particular minor device where the event has occurred. See `chpoll(9E)` and `poll(2)` for more detail.

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

**SEE ALSO**
- `poll(2)`, `chpoll(9E)`
- *Writing Device Drivers*

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

---

modified 11 Apr 1991

SunOS 5.6

9F-447
proc_signal (9F)

NAME
proc_signal, proc_ref, proc_unref – send a signal to a process

SYNOPSIS
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/signal.h>
void *proc_ref(void);
void proc_unref(void *pref);
int proc_signal(void *pref, int sig);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL
ARGUMENTS
pref A handle for the process to be signalled.
sig Signal number to be sent to the process.

DESCRIPTION
This set of routines allows a driver to send a signal to a process. The routine proc_ref() is
used to retrieve an unambiguous reference to the process for signalling purposes. The
return value can be used as a unique handle on the process, even if the process dies.
Because system resources are committed to a process reference, proc_unref() should be
used to remove it as soon as it is no longer needed.

proc_signal() is used to send signal sig to the referenced process. The following set of
signals may be sent to a process from a driver:

SIGHUP The device has been disconnected
SIGINT The interrupt character has been received
SIGQUIT The quit character has been received
SIGPOLL A pollable event has occurred.
SIGKILL Kill the process (cannot be caught or ignored)
SIGWINCH Window size change.
SIGURG Urgent data are available.

See signal(5) for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, proc_signal() returns an error
code; the caller should remove the reference on the process by calling proc_unref().

The driver writer must ensure that for each call made to proc_ref(), there is exactly one
corresponding call to proc_unref().

RETURN VALUES
proc_ref()
pref An opaque handle used to refer to the current process.
proc_signal()
0 The process existed before the signal was sent.
-1 The process no longer exists; no signal was sent.
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(5), putnextctl1(9F)
Writing Device Drivers
<table>
<thead>
<tr>
<th>NAME</th>
<th>ptob – convert size in pages to size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>unsigned long ptob(unsigned long numpages);</td>
</tr>
<tr>
<td>INTERFACE LEVEL</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td>numpages  Size in number of pages to convert to size in bytes.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This function returns the number of bytes that are contained in the specified number of pages. For example, if the page size is 2048, then ptob(2) returns 4096. ptob(0) returns 0.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>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.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>ptob() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>btop(9F), btopr(9F), ddi_ptob(9F)</td>
</tr>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
</tbody>
</table>
NAME  pullupmsg – concatenate bytes in a message

SYNOPSIS  
#include <sys/stream.h>

int pullupmsg(mblk_t *mp, ssize_t len);

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

ARGUMENTS  
m  Pointer to the message whose blocks are to be concatenated. mblk_t is an instance of the msgb(9S) structure.
len  Number of bytes to concatenate.

DESCRIPTION  pullupmsg() tries to combine multiple data blocks into a single block. pullupmsg() concatenates and aligns the first len data bytes of the message pointed to by mp. If len equals -1, all data are concatenated. If len bytes of the same message type cannot be found, pullupmsg() fails and returns 0.

RETURN VALUES  On success, 1 is returned; on failure, 0 is returned.

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

EXAMPLES  This is a driver write srv(9E) (service) routine for a device that does not support scatter/gather DMA. For all M_DATA messages, the data will be transferred to the device with DMA.

First, try to pull up the message into one message block with the pullupmsg() function (line 12). If successful, the transfer can be accomplished in one DMA job. Otherwise, it must be done one message block at a time (lines 19–22). After the data has been transferred to the device, free the message and continue processing messages on the queue.

```c
1 xxxwsrv(q)
2 queue_t *q;
3 {
4   mblk_t *mp;
5   mblk_t *tmp;
6   caddr_t dma_addr;
7   ssize_t dma_len;
8   
9   while ((mp = getq(q)) != NULL) {
10      switch (mp->b_datap->db_type) {
11         case M_DATA:
12         if (pullupmsg(mp, -1)) {
13            dma_addr = vtop(mp->b_rptr);
14            dma_len = mp->b_wptr - mp->b_rptr;
15            xxx_do_dma(dma_addr, dma_len);
16            freemsg(mp);
```

modified 11 Nov 1996  SunOS 5.6  9F-451
break;
}
for (tmp = mp; tmp; tmp = tmp->b_cont) {
    dma_addr = vtop(tmp->b_rptr);
    dma_len = tmp->b_wptr - tmp->b_rptr;
    xxx_do_dma(dma_addr, dma_len);
}
freemsg(mp);
break;

SEE ALSO

srv(9E), allocb(9F), msgpullup(9F), msgb(9S)

Writing Device Drivers
STREAMS Programming Guide

NOTES

pullupmsg() is not included in the DKI and will be removed from the system in a future release. Device driver writers are strongly encouraged to use msgpullup(9F) instead of pullupmsg().
NAME
put – call a STREAMS put procedure

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

void put(queue_t *q, mblk_t *mp);

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

ARGUMENTS
q Pointer to a STREAMS queue.
mp Pointer to message block being passed into queue.

DESCRIPTION
put calls the put procedure (put(9E) entry point) for the STREAMS queue specified by q,
passing it the message block referred to by mp. It is typically used by a driver or module
to call its own put procedure.

CONTEXT
put can be called from a STREAMS module or driver put or service routine, or from an
associated interrupt handler, timeout, bufcall, or esballoc call-back. In the latter cases the
calling code must guarantee the validity of the q argument.

Since put may cause re-entry of the module (as it is intended to do), mutexes or other
locks should not be held across calls to it, due to the risk of single-party deadlock.
put(9E), putnext(9F), putctl(9F), qreply(9F), etc). This function is provided as a DDI/DKI
conforming replacement for a direct call to a put procedure.

SEE ALSO
put(9E), freezestr(9F), putctl(9F), putctl1(9F), putnext(9F), putnextctl(9F),
putnextctl1(9F), qreply(9F)
Writing Device Drivers
STREAMS Programming Guide

NOTES
The caller cannot have the stream frozen (see freezestr(9F)) when calling this function.
DDI/DKI conforming modules and drivers are no longer permitted to call put pro-
cedures directly, but must call through the appropriate STREAMS utility function (e.g.
put(9E), putnext(9F), putctl(9F), qreply(9F), etc). This function is provided as a
DDI/DKI conforming replacement for a direct call to a put procedure.
NAME  putbq – place a message at the head of a queue

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

int putbq(queue_t *q, mblk_t *bp);
```

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

ARGUMENTS  
- `q`  Pointer to the queue.
- `bp`  Pointer to the message block.

DESCRIPTION  putbq() places a message at the beginning of the appropriate section of the message queue. There are always sections for high priority and ordinary messages. If other priority bands are used, each will have its own section of the queue, in priority band order, after high priority messages and before ordinary messages. putbq() can be used for ordinary, priority band, and high priority messages. However, unless precautions are taken, using putbq() with a high priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.

This function is usually called when bcanput(9F) or canput(9F) determines that the message cannot be passed on to the next stream component. The flow control parameters are updated to reflect the change in the queue’s status. If QNOENB is not set, the service routine is enabled.

RETURN VALUES  putbq() returns 1 on success and 0 on failure.

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

EXAMPLE  See the bufcall(9F) function page for an example of putbq().

SEE ALSO  bcanput(9F), bufcall(9F), canput(9F), getq(9F), putq(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME
putctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS
#include <sys/stream.h>

int putctl1(queue_t *q, int type, int p);

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

ARGUMENTS
q Queue to which the message is to be sent.
type Type of message.
p One-byte parameter.

DESCRIPTION
putctl1(), like putctl(9F), tests the type argument to make sure a data type has not been specified, and attempts to allocate a message block. The p parameter can be used, for example, to specify how long the delay will be when an M_DELAY message is being sent. putctl1() fails if type is M_DATA, MPROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful, putctl1() calls the put(9E) routine of the queue pointed to by q with the newly allocated and initialized message.

RETURN VALUES
On success, 1 is returned. 0 is returned if type is a data type, or if a message block cannot be allocated.

CONTEXT
putctl1() can be called from user or interrupt context.

EXAMPLE
See the putctl(9F) function page for an example of putctl1().

SEE ALSO
put(9E), allocb(9F), datamsg(9F), putctl(9F), putnextctl1(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME
putctl – send a control message to a queue

SYNOPSIS
#include <sys/stream.h>

int putctl(queue_t *q, int type);

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

ARGUMENTS
q Queue to which the message is to be sent.
type Message type (must be control, not data type).

DESCRIPTION
putctl() tests the type argument to make sure a data type has not been specified, and then
attempts to allocate a message block. putctl fails if type is M_DATA, M_PROTO, or
M_PCPROTO, or if a message block cannot be allocated. If successful, putctl() calls the
put(9E) routine of the queue pointed to by q with the newly allocated and initialized mes-
sages.

RETURN VALUES
On success, 1 is returned. If type is a data type, or if a message block cannot be allocated,
0 is returned.

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

EXAMPLE
The send_ctl routine is used to pass control messages downstream. M_BREAK messages
are handled with putctl() (line 11). putctl1(9F) (line 16) is used for M_DELAY messages,
so that parm can be used to specify the length of the delay. In either case, if a message
block cannot be allocated a variable recording the number of allocation failures is incre-
mented (lines 12, 17). If an invalid message type is detected, cmn_err(9F) panics the sys-
tem (line 21).

1 void
2 send_ctl(wrq, type, parm)
3 queue_t *wrq;
4 unchar type;
5 unchar parm;
6 {
7     extern int num_alloc_fail;
8
9     switch (type) {
10        case M_BREAK:
11            if (!putctl(wrq->q_next, M_BREAK))
12                num_alloc_fail++;
13            break;
14
15        case M_DELAY:
16            if (!putctl1(wrq->q_next, M_DELAY, parm))
17                num_alloc_fail++;
18       break;
19
20   default:
21       cmn_err(CE_PANIC, "send_ctl: bad message type passed");
22       break;
23   }
24 }

SEE ALSO  put(9E), cmn_err(9F), datamsg(9F), putctl1(9F), putnextctl(9F)

Writing Device Drivers
STREAMS Programming Guide
<table>
<thead>
<tr>
<th>NAME</th>
<th>putnext – send a message to the next queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>#include &lt;sys/stream.h&gt;</td>
</tr>
<tr>
<td></td>
<td>#include &lt;sys/ddi.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int putnext(queue_t *q, mblk_t *mp);</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Architecture independent level 1 (DDI/DKI).</td>
</tr>
<tr>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>Pointer to the queue from which the message mp will be sent.</td>
</tr>
<tr>
<td>mp</td>
<td>Message to be passed.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>putnext() is used to pass a message to the put(9E) routine of the next queue in the stream.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td>None.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>putnext() can be called from user or interrupt context.</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>See allocb(9F) for an example of using putnext().</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>put(9E), allocb(9F)</td>
</tr>
<tr>
<td></td>
<td>Writing Device Drivers</td>
</tr>
<tr>
<td></td>
<td>STREAMS Programming Guide</td>
</tr>
</tbody>
</table>
NAME  putnextctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS  
```c
#include <sys/stream.h>
int putnextctl1(queue_t *q, int type, int p);
```

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

ARGUMENTS  
- `q`: Queue to which the message is to be sent.
- `type`: Type of message.
- `p`: One-byte parameter.

DESCRIPTION  putnextctl1(), like putctl1(9F), tests the type argument to make sure a data type has not been specified, and attempts to allocate a message block. The `p` parameter can be used, for example, to specify how long the delay will be when an M_DELAY message is being sent. putnextctl1() fails if `type` is M_DATA, M_PROTO, or M_PCPROTO, or if a message block cannot be allocated. If successful, putnextctl1() calls the put(9E) routine of the queue pointed to by `q` with the newly allocated and initialized message.

A call to putnextctl1(q, type, p) is an atomic equivalent of putctl1(q->q_next, type, p). The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing `q` through its q_next field and then invoking putctl1(9F) proceeds without interference from other threads.

putnextctl1() should always be used in preference to putctl1(9F).

RETURN VALUES  On success, 1 is returned. 0 is returned if type is a data type, or if a message block cannot be allocated.

CONTEXT  putnextctl1() can be called from user or interrupt context.

EXAMPLE  See the putnextctl(9F) function page for an example of putnextctl1().

SEE ALSO  put(9E), allocb(9F), datamsg(9F), putctl1(9F), putnextctl(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME  putnextctl – send a control message to a queue

SYNOPSIS  

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

int putnextctl(queue_t *q, int type);
```

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

ARGUMENTS  

- `q` Queue to which the message is to be sent.
- `type` Message type (must be control, not data type).

DESCRIPTION  putnextctl() tests the `type` argument to make sure a data type has not been specified, and then attempts to allocate a message block. putnextctl() fails if `type` is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or if a message block cannot be allocated. If successful, putnextctl() calls the `put()` routine of the queue pointed to by `q` with the newly allocated and initialized messages.

A call to `putnextctl(q, type)` is an atomic equivalent of `putctl(q -> q_next, type)`. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing `q` through its `q_next` field and then invoking `putctl(9F)` proceeds without interference from other threads.

putnextctl() should always be used in preference to `putctl(9F)`.

RETURN VALUES  On success, 1 is returned. If `type` is a data type, or if a message block cannot be allocated, 0 is returned.

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

EXAMPLE  

The `send_ctl` routine is used to pass control messages downstream. `M_BREAK` messages are handled with putnextctl() (line 8). putnextctl(9F) (line 13) is used for `M_DELAY` messages, so that `parm` can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 9, 14). If an invalid message type is detected, `cmn_err(9F)` panics the system (line 18).

```c
1  void
2  send_ctl(queue_t *wrq, u_char type, u_char parm)
3  {
4      extern int num_alloc_fail;
5
6      switch (type) {
7          case M_BREAK:
8              if (!putnextctl(wrq, M_BREAK))
9                  num_alloc_fail++;
10              break;
11          case M_DELAY:
12          ...
```

9F-460  SunOS 5.6  modified 29 Mar 1993
if (!putnextctl1(wrq, M_DELAY, parm))
    num_alloc_fail++;
break;

default:
cmn_err(CE_PANIC, "send_ctl: bad message type passed");
break;

SEE ALSO put(9E), cmn_err(9F), datamsg(9F), putctl(9F), putnextctl(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME

putq – put a message on a queue

SYNOPSIS

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

int putq(queue_t *q, mblk_t *bp);
```

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

- **q**: Pointer to the queue to which the message is to be added.
- **bp**: Message to be put on the queue.

DESCRIPTION

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.

RETURN VALUES

putq() returns 1 on success and 0 on failure.

CONTEXT

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

EXAMPLE

See the datamsg(9F) function page for an example of putq().

SEE ALSO

datamsg(9F), putbq(9F), qenable(9F), rmvq(9F)

Writing Device Drivers

STREAMS Programming Guide
NAME
qbufcall – call a function when a buffer becomes available

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

int qbufcall (queue_t *q, size_t size, uint pri, void (*func) (intptr_t arg), intptr_t arg);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

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

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

The qbufcall() function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See mt-streams(9F).) qbufcall() schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to qbufcall(). All outstanding bufcalls should be cancelled before the close of a driver or module returns.

qprocson(9F) must be called before calling either qbufcall() or qtimeout(9F).

RETURN VALUES
If successful, qbufcall() returns a qbufcall id that can be used in a call to qunbufcall(9F) to cancel the request. If the qbufcall() scheduling fails, func is never called and 0 is returned.

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

SEE ALSO
allocb(9F), mt-streams(9F), qprocson(9F), qtimeout(9F), qunbufcall(9F), quntimeout(9F)
Writing Device Drivers
STREAMS Programming Guide

WARNINGS
Even when func is called by qbufcall(), allocb(9F) can fail if another module or driver had allocated the memory before func was able to call allocb(9F).
NAME          qenable – enable a queue

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

void qenable(queue_t *q);

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

ARGUMENTS     
q               Pointer to the queue to be enabled.

DESCRIPTION   qenable() adds the queue pointed to by q to the list of queues whose service routines are ready to be called by the STREAMS scheduler.

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

EXAMPLE       See the dupb(9F) function page for an example of the qenable().

SEE ALSO      dupb(9F)

Writing Device Drivers
STREAMS Programming Guide
Kernel Functions for Drivers

qprocson (9F)

NAME
qprocson, qprocsoff – enable, disable put and service routines

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

void qprocson(queue_t *q);
void qprocsoff(queue_t *q);

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

ARGUMENTS
q Pointer to the RD side of a STREAMS queue pair.

DESCRIPTION
qprocson() enables the put and service routines of the driver or module whose read
queue is pointed to by q. Threads cannot enter the module instance through the put and
service routines while they are disabled.

qprocson() must be called by the open routine of a driver or module before returning,
and after any initialization necessary for the proper functioning of the put and service
routines.

qprocson() must be called before calling qbufcall(9F), qtimeout(9F), qwait(9F), or
qwait_sig(9F),

qprocsoff() must be called by the close routine of a driver or module before returning,
and before deallocating any resources necessary for the proper functioning of the put and
service routines. It also removes the queue’s service routines from the service queue, and
blocks until any pending service processing completes.

The module or driver instance is guaranteed to be single-threaded before qprocson() is
called and after qprocsoff() is called, except for threads executing asynchronous events
such as interrupt handlers and callbacks, which must be handled separately.

CONTEXT
These routines can be called from user or interrupt context.

SEE ALSO
close(9E), open(9E), put(9E), srv(9E), qbufcall(9F), qtimeout(9F), qwait(9F),
qwait_sig(9F)
Writing Device Drivers
STREAMS Programming Guide

NOTES
The caller may not have the STREAM frozen during either of these calls.

modified 11 Nov 1992 SunOS 5.6 9F-465
NAME  qreply – send a message on a stream in the reverse direction

SYNOPSIS  
#include <sys/stream.h>

void qreply(queue_t *q, mblk_t *mp);

INTERFACE LEVEL  Architecture independent level 1 (DDI/DKI).

ARGUMENTS  
q  Pointer to the queue.

mp  Pointer to the message to be sent in the opposite direction.

DESCRIPTION  qreply() sends messages in the reverse direction of normal flow. That is, qreply(q, mp) is equivalent to putnext(OTHERQ(q), mp).

CONTEXT  qreply() can be called from user or interrupt context.

EXAMPLE  This example depicts the canonical flushing code for STREAMS drivers. Assume that the driver has service procedures (see srv(9E)), so that there may be messages on its queues. Its write-side put procedure (see put(9E)) 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). 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.

```c
1   xxxwput(q, mp)
2   queue_t *q;
3   mblk_t *mp;
4 {
5     switch(mp->b_datap->db_type) {
6         case M_FLUSH:
7             if (*mp->b_rptr & FLUSHW) {
8                 flushq(q, FLUSHALL);
9                     *mp->b_rptr &= ~FLUSHW;
10             }
11             if (*mp->b_rptr & FLUSHR) {
12                 flushq(RD(q), FLUSHALL);
13                 qreply(q, mp);
14             } else {
15                 freemsg(mp);
16             } break;
17         }
18     }
19 }
```
SEE ALSO put(9E), srv(9E), flushq(9F), OTHERQ(9F), putnext(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME    qsize – find the number of messages on a queue

SYNOPSIS   #include <sys/stream.h>
           int qsize(queue_t *q);

INTERFACE LEVEL
ARGUMENTS  q       Queue to be evaluated.

DESCRIPTION qsize() evaluates the queue q and returns the number of messages it contains.

RETURN VALUES If there are no message on the queue, qsize() returns 0. Otherwise, it returns the integer representing the number of messages on the queue.

CONTEXT    qsize() can be called from user or interrupt context.

SEE ALSO   Writing Device Drivers
            STREAMS Programming Guide
NAME
qtimeout – execute a function after a specified length of time

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>

int qtimeout ( queue_t *q, void (*ftn) (intptr_t), intptr_t arg, clock_t ticks );

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
q Pointer to STREAMS queue structure.
ftn Kernel function to invoke when the time increment expires.
arg Argument to the function.
ticks Number of clock ticks to wait before the function is called.

DESCRIPTION
The qtimeout() function schedules the specified function ftn to be called after a specified time interval. ftn is called with arg as a parameter. Control is immediately returned to the caller. This is useful when an event is known to occur within a specific time frame, or when you want to wait for I/O processes when an interrupt is not available or might cause problems. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The qtimeout() function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See mt-streams(9F).) qtimeout() schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to qtimeout(). All outstanding timeouts should be cancelled before a driver closes or module returns. qprocson(9F) must be called before calling qtimeout().

RETURN VALUES
Under normal conditions, an integer timeout identifier is returned.
The qtimeout() function returns an identifier that may be passed to the quntimeout(9F) function to cancel a pending request. Note: No value is returned from the called function.

CONTEXT
qtimeout() can be called from user or interrupt context.

SEE ALSO
mt-streams(9F), qbufcall(9F), qprocson(9F), qunbufcall(9F), quntimeout(9F)
Writing Device Drivers
STREAMS Programming Guide
**NAME**

qunbufcall – cancel a pending qbufcall request

**SYNOPSIS**

```c
#include <sys/stream.h>
#include <sys/ddi.h>

void qunbufcall(queue_t *q, int id);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

- `q` Pointer to STREAMS queue_t structure.
- `id` Identifier returned from qbufcall(9F)

**DESCRIPTION**

qunbufcall cancels a pending qbufcall() request. The argument `id` is a non-zero identifier of the request to be cancelled. `id` is returned from the qbufcall() function used to issue the cancel request.

The qunbufcall() function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See mt-streams(9F).) qunbufcall() returns when the bufcall has been cancelled or finished executing. The bufcall will be cancelled even if it is blocked at the perimeters associated with the queue. All outstanding bufcalls should be cancelled before the driver closes or module returns.

**CONTEXT**

qunbufcall() can be called from user or interrupt context.

**SEE ALSO**

mt-streams(9F), qbufcall(9F), qtimeout(9F), quntimeout(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*
### NAME
quntimeout – cancel previous qtimeout function call

### SYNOPSIS
```
#include <sys/stream.h>
#include <sys/ddi.h>

int quntimeout(queue_t *q, int id);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### ARGUMENTS
- `q` Pointer to a STREAMS queue structure.
- `id` Identification value generated by a previous `qtimeout(9F)` function call.

### DESCRIPTION
`quntimeout()` cancels a pending `qtimeout(9F)` request. The `quntimeout()` function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See `mt-streams(9F)`.) `quntimeout()` returns when the timeout has been cancelled or finished executing. The timeout will be cancelled even if it is blocked at the perimeters associated with the queue. `quntimeout()` should be executed for all outstanding timeouts before a driver or module close returns.

### RETURN VALUES
`quntimeout()` returns -1 if the `id` is not found. Otherwise, `quntimeout()` returns a zero or positive value.

### CONTEXT
`quntimeout()` can be called from user or interrupt context.

### SEE ALSO
- `mt-streams(9F)`, `qbufcall(9F)`, `qtimeout(9F)`, `qunbufcall(9F)`

*Writing Device Drivers*
*STREAMS Programming Guide*
NAME  qwait, qwait_sig – STREAMS wait routines

SYNOPSIS  #include <sys/stream.h>
#include <sys/ddi.h>
void qwait( queue_t ∗q);
int qwait_sig( queue_t ∗q);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  qp  Pointer to the queue that is being opened or closed.

DESCRIPTION  qwait() and qwait_sig() are used to wait for a message to arrive to the put(9E) or srv(9E) procedures. qwait() and qwait_sig() can also be used to wait for qbufcall(9F) or qtimeout(9F) callback procedures to execute. These routines can be used in the open(9E) and close(9E) procedures in a STREAMS driver or module. qwait() and qwait_sig() atomically exit the inner and outer perimeters associated with the queue, and wait for a thread to leave the module’s put(9E), srv(9E), or qbufcall(9F) / qtimeout(9F) callback procedures. Upon return they re-enter the inner and outer perimeters.

This can be viewed as there being an implicit wakeup when a thread leaves a put(9E) or srv(9E) procedure or after a qtimeout(9F) or qbufcall(9F) callback procedure has been run in the same perimeter.

qprocson(9F) must be called before calling qwait() or qwait_sig().

qwait() is not interrupted by a signal, whereas qwait_sig() is interrupted by a signal.

qwait() normally returns non-zero, and returns zero when the waiting was interrupted by a signal.

qwait() and qwait_sig() are similar to cv_wait() and cv_wait_sig() (see condvar(9F)), except that the mutex is replaced by the inner and outer perimeters and the signalling is implicit when a thread leaves the inner perimeter.

RETURN VALUES 0  For qwait_sig(), indicates that the condition was not necessarily signaled and the function returned because a signal was pending.

CONTEXT  These functions can only be called from an open(9E) or close(9E) routine.

EXAMPLES  The open routine sends down a T_INFO_REQ message and waits for the T_INFO_ACK. The arrival of the T_INFO_ACK is recorded by resetting a flag in the unit structure (WAIT_INFO_ACK).

The example assumes that the module is D_MTQPAIR or D_MTPERMOD.

xxopen(qp, . . .)
    queue_t ∗qp;
{
    struct xxdata ∗xx;

9F-472  SunOS 5.6  modified 1 Mar 1993
/* Allocate xxdata structure */
qprocson(qp);
/* Format T_INFO_ACK in mp */
putnext(qp, mp);
xx->xx_flags |= WAIT_INFO_ACK;
while (xx->xx_flags & WAIT_INFO_ACK)
    qwait(qp);
return (0);
}

xxrput(qp, mp)
    queue_t *qp;
    mblk_t *mp;
{
    struct xxdata *xx = (struct xxdata *)q->q_ptr;
    ...
    case T_INFO_ACK:
        if (xx->xx_flags & WAIT_INFO_ACK) {
            /* Record information from info ack */
            xx->xx_flags &='WAIT_INFO_ACK;
            freemsg(mp);
            return;
        }
    }
    ...
}

SEE ALSO close(9E), open(9E), put(9E), srv(9E) condvar(9F), mt-streams(9F), qbufcall(9F), qprocson(9F), qtimeout(9F)
STREAMS Programming Guide
Writing Device Drivers

modified 1 Mar 1993 SunOS 5.6 9F-473
NAME  qwriter – asynchronous STREAMS perimeter upgrade

SYNOPSIS  
```
#include <sys/stream.h>
#include <sys/ddi.h>

void qwriter( queue_t *qp, mblk_t *mp, void (*func)(), int perimeter);
```

INTERFACE  Solaris DDI specific (Solaris DDI).

LEVEL

ARGUMENTS  
- **qp**: Pointer to the queue.
- **mp**: Pointer to a message that will be passed in to the callback function.
- **func**: A function that will be called when exclusive (writer) access has been acquired at the specified perimeter.
- **perimeter**: Either `PERIM_INNER` or `PERIM_OUTER`.

DESCRIPTION  
`qwriter()` is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive (see `mt-streams(9F)` man page), and call the specified callback function when the upgrade has succeeded. The callback function is called as:

```
(*func)(queue_t *qp, mblk_t *mp);
```

`qwriter()` will acquire exclusive access immediately if possible, in which case the specified callback function will be executed before `qwriter()` returns. If this is not possible, `qwriter()` will defer the upgrade until later and return before the callback function has been executed. Modules should not assume that the callback function has been executed when `qwriter()` returns. One way to avoid dependencies on the execution of the callback function is to immediately return after calling `qwriter()` and let the callback function finish the processing of the message.

When `qwriter()` defers calling the callback function, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the upgrade has completed and the callback function has finished executing.

CONTEXT  
`qwriter()` can only be called from an `put(9E)` or `srv(9E)` routine, or from a `qwriter()`, `qtimeout(9F)`, or `qbufcall(9F)` callback function.

SEE ALSO  
`put(9E)`, `srv(9E)`, `mt-streams(9F)`, `qbufcall(9F)`, `qtimeout(9F)`

`STREAMS Programming Guide`
`Writing Device Drivers`
### NAME
RD, rd – get pointer to the read queue

### SYNOPSIS
```c
#include <sys/stream.h>
#include <sys/ddi.h>
queue_t *RD(queue_t *q);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### ARGUMENTS
`q` Pointer to the *write* queue whose *read* queue is to be returned.

### DESCRIPTION
The **RD()** function accepts a *write* queue pointer as an argument and returns a pointer to the *read* queue of the same module.

**CAUTION:** Make sure the argument to this function is a pointer to a *write* queue. **RD()** will not check for queue type, and a system panic could result if it is not the right type.

### RETURN VALUES
The pointer to the *read* queue.

### CONTEXT
**RD()** can be called from user or interrupt context.

### EXAMPLES
See the **qreply(9F)** function page for an example of **RD()**.

### SEE ALSO
**qreply(9F)**, **WR(9F)**

*Writing Device Drivers*

*STREAMS Programming Guide*
NAME  rmalloc – allocate space from a resource map

SYNOPSIS
#include <sys/map.h>
#include <sys/ddi.h>

unsigned long rmalloc(struct map *, size_t size);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
mp Resource map from where the resource is drawn.
size Number of units of the resource.

DESCRIPTION rmalloc() is used by a driver to allocate space from a previously defined and initialized
resource map. The map itself is allocated by calling the function rmallocmap(9F). rmalloc()
is one of five functions used for resource map management. The other functions include:

rmalloc_wait(9F) Allocate space from a resource map, wait if necessary.
rmfree(9F) Return previously allocated space to a map.
rmallocmap(9F) Allocate a resource map and initialize it.
rmfreemap(9F) Deallocate a resource map.

rmalloc() allocates space from a resource map in terms of arbitrary units. The system
maintains the resource map by size and index, computed in units appropriate for the
resource. For example, units may be byte addresses, pages of memory, or blocks. The
normal return value is an unsigned long set to the value of the index where sufficient
free space in the resource was found.

RETURN VALUES Under normal conditions, rmalloc() returns the base index of the allocated space. Otherwise,
rmalloc() returns a 0 if all resource map entries are already allocated.

CONTEXT rmalloc() can be called from user or interrupt context.

EXAMPLES The following example is a simple memory map, but it illustrates the principles of map
management. A driver allocates and initializes the map by calling both the
rmallocmap(9F) and rmfree(9F) functions. rmallocmap(9F) is called to establish the
number of slots or entries in the map, and rmfree(9F) to initialize the resource area the
map is to manage. The following example is a fragment from a hypothetical start routine
and illustrates the following procedures:

Panics the system if the required amount of memory can not be allocated (lines
11–15).

Uses rmallocmap(9F) to configure the total number of entries in the map, and
rmfree(9F) to initialize the total resource area.
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.
#define XX_BUFSIZE 2560
#define XX_MAXSIZE (XX_BUFSIZE / 4)

static struct map xx_mp; /* Private buffer space map */
...

xread(dev_t dev, uio_t *uiop, cred_t *credp)
{
    register caddr_t addr;
    register int size;
    size = min(COUNT, XX_MAXSIZE); /* Break large I/O request */
    /* into small ones */
    /* Get buffer. */
    if ((addr = (caddr_t)rmalloc(xx_mp, size)) == 0)
        cmn_err(CE_PANIC, "read: rmalloc failed allocation of size %d",
                size);
    /* Move data to buffer. If invalid address is found,
    * return buffer to map and return error code. */
    if (uiomove(addr, size, UIO_READ, uiop) == -1) {
        rmfree(xx_mp, size, addr);
        return(EFAULT);
    }
}

SEE ALSO kmem_alloc(9F), rmalloc_wait(9F), rmallocmap(9F), rmfree(9F), rmfreemap(9F),
uiomove(9F)

Writing Device Drivers

9F-478 SunOS 5.6 modified 19 Nov 1992
NAME  rmallocmap, rmallocmap_wait, rmfreemap – allocate and free resource maps

SYNOPSIS  
#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
mapsize  Number of entries for the map.
mp  A pointer to the map structure to be deallocated.

DESCRIPTION  
rmallocmap() dynamically allocates a resource map structure. The argument mapsize defines the total number of entries in the map. In particular it is the total number allocations that can be outstanding at any one time.

rmallocmap() initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource a call to rmfree(9F) is used to make the entirety of the actual resource available for allocation starting from the first index into the resource. Typically the call to rmallocmap() is followed by a call to rmfree(9F), passing the address of the map returned from rmallocmap(), the total size of the resource, and the first index into actual resource.

The resource map allocated by rmallocmap() can be used to describe an arbitrary resource in whatever allocation units are appropriate such blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to rmalloc(9F), rmalloc_wait(9F), and rmfree(9F).

rmallocmap_wait() is similar to rmallocmap(), with the exception that it will wait for space to become available if necessary.

rmfreemap() deallocates a resource map structure previously allocated by rmallocmap() or rmallocmap_wait(). The argument mp is a pointer to the map structure to be deallocated.

RETURN VALUES  
Upon successful completion, rmallocmap() and rmallocmap_wait() return a pointer to the newly allocated map structure. Upon failure, rmallocmap() returns a NULL pointer.

CONTEXT  
rmallocmap() and rmfreemap() can be called from user, kernel or interrupt context.
rmallocmap_wait() can only be called from user or kernel context.

modified 20 Nov 1996
SEE ALSO  `rmalloc(9F), rmalloc_wait(9F), rmfree(9F)`

*Writing Device Drivers*
<table>
<thead>
<tr>
<th>NAME</th>
<th>rmalloc_wait – allocate space from a resource map, wait if necessary</th>
</tr>
</thead>
</table>
| SYNOPSIS   | #include <sys/map.h>  
              #include <sys/ddi.h>  
              unsigned long rmalloc_wait(struct map *mp, size_t size); |
| INTERFACE LEVEL | Architecture independent level 1 (DDI/DKI). |
| ARGUMENTS  | mp Pointer to the resource map from which space is to be allocated.  
            size Number of units of space to allocate. |
| DESCRIPTION| rmalloc_wait() requests an allocation of space from a resource map. rmalloc_wait() is similar to the malloc(9F) function with the exception that it will wait for space to become available if necessary. |
| RETURN VALUES| rmalloc_wait() returns the base of the allocated space. |
| CONTEXT    | This functions can be called from user or interrupt context. However in most cases rmalloc_wait() should be called from user context only. |
| SEE ALSO   | rmalloc(9F), rmallocmap(9F), rmfree(9F), rmfreemap(9F)  
              Writing Device Drivers |
NAME  rmfree – free space back into a resource map

SYNOPSIS  
```
#include <sys/map.h>
#include <sys/ddi.h>
void rmfree(struct map *mp, size_t size, ulong_t index);
```

INTERFACE  Architecture independent level 1 (DDI/DKI).

LEVEL  

ARGUMENTS  
- `mp`  Pointer to the map structure.
- `size`  Number of units being freed.
- `index`  Index of the first unit of the allocated resource.

DESCRIPTION  rmfree() releases space back into a resource map. It is the opposite of rmalloc(9F), which allocates space that is controlled by a resource map structure.

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.

CONTEXT  rmfree() can be called from user or interrupt context.

SEE ALSO  rmalloc(9F), rmalloc_wait(9F), rmallocmap(9F), rmfreemap(9F)

Writing Device Drivers
NAME
rmvb – remove a message block from a message

SYNOPSIS
#include <sys/stream.h>
mblk_t *rmvb(mblk_t *mp, mblk_t *bp);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
mp Message from which a block is to be removed. mblk_t is an instance of the msgb(9S) structure.
bp Message block to be removed.

DESCRIPTION
rmvb( ) removes a message block (bp) from a message (mp), and returns a pointer to the altered message. The message block is not freed, merely removed from the message. It is the module or driver’s responsibility to free the message block.

RETURN VALUES
If successful, a pointer to the message (minus the removed block) is returned. The pointer is NULL if bp was the only block of the message before rmvb( ) was called. If the designated message block (bp) does not exist, -1 is returned.

CONTEXT
rmvb( ) can be called from user or interrupt context.

EXAMPLE
This routine removes all zero-length M_DATA message blocks from the given message. For each message block in the message, save the next message block (line 10). If the current message block is of type M_DATA and has no data in its buffer (line 11), then remove it from the message (line 12) and free it (line 13). In either case, continue with the next message block in the message (line 16).

```c
void xxclean(mp)
    mblk_t *mp;
{ 
    mblk_t *tmp;
    mblk_t *nmp;
    tmp = mp;
    while (tmp) {
        nmp = tmp->b_cont;
        if (((tmp->b_datap->db_type == M_DATA) &&
            (tmp->b_rptr == tmp->b_wptr)) { 
            (void) rmvb(mp, tmp);
            freeb(tmp);
        }
        tmp = nmp;
    }
}
```

modified 11 Apr 1991 SunOS 5.6 9F-483
### SEE ALSO
- `freeb(9F), msgb(9S)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
NAME
rmvq – remove a message from a queue

SYNOPSIS
#include <sys/stream.h>
void rmvq(queue_t *q, mblk_t *mp);

INTERFACE
LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
q  Queue containing the message to be removed.
mp  Message to remove.

DESCRIPTION
rmvq() removes a message from a queue. A message can be removed from anywhere on a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either rmvq() or getq(9F) should be used to remove a message from a queue.

CONTEXT
rmvq() can be called from user or interrupt context.

EXAMPLE
This code fragment illustrates how one may flush one type of message from a queue. In this case, only M_PROTO T_DATA_IND messages are flushed. For each message on the queue, if it is an M_PROTO message (line 8) of type T_DATA_IND (line 10), save a pointer to the next message (line 11), remove the T_DATA_IND message (line 12) and free it (line 13). Continue with the next message in the list (line 19).

```c
1 mblk_t *mp, *nmp;
2 queue_t *q;
3 union T_primitives *tp;
4
5 freezestr(q);
6 mp = q->q_first;
7 while (mp) {
8     if (mp->b_datap->db_type == M_PROTO) {
9         tp = (union T_primitives *)mp->b_rptr;
10         if (tp->type == T_DATA_IND) {
11             nmp = mp->b_next;
12             rmvq(q, mp);
13             freemsg(mp);
14             mp = nmp;
15         } else {
16             mp = mp->b_next;
17         }
18     } else {
19         mp = mp->b_next;
20     }
21 }
22 unfreezestr(q);
```

modified 28 Jan 1993 SunOS 5.6 9F-485
SEE ALSO freemsg(9F), freezeustr(9F), getq(9F), insq(9F), unfreezeustr(9F)

Writing Device Drivers
STREMS Programming Guide

WARNINGS Make sure that the message mp is linked onto q to avoid a possible system panic.

NOTES The stream must be frozen using freezeustr(9F) before calling rmvq().
NAME
rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade,
rw_tryupgrade, rw_read_locked – readers/writer lock functions

SYNOPSIS
#include <sys/ksynch.h>

void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
rwlp Pointer to a krwlock_t readers/writer lock.
name Descriptive string. This is obsolete and should be NULL. (Non-NULL
strings are legal, but they’re a waste of kernel memory.)
type Type of readers/writer lock.
arg Type-specific argument for initialization function.
enter_type Indication of whether the lock is to be acquired non-exclusively or
exclusively RW_READER or RW_WRITER.

DESCRIPTION
A multiple-readers, single-writer lock is represented by the krwlock_t data type. This
type of lock will allow many threads to have simultaneous read-only access to an object.
Only one thread may have write access at any one time. An object which is searched
more frequently than it is changed is a good candidate for a readers/writer lock.
Readers/writer locks are slightly more expensive than mutex locks, and the advantage of
multiple read access may not occur if the lock will only be held for a short time.

rw_init() initializes a readers/writer lock. It is an error to initialize a lock more than
once. The type argument should be set to RW_DRIVER. If the lock is used by the inter-
rupt handler, the type-specific argument, arg, should be the ddi_iblock_cookie returned
from ddi_get_iblock_cookie(9F) or ddi_get_soft_iblock_cookie(9F). If the lock is not
used by any interrupt handler, the argument should be NULL.

rw_destroy() releases any resources that might have been allocated by rw_init(). It
should be called before freeing the memory containing the lock.

rw_enter() acquires the lock, and blocks if necessary. If enter_type is RW_READER, the
caller blocks if there is a writer or a thread attempting to enter for writing. If enter_type is
RW_WRITER, the caller blocks if any thread holds the lock.
NOTE: it is a programming error for any thread to acquire an rwlock it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

**rw_exit()** releases the lock and may wake up one or more threads waiting on the lock.

**rw_tryenter()** attempts to enter the lock, like **rw_enter()**, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with **RW_WRITER**), may call **rw_downgrade()** to convert to holding the lock non-exclusively (as if entered with **RW_READER**). One or more waiting readers may be unblocked.

**rw_tryupgrade()** can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

**rw_read_locked()** returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if **rw_read_locked()** is called for a lock that isn’t held by the caller.

**RETURN VALUES**
- 0: **rw_tryenter()** could not obtain the lock without blocking.
- 0: **rw_tryupgrade()** was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
- 0: **rw_read_locked()** returns 0 if the lock is held by the caller for write.
- non-zero: from **rw_read_locked()** if the lock is held by the caller for read.
- non-zero: successful return from **rw_tryenter()** or **rw_tryupgrade()**.

**CONTEXT**
These functions can be called from user or interrupt context, except for **rw_init()** and **rw_destroy()**, which can be called from user context only.

**SEE ALSO**
- condvar(9F), ddi_add_intr(9F), ddi_get_iblock_cookie(9F), ddi_get_soft_iblock_cookie(9F), mutex(9F), semaphore(9F)
- Writing Device Drivers

**NOTES**
Compiling with _LOCKTEST or _MPSTATS defined no longer has any effect. To gather lock statistics, see lockstat(1M).
### NAME
SAMESTR, samestr – test if next queue is in the same stream

### SYNOPSIS
```c
#include <sys/stream.h>
int SAMESTR(queue_t *q);
```

### INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

### ARGUMENTS
- `q` Pointer to the queue.

### DESCRIPTION
The `SAMESTR()` function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This function accounts for the twisted queue connections that occur in a STREAMS pipe and should be used in preference to direct examination of the `q_next` field of `queue(9S)` to see if the stream continues beyond `q`.

### RETURN VALUES
- `SAMESTR()` returns 1 if the next queue is the same type as the current queue. It returns 0 if the next queue does not exist or if it is not the same type.

### CONTEXT
`SAMESTR()` can be called from user or interrupt context.

### SEE ALSO
- `OTHERQ(9F)`
- *Writing Device Drivers*
- *STREAMS Programming Guide*
NAME  scsi_abort – abort a SCSI command

SYNOPSIS  
```c
#include <sys/scsi/scsi.h>
int scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
```

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
- `ap`  Pointer to a `scsi_address` structure.
- `pkt`  Pointer to a `scsi_pkt(9S)` structure.

DESCRIPTION  `scsi_abort()` terminates a command that has been transported to the host adapter driver. A NULL `pkt` causes all outstanding packets to be aborted. On a successful abort, the `pkt_reason` is set to `CMD_ABORTED` and `pkt_statistics` is OR’ed with `STAT_ABORTED`.

RETURN VALUES  `scsi_abort()` returns:
- 1 on success.
- 0 on failure.

CONTEXT  `scsi_abort()` can be called from user or interrupt context.

EXAMPLE  
```c
if (scsi_abort(&devp->sd_address, pkt) == 0) {
    (void) scsi_reset(&devp->sd_address, RESET_ALL);
}
```

SEE ALSO  `tran_abort(9E), scsi_reset(9F), scsi_pkt(9S)`

`Writing Device Drivers`
**NAME**
scsi_alloc_consistent_buf – allocate an I/O buffer for SCSI DMA

**SYNOPSIS**
```
#include <sys/scsi/scsi.h>

struct buf *scsi_alloc_consistent_buf(struct scsi_address *ap, struct buf *bp,
    size_t datalen, u_int bflags, int (*callback) (caddr_t), caddr_t arg);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**ARGUMENTS**
- `ap` Pointer to the `scsi_address(9S)` structure.
- `bp` Pointer to the `buf(9S)` structure.
- `datalen` Number of bytes for the data buffer.
- `bflags` Flags setting for the allocated buffer header.
- `callback` A pointer to a callback function, `NULL_FUNC` or `SLEEP_FUNC`.
- `arg` The callback function argument.

**DESCRIPTION**
`sisci_alloc_consistent_buf()` allocates a buffer header and the associated data buffer for direct memory access (DMA) transfer. This buffer is allocated from the `iobp` space, which is considered consistent memory. For more details, see `ddi_dma_mem_alloc(9F)` and `ddi_dma_sync(9F)`.

For buffers allocated via `scsi_alloc_consistent_buf()`, and marked with the `PKT_CONSISTENT` flag via `scsi_init_pkt(9F)`, the HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver’s command completion callback is performed.

If `bp` is `NULL`, a new buffer header will be allocated using `getrbuf(9F)`. In addition, if `datalen` is non-zero, a new buffer will be allocated using `ddi_dma_mem_alloc(9F)`.

`callback` indicates what the allocator routines should do when direct memory access (DMA) resources are not available; the valid values are:

- `NULL_FUNC` Do not wait for resources. Return a `NULL` pointer.
- `SLEEP_FUNC` Wait indefinitely for resources.
- Other Values `callback` points to a function that is called when resources may become available. `callback` must return either 0 (indicating that it attempted to allocate resources but failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry. The last argument `arg` is supplied to the `callback` function when it is invoked.

**RETURN VALUES**
`scci_alloc_consistent_buf()` returns a pointer to a `buf(9S)` structure on success. It returns `NULL` if resources are not available and `waitfunc` was not `SLEEP_FUNC`.

modified 20 Nov 1996

SunOS 5.6

9F-491
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

Allocate a request sense packet with consistent DMA resources attached.

```c
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);
```

Allocate an inquiry packet with consistent DMA resources attached.

```c
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
                               SUN_INQSIZE, B_READ, canwait, NULL);
if (bp) {
    pkt = scsi_init_pkt(devp->sd_address, NULL, bp,
                        CDB_GROUP0, 1, PP_LEN, PKT_CONSISTENT,
                        canwait, NULL);
}
```

SEE ALSO

`ddi_dma_mem_alloc(9F)`, `ddi_dma_sync(9F)`, `getrbuf(9F)`, `scsi_destroy_pkt(9F)`, `scsi_init_pkt(9F)`, `scsi_free_consistent_buf(9F)`, `buf(9S)`, `scsi_address(9S)`

_Writing Device Drivers_
### NAME

`scsi_cname`, `scsi_dname`, `scsi_mname`, `scsi_rname`, `scsi_sname` – decode a SCSI name

### SYNOPSIS

```c
#include <sys/scsi/scsi.h>
char *scsi_cname(u_char cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(u_char msg);
char *scsi_rname(u_char reason);
char *scsi_sname(u_char sense_key);
```

### INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

### ARGUMENTS

- `cmd` A SCSI command value.
- `cmdvec` Pointer to an array of command strings.
- `dtype` Device type.
- `msg` A message value.
- `reason` A packet reason value.
- `sense_key` A SCSI sense key value.

### DESCRIPTION

- **`scsi_cname()`** decodes SCSI commands. `cmdvec` is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.
- **`scsi_dname()`** decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.
- **`scsi_mname()`** decodes SCSI messages.
- **`scsi_rname()`** decodes packet completion reasons.
- **`scsi_sname()`** decodes SCSI sense keys.

### RETURN VALUES

These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

### CONTEXT

These functions can be called from user or interrupt context.

---

modified 21 Dec 1992

SunOS 5.6

9F-493
**EXAMPLES**

`scsi_cname()` decodes SCSI tape commands as follows:

```c
static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
    "\003request sense",
    "\010read",
    "\012write",
    "\020write file mark",
    "\021space",
    "\022inquiry",
    "\025mode select",
    "\031erase tape",
    "\032mode sense",
    "\033load tape",
    NULL
};
```

```c
.. cmn_err(CE_CONT, "st: cmd=%s", scsi_cname(cmd, st_cmds));
```

**SEE ALSO**

*Writing Device Drivers*
NAME       scsi_destroy_pkt – free an allocated SCSI packet and its DMA resource

SYNOPSIS  #include <sys/scsi/scsi.h>
           void scsi_destroy_pkt(struct scsi_pkt *pktp);

INTERFACE LEVEL       Solaris DDI specific (Solaris DDI).

ARGUMENTS       pktp    Pointer to a scsi_pkt(9S) structure.

DESCRIPTION  scsi_destroy_pkt() releases all necessary resources, typically at the end of an I/O transfer. The data is synchronized to memory, then the DMA resources are deallocated and pktp is freed.

CONTEXT      scsi_destroy_pkt() may be called from user or interrupt context.

EXAMPLE      scsi_destroy_pkt(un->un_rq);

SEE ALSO     tran_destroy_pkt(9E), scsi_init_pkt(9F), scsi_pkt(9S)

Writing Device Drivers

modified 30 Aug 1995    SunOS 5.6    9F-495
**NAME**

`scsi_dmaget`, `scsi_dmafree` – SCSI dma utility routines

**SYNOPSIS**

```c
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_dmaget(struct scsi_pkt *pkt, opaque_t dmatoken,
                              int (*callback)(void));

void scsi_dmafree(struct scsi_pkt *pkt);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

- **pkt** A pointer to a `scsi_pkt(9S)` structure.
- **dmatoken** Pointer to an implementation dependent object
- **callback** Pointer to a callback function, or `NULL_FUNC` or `SLEEP_FUNC`.

**DESCRIPTION**

`scsi_dmaget()` allocates DMA resources for an already allocated SCSI packet. `pkt` is a pointer to the previously allocated SCSI packet (see `scsi_pktalloc(9F)`).

`dmatoken` is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The `dmatoken` must be a pointer to a `buf(9S)` structure. If `dmatoken` is `NULL`, no resources are allocated.

`callback` indicates what `scsi_dmaget()` should do when resources are not available:

- `NULL_FUNC` Do not wait for resources. Return a `NULL` pointer.
- `SLEEP_FUNC` Wait indefinitely for resources.
- Other Values `callback` points to a function which is called when resources may have become available. `callback` must return either `0` (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or `1` indicating either success in allocating resources or indicating that it no longer cares for a retry.

`scsi_dmafree()` frees the DMA resources associated with the SCSI packet. The packet itself remains allocated.

**RETURN VALUES**

`scsi_dmaget()` returns a pointer to a `scsi_pkt` on success. It returns `NULL` if resources are not available.

**CONTEXT**

If `callback` is `SLEEP_FUNC`, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The `callback` function may not block or call routines that block.

`scsi_dmafree()` can be called from user or interrupt context.

---

9F-496 SunOS 5.6 modified 21 Dec 1992
SEE ALSO

- scsi_pktalloc(9F), scsi_pktfree(9F), scsi_resalloc(9F), scsi_resfree(9F), buf(9S),
- scsi_pkt(9S)

Writing Device Drivers

modified 21 Dec 1992

SunOS 5.6

9F-497
NAME  scsi_errmsg – display a SCSI request sense message

SYNOPSIS  

```c
#include <sys/scsi/scsi.h>
void scsi_errmsg(struct scsi_device *devp, struct scsi_pkt *pktp, char *drv_name,
                int severity, int blkno, int err_blkno, struct scsi_key_strings *cmdlist,
                struct scsi_extended_sense *sensep);
```

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  

- `devp`  Pointer to the `scsi_device(9S)` structure.
- `pktp`  Pointer to a `scsi_pkt(9S)` structure.
- `drv_name`  String used by `scsi_log(9F)`.
- `severity`  Error severity level, maps to severity strings below.
- `blkno`  Requested block number.
- `err_blkno`  Error block number.
- `cmdlist`  An array of SCSI command description strings.
- `sensep`  A pointer to a `scsi_extended_sense(9S)` structure.

DESCRIPTION  `scsi_errmsg()` interprets the request sense information in the `sensep` pointer and generates a standard message that is displayed using `scsi_log(9F)`. The first line of the message is always a `CE_WARN`, with the continuation lines being `CE_CONT`. `sensep` may be `NULL`, in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The `scsi_device(9S)` structure denoted by `devp` supplies the identification of the device that requested the display. `severity` selects which string is used in the "Error Level:" reporting, according to the table below:

<table>
<thead>
<tr>
<th>Severity Value:</th>
<th>String:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_ERR_ALL</td>
<td>All</td>
</tr>
<tr>
<td>SCSI_ERR_UNKNOWN</td>
<td>Unknown</td>
</tr>
<tr>
<td>SCSI_ERR_INFO</td>
<td>Informational</td>
</tr>
<tr>
<td>SCSI_ERR_RECOVERED</td>
<td>Recovered</td>
</tr>
<tr>
<td>SCSI_ERR_RETRYABLE</td>
<td>Retryable</td>
</tr>
<tr>
<td>SCSI_ERR_FATAL</td>
<td>Fatal</td>
</tr>
</tbody>
</table>

`blkno` is the block number of the original request that generated the error. `err_blkno` is the block number where the error occurred. `cmdlist` is a mapping table for translating the SCSI command code in `pktp` to the actual command string.
The `cmdlist` is described in the structure below:

```c
struct scsi_key_strings {
    int key;
    char *message;
};
```

For a basic SCSI disk, the following list is appropriate:

```c
static struct scsi_key_strings scsi_cmds[] = {
    0x00, "test unit ready",
    0x01, "rezero/rewind",
    0x03, "request sense",
    0x04, "format",
    0x07, "reassign",
    0x08, "read",
    0x0a, "write",
    0x0b, "seek",
    0x12, "inquiry",
    0x15, "mode select",
    0x16, "reserve",
    0x17, "release",
    0x18, "copy",
    0x1a, "mode sense",
    0x1b, "start/stop",
    0x1e, "door lock",
    0x28, "read(10)"
    0x2a, "write(10)"
    0x2f, "verify",
    0x37, "read defect data",
    0x3b, "write buffer",
    -1, NULL
};
```

**EXAMPLES**

This entry:

```c
scsi_errmsg(devp, pkt, "sd", SCSI_ERR_INFO, bp->b_blkno, err_blkno, sd_cmds, rqsense);
```

Generates:

```
WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1):
Error for Command: read  Error Level: Informational
Requested Block: 23936  Error Block: 23936
Vendor: QUANTUM  Serial Number: 123456
Sense Key: Unit Attention
ASC: 0x29 (reset), ASCQ: 0x0, FRU: 0x0
```
SEE ALSO

cmn_err(9F), scsi_log(9F), scsi_device(9S), scsi_extended_sense(9S), scsi_pkt(9S)

Writing Device Drivers
NAME

scsi_free_consistent_buf – free a previously allocated SCSI DMA I/O buffer

SYNOPSIS

```c
#include <sys/scsi/scsi.h>
void scsi_free_consistent_buf(struct buf *bp);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

bp Pointer to the buf(9S) structure.

DESCRIPTION

`scsi_free_consistent_buf()` frees a buffer header and consistent data buffer that was previously allocated using `scsi_alloc_consistent_buf(9F)`.

CONTEXT

`scsi_free_consistent_buf()` may be called from either the user or the interrupt levels.

SEE ALSO

`freerbuf(9F), scsi_alloc_consistent_buf(9F), buf(9S)`

`Writing Device Drivers`

WARNING

`scsi_free_consistent_buf()` will call `freerbuf(9F)` to free the `buf(9S)` that was allocated before or during the call to `scsi_alloc_consistent_buf(9F)`.
**NAME**

scsi_hba_attach_setup, scsi_hba_attach, scsi_hba_detach – SCSI HBA attach and detach routines

**SYNOPSIS**

```c
#include <sys/scsi/scsi.h>

int scsi_hba_attach_setup(dev_info_t *dip, ddi_dma_attr_t *hba_dma_attr,
                           scsi_hba_tran_t *hba_tran, int hba_flags);
int scsi_hba_attach(dev_info_t *dip, ddi_dma_lim_t *hba_lim,
                    scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);
int scsi_hba_detach(dev_info_t *dip);
```

**INTERFACE LEVEL**

Solaris architecture specific (Solaris DDI).

**ARGUMENTS**

- **dip**
  A pointer to the `dev_info_t` structure, referring to the instance of the HBA device.
- **hba_lim**
  A pointer to a `ddi_dma_lim(9S)` structure.
- **hba_tran**
  A pointer to a `scsi_hba_tran(9S)` structure.
- **hba_flags**
  Flag modifiers. The only defined flag value is `SCSI_HBA_TRAN_CLONE`.
- **hba_options**
  Optional features provided by the HBA driver for future extensions; must be `NULL`.
- **hba_dma_attr**
  A pointer to a `ddi_dma_attr(9S)` structure.

**DESCRIPTION**

`scsi_hba_attach_setup()` is the recommended interface over `scsi_hba_attach()`.

`scsi_hba_attach()` registers the DMA limits `hba_lim` and the transport vectors `hba_tran` of each instance of the HBA device defined by `dip`. `scsi_hba_attach_setup()` registers the DMA attributes `hba_dma_attr` and the transport vectors `hba_tran` of each instance of the HBA device defined by `dip`. The HBA driver can pass different DMA limits or DMA attributes, and transport vectors for each instance of the device, as necessary, to support any constraints imposed by the HBA itself.

`scsi_hba_attach()` and `scsi_hba_attach_setup()` use the `dev_bus_ops` field in the `dev_ops(9S)` structure. The HBA driver should initialize this field to `NULL` before calling `scsi_hba_attach()` or `scsi_hba_attach_setup()`.

If `SCSI_HBA_TRAN_CLONE` is requested in `hba_flags`, the `hba_tran` structure will be cloned once for each target attached to the HBA. The cloning of the structure will occur before the `tran_tgt_init(9E)` entry point is called to initialize a target. At all subsequent HBA entry points, including `tran_tgt_init(9E)`, the `scsi_hba_tran_t` structure passed as an argument or found in a `scsi_address` structure will be the ’cloned’ `scsi_hba_tran_t` structure, thus allowing the HBA to use the `tran_tgt_private` field in the `scsi_hba_tran_t` structure to point to per-target data. The HBA must take care to free only the same `scsi_hba_tran_t` structure allocated when detaching; all ’cloned’ `scsi_hba_tran_t` structures allocated by the system will be freed by the system.
scsi_hba_attach() and scsi_hba_attach_setup() attach a number of integer-valued properties to dip, unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via ddi_prop_get_int(9F), and respect any settings for features provided the HBA.

**scsi-options**
optional SCSI configuration bits

**SCSI_OPTIONS_DR**
if not set, the HBA should not grant Disconnect privileges to target devices.

**SCSI_OPTIONS_LINK**
if not set, the HBA should not enable Linked Commands.

**SCSI_OPTIONS_TAG**
if not set, the HBA should not operate in Command Tagged Queueing mode.

**SCSI_OPTIONS_FAST**
if not set, the HBA should not operate the bus in FAST SCSI mode.

**SCSI_OPTIONS_FAST20**
if not set, the HBA should not operate the bus in FAST20 SCSI mode.

**SCSI_OPTIONS_WIDE**
if not set, the HBA should not operate the bus in WIDE SCSI mode.

**SCSI_OPTIONS_SYNC**
if not set, the HBA should not operate the bus in synchronous transfer mode.

**scsi-reset-delay**
SCSI bus or device reset recovery time, in milliseconds.

scsi_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

modified 30 Aug 1995  
SunOS 5.6  
9F-503
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.
NAME

scsi_hba_init, scsi_hba_fini – SCSI Host Bus Adapter system initialization and completion routines

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_init(struct modlinkage *modlp);
void scsi_hba_fini(struct modlinkage *modlp);

INTERFACE

LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

modlp Pointer to the Host Bus Adapters module linkage structure.

DESCRIPTION

scsi_hba_init() scsi_hba_init() is the system-provided initialization routine for SCSI HBA drivers. The scsi_hba_init() function registers the HBA in the system and allows the driver to accept configuration requests on behalf of SCSI target drivers. The scsi_hba_init() routine must be called in the HBA’s _init(9E) routine before mod_install(9F) is called. If mod_install(9F) fails, the HBA’s _init(9E) should call scsi_hba_fini() before returning failure.

scsi_hba_fini() scsi_hba_fini() is the system provided completion routine for SCSI HBA drivers.
scsi_hba_fini() removes all of the system references for the HBA that were created in scsi_hba_init(). The scsi_hba_fini() routine should be called in the HBA’s _fini(9E) routine if mod_remove(9F) is successful.

RETURN VALUES

scsi_hba_init() returns 0 if successful, and a non-zero value otherwise. If scsi_hba_init() fails, the HBA’s _init() entry point should return the value returned by scsi_hba_init().

CONTEXT

scsi_hba_init() and scsi_hba_fini() should be called from _init(9E) or _fini(9E), respectively.

SEE ALSO

_mod_install(9F), _mod_remove(9F), scsi_pktalloc(9F),
scsi_pktfree(9F), scsi_hba_tran(9S)

Writing Device Drivers

NOTES

The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once scsi_hba_fini() is called.

modified 1 Nov 1993

SunOS 5.6

9F-505
**NAME**
scsi_hba_lookup_capstr – return index matching capability string

**SYNOPSIS**
```c
#include <sys/scsi/scsi.h>
int scsi_hba_lookup_capstr(char *capstr);
```

**INTERFACE LEVEL**
Solaris architecture specific (Solaris DDI).

**ARGUMENTS**
- `capstr` Pointer to a string.

**DESCRIPTION**
`scsi_hba_lookup_capstr()` attempts to match `capstr` against a known set of capability strings, and returns the defined index for the matched capability, if found.

The set of indices and capability strings is:

<table>
<thead>
<tr>
<th>Capability String</th>
<th>Matched String</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_CAP_DMA_MAX</td>
<td>&quot;dma-max&quot; or &quot;dma_max&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_MSG_OUT</td>
<td>&quot;msg-out&quot; or &quot;msg_out&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_DISCONNECT</td>
<td>&quot;disconnect&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_SYNCHRONOUS</td>
<td>&quot;synchronous&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_WIDE_XFER</td>
<td>&quot;wide-xfer&quot; or &quot;wide_xfer&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_PARITY</td>
<td>&quot;parity&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_INITIATOR_ID</td>
<td>&quot;initiator-id&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_UNTAGGED_QING</td>
<td>&quot;untagged-qing&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_TAGGED_QING</td>
<td>&quot;tagged-qing&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_ARQ</td>
<td>&quot;auto-rqsense&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_LINKED_CMDS</td>
<td>&quot;linked-cmds&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_SECTOR_SIZE</td>
<td>&quot;sector-size&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_TOTAL_SECTORS</td>
<td>&quot;total-sectors&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_GEOMETRY</td>
<td>&quot;geometry&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_RESET_NOTIFICATION</td>
<td>&quot;reset-notification&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_QFULL_RETRIES</td>
<td>&quot;qfull-retries&quot;</td>
</tr>
<tr>
<td>SCSI_CAP_QFULL_RETRY_INTERVAL</td>
<td>&quot;qfull-retry-interval&quot;</td>
</tr>
</tbody>
</table>

**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*

9F-506 SunOS 5.6 modified 30 Aug 1995
### NAME
scsi_hba_pkt_alloc, scsi_hba_pkt_free – allocate and free a scsi_pkt structure

### SYNOPSIS
```c
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_hba_pkt_alloc(dev_info_t *dip, struct scsi_address *ap,
    int cmdlen, int statuslen, int tgtlen, int hbalen, int (*callback)(caddr_t arg),
    caddr_t arg);

void scsi_hba_pkt_free(struct scsi_address *ap, struct scsi_pkt *pkt);
```

### INTERFACE LEVEL
Solaris architecture specific (Solaris DDI).

### ARGUMENTS
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip</td>
<td>Pointer to a dev_info_t structure, defining the HBA driver instance.</td>
</tr>
<tr>
<td>ap</td>
<td>Pointer to a scsi_address(9S) structure, defining the target instance.</td>
</tr>
<tr>
<td>cmdlen</td>
<td>Length in bytes to be allocated for the SCSI command descriptor block (CDB).</td>
</tr>
<tr>
<td>statuslen</td>
<td>Length in bytes to be allocated for the SCSI status completion block (SCB).</td>
</tr>
<tr>
<td>tgtlen</td>
<td>Length in bytes to be allocated for a private data area for the target driver’s exclusive use.</td>
</tr>
<tr>
<td>hbalen</td>
<td>Length in bytes to be allocated for a private data area for the HBA driver’s exclusive use.</td>
</tr>
<tr>
<td>callback</td>
<td>indicates what scsi_hba_pkt_alloc() should do when resources are not available:</td>
</tr>
<tr>
<td>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>arg</td>
<td>Must be NULL.</td>
</tr>
<tr>
<td>pkt</td>
<td>A pointer to a scsi_pkt(9S) structure.</td>
</tr>
</tbody>
</table>

### DESCRIPTION
scsi_hba_pkt_alloc() allocates space for a scsi_pkt structure. HBA drivers should use this interface when allocating a scsi_pkt from their tran_init_pkt(9E) entry point.

If callback is NULL_FUNC, scsi_hba_pkt_alloc() may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.

scsi_hba_pkt_alloc() copies the scsi_address(9S) structure pointed to by ap to the pkt_address field in the scsi_pkt(9S).
scsi_hba_pkt_alloc() also allocates memory for these scsi_pkt(9S) data areas, and sets these fields to point to the allocated memory:

- **pkt_ha_private**: HBA private data area
- **pkt_private**: target driver private data area
- **pkt_scbp**: SCSI status completion block
- **pkt_cdbp**: SCSI command descriptor block

**scsi_hba_pkt_free()**

frees the space allocated for the scsi_pkt(9S) structure.

**RETURN VALUES**

scsi_hba_pkt_alloc() returns a pointer to the scsi_pkt structure, or NULL if no space is available.

**CONTEXT**

scsi_hba_pkt_alloc() can be called from user or interrupt context. Drivers must not allow scsi_hba_pkt_alloc() to sleep if called from an interrupt routine.

scsi_hba_pkt_free() can be called from user or interrupt context.

**SEE ALSO**

tran_init_pkt(9E), scsi_address(9S), scsi_pkt(9S)

Writing Device Drivers
NAME

scsi_hba_probe – default SCSI HBA probe function

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_probe(struct scsi_device *sd, int (*waitfunc)(void));

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

sd Pointer to a scsi_device(9S) structure describing the target.

waitfunc NULL_FUNC or SLEEP_FUNC.

DESCRIPTION

scsi_hba_probe() is a function providing the semantics of scsi_probe(9F). An HBA
driver may call scsi_hba_probe() from its tran_tgt_probe(9E) entry point, to probe for
the existence of a target on the SCSI bus, or the HBA may set tran_tgt_probe(9E) to point
to scsi_hba_probe directly.

RETURN VALUES

See scsi_probe(9F) for the return values from scsi_hba_probe().

CONTEXT

scsi_hba_probe() should only be called from the HBA’s tran_tgt_probe(9E) entry point.

SEE ALSO

tran_tgt_probe(9E), scsi_probe(9F), scsi_device(9S)

Writing Device Drivers

modified 30 Aug 1995

SunOS 5.6

9F-509
### NAME
scsi_hba_tran_alloc, scsi_hba_tran_free – allocate and free transport structures

### SYNOPSIS
```c
#include <sys/scsi/scsi.h>

scsi_hba_tran_t *scsi_hba_tran_alloc(dev_info_t *dip, int flags);

void scsi_hba_tran_free(scsi_hba_tran_t *hba_tran);
```

### INTERFACE
Solaris architecture specific (Solaris DDI).

### ARGUMENTS
- **dip**
  Pointer to a `dev_info` structure, defining the HBA driver instance.
- **flag**
  Flag modifiers. The only possible flag value is `SCSI_HBA_CANSLEEP` (memory allocation may sleep).
- **hba_tran**
  Pointer to a `scsi_hba_tran(9S)` structure.

### DESCRIPTION
**scsi_hba_tran_alloc()**
allocates a `scsi_hba_tran(9S)` structure for a HBA driver. The HBA must use this structure to register its transport vectors with the system by using `scsi_hba_attach_setup(9F)`.

If the flag `SCSI_HBA_CANSLEEP` is set in `flags`, `scsi_hba_tran_alloc()` may sleep when allocating resources; otherwise it may not sleep, and callers should be prepared to deal with allocation failures.

**scsi_hba_tran_free()**
is used to free the `scsi_hba_tran(9S)` structure allocated by `scsi_hba_tran_alloc()`.

### RETURN VALUES
`scsi_hba_tran_alloc()` returns a pointer to the allocated transport structure, or NULL if no space is available.

### CONTEXT
`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.

### SEE ALSO
`scsi_hba_attach_setup(9F), scsi_hba_tran(9S)`
*Writing Device Drivers*

---

9F-510 SunOS 5.6 modified 30 Aug 1995
Kernel Functions for Drivers

NAME  scsi_ifgetcap, scsi_ifsetcap – get/set SCSI transport capability

SYNOPSIS  #include <sys/scsi/scsi.h>
int scsi_ifgetcap(struct scsi_address *ap, char *cap, int whom);
int scsi_ifsetcap(struct scsi_address *ap, char *cap, int value, int whom);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS  ap  Pointer to the scsi_address structure.
cap  Pointer to the string capability identifier.
value  Defines the new state of the capability.
whom  Determines if all targets or only the specified target is affected.

DESCRIPTION  The target drivers use scsi_ifsetcap() to set the capabilities of the host adapter driver. A cap is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using scsi_ifgetcap(). If whom is 0 all targets are affected, else the target specified by the scsi_address structure pointed to by ap is affected.

A device may support only a subset of the capabilities listed below. It is the responsibility of the driver to make sure that these functions are called with a cap supported by the device.

The following capabilities have been defined:

dma-max  Maximum dma transfer size supported by host adapter.
msg-out  Message out capability supported by host adapter: 0 disables, 1 enables.
disconnect  Disconnect capability supported by host adapter: 0 disables, 1 enables.
synchronous  Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.
wide-xfer  Wide transfer capability supported by host adapter: 0 disables, 1 enables.
parity  Parity checking by host adapter: 0 disables, 1 enables.
initiator-id  The host’s bus address is returned.
untagged-qing  The host adapter’s capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.
tagged-qing  The host adapter’s capability to support tagged queueing: 0 disables, 1 enables.
auto-reqsense  The host adapter’s capability to support auto request sense on check conditions: 0 disables, 1 enables.
sector-size  The target driver sets this capability to inform the HBA of the

modified 30 Aug 1995  SunOS 5.6  9F-511
granularity, in bytes, of DMA breakup; the HBA’s DMA limit structure will be set to reflect this limit (see ddi_dma_lim_sparc(9S) or ddi_dma_lim_x86(9S)). It should be set to the physical disk sector size. This capability defaults to 512.

**total-sectors**
The target driver sets this capability to inform the HBA of the total number of sectors on the device, as returned from the SCSI get capacity command. This capability must be set before the target driver “gets” the geometry capability.

**geometry**
This capability returns the HBA geometry of a target disk. The target driver must set the total-sectors capability before “getting” the geometry capability. The geometry is returned as a 32-bit value: the upper 16 bits represent the number of heads per cylinder; the lower 16 bits represent the number of sectors per track. The geometry capability cannot be “set.”

**reset-notification**
The host adapter’s capability to support bus reset notification: 0 disables, 1 enables. Refer to scsi_reset_notify(9F).

**linked -cmds**
The host adapter’s capability to support linked commands: 0 disables, 1 enables.

**qfull-retries**
This capability enables/disables QUEUE FULL handling. If 0, the HBA will not retry a command when a QUEUE FULL status is returned. If greater than 0, then the HBA driver will retry the command at specified number of times at an interval determined by the "qfull-retry-interval". The range for qfull-retries is 0-255.

**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.

**RETURN VALUES**

scsi_ifsetcap() returns:
1 if the capability was successfully set to the new value,
0 if the capability is not variable, and
−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
−1 if the capability was not defined.

**CONTEXT**
These functions can be called from user or interrupt context.

**EXAMPLES**

```c
un->un_arq_enabled = ((scsi_ifsetcap(&devp->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;
}
```

9F-512 SunOS 5.6 modified 30 Aug 1995
} 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;
}

SEE ALSO  
  scsi_reset_notify(9F), ddi_dma_lim_sparc(9S), ddi_dma_lim_x86(9S), scsi_address(9S),
  scsi_arq_status(9S)

Writing Device Drivers
NAME
scsi_init_pkt – prepare a complete SCSI packet

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pktp,
    struct buf *bp, int cmdlen, int statuslen, int privatelen, int flags,
    int (*callback)(caddr_t, caddr_t arg);

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>Pointer to a scsi_address(9S) structure.</td>
</tr>
<tr>
<td>pktp</td>
<td>A pointer to a scsi_pkt(9S) structure.</td>
</tr>
<tr>
<td>bp</td>
<td>Pointer to a buf(9S) structure.</td>
</tr>
<tr>
<td>cmdlen</td>
<td>The required length for the SCSI command descriptor block (CDB) in bytes.</td>
</tr>
<tr>
<td>statuslen</td>
<td>The required length for the SCSI status completion block (SCB) in bytes.</td>
</tr>
<tr>
<td>privatelen</td>
<td>The required length for the pkt_private area.</td>
</tr>
<tr>
<td>flags</td>
<td>Flags modifier.</td>
</tr>
<tr>
<td>callback</td>
<td>A pointer to a callback function, NULL_FUNC, or SLEEP_FUNC.</td>
</tr>
<tr>
<td>arg</td>
<td>The callback function argument.</td>
</tr>
</tbody>
</table>

DESCRIPTION
Target drivers use scsi_init_pkt() to request the transport layer to allocate and initialize a packet for a SCSI command which possibly includes a data transfer. If pktp is NULL, a new scsi_pkt(9S) is allocated using the HBA driver’s packet allocator. The bp is a pointer to a buf(9S) structure. If bp is non-NULL and contains a valid byte count, the buf(9S) structure is also set up for DMA transfer using the HBA driver DMA resources allocator. When bp is allocated by scsi_alloc_consistent_buf(9F), the PKT_CONSISTENT bit must be set in the flags argument to ensure proper operation. If privatelen is non-zero then additional space is allocated for the pkt_private area of the scsi_pkt(9S). On return pkt_private points to this additional space. Otherwise pkt_private is a pointer that is typically used to store the bp during execution of the command. In this case pkt_private is NULL on return.

The flags argument is a set of bit flags. Possible bits include:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKT_CONSISTENT</td>
<td>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.</td>
</tr>
</tbody>
</table>
| PKT_DMA_PARTIAL | This may be set if the driver can accept a partial DMA mapping. If set, scsi_init_pkt() will allocate DMA resources with the DDI_DMA_PARTIAL bit set in the dmar_flag element of the ddi_dma_req(9S) structure. The pkt_resid field of the scsi_pkt(9S) structure may be returned with a non-zero value, which indicates the number of bytes for which scsi_init_pkt() was
unable to allocate DMA resources. In this case, a subsequent call to \texttt{scsi\_init\_pkt()} may be made for the same \texttt{pkt} and \texttt{bp} to adjust the DMA resources to the next portion of the transfer. This sequence should be repeated until the \texttt{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 \texttt{scsi\_init\_pkt()} to move already-allocated DMA resources, the \texttt{cmdlen}, \texttt{statuslen} and \texttt{privatelen} fields are ignored.

The last argument \texttt{arg} is supplied to the \texttt{callback} function when it is invoked. \texttt{callback} indicates what the allocator routines should do when resources are not available:

- \texttt{NULL\_FUNC} Do not wait for resources. Return a \texttt{NULL} pointer.
- \texttt{SLEEP\_FUNC} Wait indefinitely for resources.
- Other Values \texttt{callback} points to a function which is called when resources may have become available. \texttt{callback} \textbf{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, \texttt{scsi\_init\_pkt()} returns the \texttt{scsi\_pkt} field \texttt{pkt\_resid} as the number of residual bytes for which the system was unable to allocate DMA resources. A \texttt{pkt\_resid} of 0 means that all necessary DMA resources were allocated.

\textbf{RETURN VALUES} \texttt{scsi\_init\_pkt()} returns \texttt{NULL} if the packet or dma resources could not be allocated. Otherwise, it returns a pointer to an initialized \texttt{scsi\_pkt(9S)}. If \texttt{pkt} was not \texttt{NULL} the return value will be \texttt{pkt} on successful initialization of the packet.

\textbf{CONTEXT} If \texttt{callback} is \texttt{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 \texttt{callback} function may not block or call routines that block.

\textbf{EXAMPLES} To allocate a packet without DMA resources attached, use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, NULL, CDB\_GROUP1,
           STATUS\_LEN, sizeof (struct \_pkt\_private *), 0,
           sd_runout, sd\_unit);
```

To allocate a packet with DMA resources attached use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB\_GROUP1,
           STATUS\_LEN, 0, 0, NULL\_FUNC, NULL);
```

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);
```
Since the packet is already allocated the cmdlen, statuslen and privatelen are 0.

To allocate a packet with consistent DMA resources attached, use:

```c
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
   SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
   STATUS_LEN, sizeof (struct my_pkt_private *), PKT_CONSISTENT,
   SLEEP_FUNC, NULL);
```

To allocate a packet with partial DMA resources attached, use:

```c
my_pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
   STATUS_LEN, sizeof (struct buf *), PKT_DMA_PARTIAL,
   SLEEP_FUNC, NULL);
```

SEE ALSO `scsi_alloc_consistent_buf(9F)`, `scsi_destroy_pkt(9F)`, `scsi_dmaget(9F)`, `scsi_pktalloc(9F)`, `buf(9S)`, `ddi_dma_req(9S)`, `scsi_address(9S)`, `scsi_pkt(9S)`

Writing Device Drivers

NOTES If a DMA allocation request fails with DDI_DMA_NOMAPPING, the B_ERROR flag will be set in bp, and the b_error field will be set to EFAULT.

If a DMA allocation request fails with DDI_DMA_TOOBIG, the B_ERROR flag will be set in bp, and the b_error field will be set to EINVAL.
NAME

scsi_log – display a SCSI-device-related message

SYNOPSIS

```c
#include <sys/scsi/scsi.h>
#include <sys/cmn_err.h>

void scsi_log(dev_info_t *dip, char *drv_name, u_int level, const char *fmt, ...);
```

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to the dev_info structure.
drv_name String naming the device.
level Error level.
fmt Display format.

DESCRIPTION

scsi_log() is a utility function that displays a message via the cmn_err(9F) routine. The error levels that can be passed in to this function are CE_PANIC, CE_WARN, CE_NOTE, CE_CONT, and SCSI_DEBUG. The last level is used to assist in displaying debug messages to the console only. drv_name is the short name by which this device is known; example disk driver names are sd and cmdk. If the dev_info_t pointer is NULL, then the drv_name will be used with no unit or long name.

If the first character in format is an ‘!’ (exclamation point), the message goes only to the system buffer. If the first character in format is a ‘^’ (circumflex), the message goes only to the console. If the first character is a ‘?’ (question mark), and level is CE_CONT, the message is always sent to the system buffer, but is only written to the console when the system has been booted in verbose mode. See kernel(1M). If neither condition is met, the ‘?’ character has no effect and is simply ignored.

All formatting conversions in use by cmn_err() also work with scsi_log().

CONTEXT

scsi_log() may be called from user or interrupt context.

EXAMPLES

a. 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.

b. scsi_log(dev, "sd", CE_WARN, "Label Bad\n");
   Generates:
   WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1): Label Bad

c. scsi_log((dev_info_t *) NULL, "Disk Unit ", CE_NOTE, "Disk Ejected\n");
   Generates:
   Disk Unit: Disk Ejected

modified 7 Jun 1993 SunOS 5.6 9F-517
d. `scsi_log(cmdk_unit, "Disk Unit ", CE_CONT, "Disk Inserted\n");`
   Generates:
   
   Disk Inserted

e. `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`
NAME
scsi_pktalloc, scsi_resalloc, scsi_pktfree, scsi_resfree – SCSI packet utility routines

SYNOPSIS
#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_pktalloc(struct scsi_address *ap, int cmdlen, int statuslen,
   int (*callback)(void));

struct scsi_pkt *scsi_resalloc(struct scsi_address *ap, int cmdlen, int statuslen,
   opaque_t dmatoken, int (*callback)(void));

void scsi_pktfree(struct scsi_pkt *pkt);

void scsi_resfree(struct scsi_pkt *pkt);

INTERFACE
Solaris DDI specific (Solaris DDI).

LEVEL

ARGUMENTS
ap Pointer to a scsi_address structure.

cmdlen The required length for the SCSI command descriptor block (CDB) in bytes.

statuslen The required length for the SCSI status completion block (SCB) in bytes.

dmatoken Pointer to an implementation-dependent object.

callback A pointer to a callback function, or NULL_FUNC or SLEEP_FUNC.

pkt Pointer to a scsi_pkt(9S) structure.

DESCRIPTION
scsi_pktalloc() requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, scsi_resalloc() should be used.

ap is a pointer to a scsi_address structure. Allocator routines use it to determine the associated host adapter.

cmdlen is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the pkt_cdbp field of the allocated scsi_pkt structure.

statuslen is the required length for the SCSI status completion block. The address of the allocated block is placed into the pkt_scbp field of the scsi_pkt structure.

dmatoken is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The dmatoken must be a pointer to a buf(9S) structure. If dmatoken is NULL, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data transfer phase (either no data transfer phase expected, or the wrong direction encountered), the command is terminated with the pkt_reason set to CMD_DMA_DERR. dmatoken provides the information to determine if the transfer count is correct.
callback indicates what the allocator routines should do when resources are not available:

- **NULL_FUNC**: Do not wait for resources. Return a NULL pointer.
- **SLEEP_FUNC**: Wait indefinitely for resources.

*Other Values* **callback** points to a function which is called when resources may have become available. **callback** must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

**scsi_pktfree()** frees the packet.

**scsi_resfree()** free all resources held by the packet and the packet itself.

**RETURN VALUES** Both allocation routines return a pointer to a scsi_pkt structure on success, or NULL on failure.

**CONTEXT** If callback is SLEEP_FUNC, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The callback function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

**SEE ALSO** scsi_dmafree(9F), scsi_dmaget(9F), buf(9S), scsi_pkt(9S)

*Writing Device Drivers*
**NAME**

`scci_poll` - run a polled SCSI command on behalf of a target driver

**SYNOPSIS**

```c
#include <sys/scsi/scsi.h>

int scsi_poll(struct scsi_pkt *pkt);
```

**INTERFACE LEVEL**

Solaris DDI specific (Solaris DDI).

**ARGUMENTS**

`pkt` Pointer to the `scsi_pkt(9S)` structure.

**DESCRIPTION**

`scci_poll()` requests the host adapter driver to run a polled command. Unlike `scsi_transport(9F)` which runs commands asynchronously, `scci_poll()` runs commands to completion before returning. If the `pkt_time` member of `pkt` is 0, the value of `pkt_time` is defaulted to `SCSI_POLL_TIMEOUT` to prevent an indefinite hang of the system.

**RETURN VALUES**

`scci_poll()` returns:

- 0 command completed successfully.
- -1 command failed.

**CONTEXT**

`scci_poll()` can be called from user or interrupt level.

**SEE ALSO**

`makecom(9F), scsi_transport(9F), scsi_pkt(9S)`

*Writing Device Drivers*

**WARNINGS**

Since `scci_poll()` runs commands to completion before returning, it may require more time than is desirable when called from interrupt context. Therefore, calling `scci_poll` from interrupt context is not recommended.
NAME
scsi_probe – utility for probing a scsi device

SYNOPSIS
#include <sys/scsi/scsi.h>
int scsi_probe(struct scsi_device *devp, int (*waitfunc)());

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
devp Pointer to a scsi_device(9S) structure
waitfunc NULL_FUNC or SLEEP_FUNC

DESCRIPTION
scsi_probe() determines whether a target/lun is present and sets up the scsi_device structure with inquiry data.

scsi_probe() uses the SCSI Inquiry command to test if the device exists. It may retry the Inquiry command as appropriate. If scsi_probe() is successful, it will allocate space for the scsi_inquiry structure and assign the address to the sd_inq member of the scsi_device(9S) structure. scsi_probe() will then fill in this scsi_inquiry(9S) structure and return SCSIPROBE_EXISTS.

scsi_unprobe(9F) is used to undo the effect of scsi_probe().
If the target is a non-CCS device, SCSIPROBE_NONCCS will be returned.

waitfunc indicates what the allocator routines should do when resources are not available; the valid values are:

NULL_FUNC Do not wait for resources. Return SCSIPROBE_NOMEM or SCSIPROBE_FAILURE
SLEEP_FUNC Wait indefinitely for resources.

RETURN VALUES
scsi_probe() returns:
SCSIPROBE_BUSY Device exists but is currently busy.
SCSIPROBE_EXISTS Device exists and inquiry data is valid.
SCSIPROBE_FAILURE Polled command failure.
SCSIPROBE_NOMEM No space available for structures.
SCSIPROBE_NONCCS Device exists but inquiry data is not valid.
SCSIPROBE_NORESP Device does not respond to an INQUIRY.

CONTEXT
scsi_probe() is normally called from the target driver’s probe(9E) or attach(9E) routine.
If waitfunc 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.
EXAMPLE

```c
switch (scsi_probe(devp, NULL_FUNC)) {
default:
    case SCSIPROBE_NORESP:
    case SCSIPROBE_NONCCS:
    case SCSIPROBE_NOMEM:
    case SCSIPROBE_FAILURE:
    case SCSIPROBE_BUSY:
        break;

    case SCSIPROBE_EXISTS:
        switch (devp->sd_inq->inq_dtype) {
            case DTYPE_DIRECT:
                rval = DDI_PROBE_SUCCESS;
                break;
            case DTYPE_RODIRECT:
                rval = DDI_PROBE_SUCCESS;
                break;
            case DTYPE_NOTPRESENT:
                default:
                    break;
                
        }
    
        scsi_unprobe(devp);
```

SEE ALSO `attach(9E)`, `probe(9E)`, `scsi_slave(9F)`, `scsi_unprobe(9F)`, `scsi_unslave(9F)`, `scsi_device(9S)`, `scsi_inquiry(9S)`

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

NOTES A `waitfunc` function other than NULL_FUNC or SLEEP_FUNC is not supported and may have unexpected results.
### NAME
`scsi_reset` – reset a SCSI bus or target

### SYNOPSIS
```c
#include <sys/scsi/scsi.h>
int scsi_reset(struct scsi_address *ap, int level);
```

### INTERFACE
Solaris DDI specific (Solaris DDI).

### ARGUMENTS
- **ap**: Pointer to the `scsi_address` structure.
- **level**: The level of reset required.

### DESCRIPTION
`scsi_reset()` asks the host adapter driver to reset the SCSI bus or a SCSI target as specified by `level`. If `level` equals `RESET_ALL`, the SCSI bus is reset. If it equals `RESET_TARGET`, `ap` is used to determine the target to be reset.

On a successful reset, the `pkt_reason` is set to `CMD_RESET` and `pkt_statistics` is OR’ed with `STAT_BUS_RESET` or `STAT_DEV_RESET`.

### RETURN VALUES
`scsi_reset()` returns:
- **1**: on success.
- **0**: on failure.

### CONTEXT
`scsi_reset()` can be called from user or interrupt context.

### SEE ALSO
- `tran_reset(9E)`, `tran_reset_notify(9E)`, `scsi_abort(9F)`
- *Writing Device Drivers*
NAME

scsi_reset_notify – notify target driver of bus resets

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_reset_notify(struct scsi_address *ap, int flag, 
    void (*callback)(caddr_t, caddr_t arg);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ap Pointer to the scsi_address structure.
flag A flag indicating registration or cancellation of the notification request.
callback A pointer to the target driver’s reset notification function.
arg The callback function argument.

DESCRIPTION

scsi_reset_notify( ) is used by a target driver when it needs to be notified of a bus reset. The bus reset could be issued by the transport layer (e.g., the host bus adapter (HBA) driver or controller) or by another initiator.

The argument flag is used to register or cancel the notification. The supported values for flag are as follows:

- SCSI_RESET_NOTIFY Register callback as the reset notification function for the target driver.
- SCSI_RESET_CANCEL Cancel the reset notification request.

Target drivers can find out whether the HBA driver and controller support reset notification by checking the reset-notification capability using the scsi_ifgetcap(9F) function.

RETURN VALUES

If flag is SCSI_RESET_NOTIFY, scsi_reset_notify( ) returns:
- DDI_SUCCESS the notification request has been accepted.
- DDI_FAILURE the transport layer does not support reset notification or could not accept this request.

If flag is SCSI_RESET_CANCEL, scsi_reset_notify( ) returns:
- DDI_SUCCESS the notification request has been canceled.
- DDI_FAILURE no notification request was registered.

CONTEXT

scsi_reset_notify( ) can be called from user or interrupt context.

SEE ALSO

scsi_address(9S), scsi_ifgetcap(9F)

Writing Device Drivers

modified 8 Sep 1994 SunOS 5.6 9F-525
NAME  
scsi_slave – utility for SCSI target drivers to establish the presence of a target

SYNOPSIS  
#include <sys/scsi/scsi.h>

int scsi_slave(struct scsi_device *devp, int (*callback)(void));

INTERFACE  
LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
devp     Pointer to a scsi_device(9S) structure.
callback Pointer to a callback function, NULL_FUNC or SLEEP_FUNC.

DESCRIPTION  
scsi_slave() checks for the presence of a SCSI device. Target drivers may use this function in their probe(9E) routines. scsi_slave() determines if the device is present by using a Test Unit Ready command followed by an Inquiry command. If scsi_slave() is successful, it will fill in the scsi_inquiry structure, which is the sd_inq member of the scsi_device(9S) structure, and return SCSI_PROBE_EXISTS. This information can be used to determine if the target driver has probed the correct SCSI device type. callback indicates what the allocator routines should do when DMA resources are not available:

NULL_FUNC    Do not wait for resources. Return a NULL pointer.
SLEEP_FUNC   Wait indefinitely for resources.

Other Values  callback points to a function which is called when resources may have become available. callback must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

RETURN VALUES  
scsi_slave() returns:

SCSI_PROBE_NOMEM No space available for structures.
SCSI_PROBE_EXISTS Device exists and inquiry data is valid.
SCSI_PROBE_NONCCS Device exists but inquiry data is not valid.
SCSI_PROBE_FAILURE Polled command failure.
SCSI_PROBE_NORESP No response to TEST UNIT READY.

CONTEXT  
scsi_slave() is normally called from the target driver’s probe(9E) or attach(9E) routine. 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.

SEE ALSO  
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
NAME  
scsi_sync_pkt – synchronize CPU and I/O views of memory

SYNOPSIS  
#include <sys/scsi/scsi.h>

void scsi_sync_pkt(struct scsi_pkt *pktp);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
pktp  
pointer to a scsi_pkt(9S) structure.

DESCRIPTION  
scsi_sync_pkt() is used to selectively synchronize a CPU’s or device’s view of the data associated with the SCSI packet that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When a SCSI packet is mapped for I/O using scsi_init_pkt(9F) and destroyed using scsi_destroy_pkt(9F), then an implicit scsi_sync_pkt() will be performed. However, if the memory object has been modified by either the device or a CPU after the mapping by scsi_init_pkt(9F), then a call to scsi_sync_pkt() is required.

CONTEXT  
scsi_sync_pkt() may be called from user or interrupt context.

EXAMPLES  
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.

SEE ALSO  
tran_sync_pkt(9E), ddi_dma_sync(9F), scsi_destroy_pkt(9F), scsi_init_pkt(9F), scsi_transport(9F), scsi_pkt(9S)

Writing Device Drivers
**NAME**
scsi_transport – request by a SCSI target driver to start a command

**SYNOPSIS**
```c
#include <sys/scsi/scsi.h>
int scsi_transport(struct scsi_pkt *pkt);
```

**INTERFACE LEVEL**
Solaris DDI specific (Solaris DDI).

**ARGUMENTS**
pkt Pointer to a scsi_pkt(9S) structure.

**DESCRIPTION**
Target drivers use `scsi_transport()` to request the host adapter driver to transport a command to the SCSI target device specified by `pkt`. The target driver must obtain resources for the packet using `scsi_init_pkt(9F)` prior to calling this function. The packet may be initialized using one of the `makecom(9F)` functions. `scsi_transport()` does not wait for the SCSI command to complete. See `scsi_poll(9F)` for a description of polled SCSI commands. Upon completion of the SCSI command the host adapter calls the completion routine provided by the target driver in the `pkt_comp` member of the `scsi_pkt` pointed to by `pkt`.

**RETURN VALUES**
`scsi_transport()` returns:
- **TRAN_ACCEPT** The packet was accepted by the transport layer.
- **TRAN_BUSY** The packet could not be accepted because there was already a packet in progress for this target/lun, the host adapter queue was full, or the target device queue was full.
- **TRAN_BADPKT** The DMA count in the packet exceeded the DMA engine’s maximum DMA size.
- **TRAN_FATAL_ERROR** A fatal error has occurred in the transport layer.

**CONTEXT**
`scsi_transport()` can be called from user or interrupt context.

**EXAMPLE**
```c
if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {
    scsi_log(devp, sd_label, CE_WARN,
             "transport of request sense pkt fails (0x%x)\n", status);
}
```

**SEE ALSO**
tran_start(9E), makecom(9F), scsi_init_pkt(9F), scsi_pktalloc(9F), scsi_poll(9F), scsi_pkt(9S)

*Writing Device Drivers*
| NAME | scsi_unprobe, scsi_unslave – free resources allocated during initial probing |
| SYNOPSIS | 
```c
#include <sys/scsi/scsi.h>
void scsi_unslave(struct scsi_device *devp);
void scsi_unprobe(struct scsi_device *devp);
```
| INTERFACE LEVEL | Solaris DDI specific (Solaris DDI). |
| ARGUMENTS | devp Pointer to a scsi_device(9S) structure. |
| DESCRIPTION | scsi_unprobe() and scsi_unslave() are used to free any resources that were allocated on the driver's behalf during scsi_slave(9F) and scsi_probe(9F) activity. |
| CONTEXT | scsi_unprobe() and scsi_unslave() may be called from either the user or the interrupt levels. |
| SEE ALSO | scsi_probe(9F), scsi_slave(9F), scsi_device(9S) |

Writing Device Drivers

modified 21 Dec 1992 SunOS 5.6 9F-529
NAME
semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

SYNOPSIS
#include <sys/ksynch.h>

void sema_init(ksema_t *sp, u_int val, char *name, ksema_type_t type, void *arg);
void sema_destroy(ksema_t *sp);
void sema_p(ksema_t *sp);
void sema_v(ksema_t *sp);
int sema_p_sig(ksema_t *sp);
int sema_tryp(ksema_t *sp);

INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS

sp A pointer to a semaphore, type ksema_t.
val Initial value for semaphore.
name Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they’re a waste of kernel memory.)
type Variant type of the semaphore. Currently only SEMA_DRIVER is supported.
arg Type-specific argument, should be NULL.

DESCRIPTION
These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by sema_p() and atomically incremented by sema_v(). The value must always be greater than or equal to zero. If sema_p() is called and the value is zero, the calling thread is blocked until another thread performs a sema_v() operation on the semaphore.

Semaphores are initialized by calling sema_init(). The argument, val, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by sema_init(). For this reason, sema_destroy() should be called before deallocating the storage containing the semaphore.

sema_p_sig() decrements the semaphore, as does sema_p(), however, if the semaphore value is zero, sema_p_sig() will return without decrementing the value if a signal (e.g. from kill(2)) is pending for the thread.

sema_tryp() will decrement the semaphore value only if it is greater than zero, and will not block.

RETURN VALUES
0 sema_tryp() could not decrement the semaphore value because it was zero.
1 sema_p_sig() was not able to decrement the semaphore value and detected a pending signal.
CONTEXT

These functions can be called from user or interrupt context, except for `sema_init()` and `sema_destroy()`, which can be called from user context only. None of these functions can be called from a high-level interrupt context. In most cases, `sema_v()` and `sema_p()` should not be called from any interrupt context.

If `sema_p()` is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the `sema_v()` becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the `sema_v()` may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

SEE ALSO

`kill(2), condvar(9F), mutex(9F)`

*Writing Device Drivers*
NAME  sprintf – format characters in memory

SYNOPSIS  
```
#include <sys/ddi.h>

char *sprintf(char *buf, const char *fmt, ...);
```

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).

ARGUMENTS  
- **buf**: Pointer to a character string.
- **fmt**: Pointer to a character string.

DESCRIPTION  sprintf() builds a string in **buf** under the control of the format **fmt**. The format is a character string with either plain characters, which are simply copied into **buf**, or conversion specifications, each of which converts zero or more arguments, again copied into **buf**. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user’s responsibility to ensure that enough storage is available for **buf**.

Each conversion specification is introduced by the % character, after which the following appear in sequence:

- An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.
- An optional **l** (**ll**) specifying that a following **d**, **D**, **o**, **O**, **x**, **X**, or **u** conversion character applies to a **long** (**long long**) integer argument. An **l** (**ll**) before any other conversion character is ignored.
- A character indicating the type of conversion to be applied:
  - **d**, **D**, **o**, **O**, **x**, **X**, **u**: The integer argument is converted to signed decimal (**d**, **D**), unsigned octal (**o**, **O**), unsigned hexadecimal (**x**, **X**) or unsigned decimal (**u**), respectively, and copied. The letters **abcdef** are used for **x** and **X** conversion.
  - **c**: The character value of argument is copied.
  - **b**: This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form `<base>[<arg>...]. The base supplies the conversion base for the first argument as a binary value; \10 gives octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.
A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by < and >. Thus, the following function call would generate

```
reg=3<BitTwo,BitOne>
```
in `buf`.

```c
sprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```

The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is `NULL`, the string `<null string>` is used in its place.

```
% Copy a %; no argument is converted.
```

**RETURN VALUES**

`sprintf()` returns its first argument, `buf`.

**CONTEXT**

`sprintf()` can be called from user or interrupt context.

**SEE ALSO**

`Writing Device Drivers`
### NAME
stoi, numtos – convert between an integer and a decimal string

### SYNOPSIS
```c
#include <sys/ddi.h>

int stoi(char **str);
void numtos(unsigned long num, char *s);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI).

### ARGUMENTS
- **str**: Pointer to a character string to be converted.
- **num**: Decimal number to be converted to a character string.
- **s**: Character buffer to hold converted decimal number.

### DESCRIPTION
- **stoi()**: Returns the integer value of a string of decimal numeric characters beginning at `**str`. No overflow checking is done. `*str` is updated to point at the last character examined.

- **numtos()**: Converts a `long` into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.

### RETURN VALUES
- **stoi()**: Returns the integer value of the string `str`.

### CONTEXT
**stoi()** can be called from user or interrupt context.

### SEE ALSO
*Writing Device Drivers*

### NOTES
stoi() handles only positive integers; it does not handle leading minus signs.
<table>
<thead>
<tr>
<th>NAME</th>
<th>strchr – find a character in a string</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td><code>#include &lt;sys/ddi.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>#include &lt;sys/sunddi.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>char *strchr(const char *str, int chr);</code></td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Solaris DDI specific (Solaris DDI).</td>
</tr>
<tr>
<td>LEVEL ARGUMENTS</td>
<td><code>str</code> Pointer to a string to be searched.</td>
</tr>
<tr>
<td></td>
<td><code>chr</code> The character to search for.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td><code>strchr()</code> returns a pointer to the first occurrence of <code>chr</code> in the string pointed to by <code>str</code>.</td>
</tr>
<tr>
<td>RETURN VALUES</td>
<td><code>strchr()</code> returns a pointer to a character, or NULL, if the search fails.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>This function can be called from user or interrupt context.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td><code>strcmp(9F)</code></td>
</tr>
<tr>
<td></td>
<td><em>Writing Device Drivers</em></td>
</tr>
</tbody>
</table>
**NAME**  
strcmp, strncmp – compare two null terminated strings.

**SYNOPSIS**  
```
#include <sys/ddi.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

**INTERFACE LEVEL**  
Solaris DDI specific (Solaris DDI).

**ARGUMENTS**  
s1, s2  
Pointers to character strings.

n  
Count of characters to be compared.

**DESCRIPTION**  
*strcmp()*  
`strcmp()` returns 0 if the strings are the same, or the integer value of the expression (`*s1 - *s2`) for the last characters compared if they differ.

*strncmp()*  
`strncmp()` returns 0 if the first *n* characters of *s1* and *s2* are the same, or (`*s1 - *s2`) for the last characters compared if they differ.

**RETURN VALUES**  
*strcmp()* returns 0 if the strings are the same, or (`*s1 - *s2`) for the last characters compared if they differ.

*strncmp()* returns 0 if the first *n* characters of strings are the same, or (`*s1 - *s2`) for the last characters compared if they differ.

**CONTEXT**  
These functions can be called from user or interrupt context.

**SEE ALSO**  
*Writing Device Drivers*
NAME

strcpy, strncpy – copy a string from one location to another.

SYNOPSIS

```c
#include <sys/ddi.h>

char *strcpy(char *dst, char *srs);
char *strncpy(char *dst, char *srs, size_t n);
```

INTERFACE

Solaris DDI specific (Solaris DDI).

ARGUMENTS

- `dst`, `srs`  Pointers to character strings.
- `n`  Count of characters to be copied.

DESCRIPTION

`strcpy()` copies characters in the string `srs` to `dst`, terminating at the first null character in `srs`, and returns `dst` to the caller. No bounds checking is done.

`strncpy()` copies `srs` to `dst`, null-padding or truncating at `n` bytes, and returns `dst`. No bounds checking is done.

RETURN VALUES

`strcpy()`, and `strncpy()` return `dst`.

CONTEXT

`strcpy()` can be called from user or interrupt context.

SEE ALSO

*Writing Device Drivers*
**NAME**  
strlen – determine the number of non-null bytes in a string

**SYNOPSIS**  
```c
#include <sys/ddi.h>

size_t strlen(const char *s);
```

**INTERFACE LEVEL**  
Solaris DDI specific (Solaris DDI).

**ARGUMENTS**  
- `s`  
  Pointer to a character string.

**DESCRIPTION**  
`strlen()` returns the number of non-null bytes in the string argument `s`.

**RETURN VALUES**  
`strlen()` returns the number of non-null bytes in `s`.

**CONTEXT**  
`strlen()` can be called from user or interrupt context.

**SEE ALSO**  
*Writing Device Drivers*
NAME
strlog – submit messages to the log driver

SYNOPSIS
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/log.h>

int strlog(short mid, short sid, char level, unsigned short flags, char *fmt, ...);

INTERFACE
Architecture independent level 1 (DDI/DKI).

LEVEL
ARGUMENTS
mid Identification number of the module or driver submitting the message (in the case of a module, its mi_idnum value from module_info(9S)).
sid Identification number for a particular minor device.
level Tracing level for selective screening of low priority messages. Larger values imply less important information.
flags Valid flag values are:
  SL_ERROR Message is for error logger.
  SL_TRACE Message is for trace.
  SL_NOTIFY Mail copy of message to system administrator.
  SL_CONSOLE Log message to console.
  SL_FATAL Error is fatal.
  SL_WARN Error is a warning.
  SL_NOTE Error is a notice.
fmt printf(3S) style format string. %s, %e, %g, and %G formats are not allowed.

DESCRIPTION
strlog() submits formatted messages to the log(7D) driver. The messages can be retrieved with the getmsg(2) system call. The flags argument specifies the type of the message and where it is to be sent. strace(1M) receives messages from the log driver and sends them to the standard output. strerr(1M) receives error messages from the log driver and appends them to a file called /var/adm/streams/error.mm-dd, where mm-dd identifies the date of the error message.

RETURN VALUES
strlog() returns 0 if the message is not seen by all the readers, 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

modified 11 Apr 1991 SunOS 5.6

9F-539
# NAME
strqget – get information about a queue or band of the queue

# SYNOPSIS
```c
#include <sys/stream.h>

int strqget (queue_t *q, qfields_t what, unsigned char pri, void *valp);
```

# INTERFACE LEVEL ARGUMENTS
- **q**: Pointer to the queue.
- **what**: Field of the `queue` structure for (or the specified priority band) to return information about. Valid values are one of:
  - `QHIWAT`: High water mark.
  - `QLOWAT`: Low water mark.
  - `QMAXPSZ`: Largest packet accepted.
  - `QMINPSZ`: Smallest packet accepted.
  - `QCOUNT`: Approximate size (in bytes) of data.
  - `QFIRST`: First message.
  - `QLAST`: Last message.
  - `QFLAG`: Status.
- **pri**: Priority band of interest.
- **valp**: The address of where to store the value of the requested field.

# DESCRIPTION
`strqget()` gives drivers and modules a way to get information about a queue or a particular band of a queue without directly accessing STREAMS data structures, thus insulating them from changes in the implementation of these data structures from release to release.

# RETURN VALUES
On success, 0 is returned and the value of the requested field is stored in the location pointed to by `valp`. An error number is returned on failure.

# CONTEXT
`strqget()` can be called from user or interrupt context.

# SEE ALSO
- `freezestr(9F)`, `strqset(9F)`, `unfreezestr(9F)`, `queue(9S)`

Writing Device Drivers
STREAMS Programming Guide

# NOTES
The stream must be frozen using `freezestr(9F)` before calling `strqget()`.
NAME
strqset – change information about a queue or band of the queue

SYNOPSIS
#include <sys/stream.h>

int strqset(queue_t *q, qfields_t what, unsigned char pri, intptr_t val);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
q Pointer to the queue.
what Field of the queue structure (or the specified priority band) to return information about. Valid values are one of:
   QHIWAT High water mark.
   QLOWAT Low water mark.
   QMAXPSZ Largest packet accepted.
   QMINPSZ Smallest packet accepted.
pri Priority band of interest.
val The value for the field to be changed.

DESCRIPTION
strqset() gives drivers and modules a way to change information about a queue or a particular band of a queue without directly accessing STREAMS data structures.

RETURN VALUES
On success, 0 is returned. EINVAL is returned if an undefined attribute is specified.

CONTEXT
strqset() can be called from user or interrupt context.

SEE ALSO
freezestr(9F), strqget(9F), unfreezestr(9F), queue(9S)
Writing Device Drivers
STREAMS Programming Guide

NOTES
The stream must be frozen using freezestr(9F) before calling strqset().
To set the values of QMINPSZ and QMAXPSZ from within a single call to freezestr(9F) and unfreezestr(9F): when lowering the existing values, set QMINPSZ before setting QMAXPSZ; when raising the existing values, set QMAXPSZ before setting QMINPSZ.
NAME  swab – swap bytes in 16-bit halfwords

SYNOPSIS  
#include <sys/sunddi.h>

void swab ( void *src, void *dst, size_t nbytes);

INTERFACE LEVEL  Architecture independent level 1 (DDI/DKI).

ARGUMENTS  
src  A pointer to the buffer containing the bytes to be swapped.

dst  A pointer to the destination buffer where the swapped bytes will be written. If 
     dst is the same as src the buffer will be swapped in place.

nbytes  Number of bytes to be swapped, rounded down to the nearest half-word.

DESCRIPTION  swab() copies the bytes in the buffer pointed to by src to the buffer pointer to by dst, 
swapping the order of adjacent bytes in half-word pairs as the copy proceeds. A total of 
nbytes bytes are copied, rounded down to the nearest half-word.

CONTEXT  swab() can be called from user or interrupt context.

SEE ALSO  Writing Device Drivers

NOTES  Since swab() operates byte-by-byte, it can be used on non-aligned buffers.
NAME  testb – check for an available buffer

SYNOPSIS  
#include <sys/stream.h>
int testb(size_t size, uint pri);

INTERFACE LEVEL  Architecture independent level 1 (DDI/DKI).

ARGUMENTS  
size  Size of the requested buffer.

pri  Priority of the allocb request.

DESCRIPTION  testb() checks to see if an allocb(9F) call is likely to succeed if a buffer of size bytes at priority pri is requested. Even if testb() returns successfully, the call to allocb (9F) can fail. The pri argument is no longer used, but is retained for compatibility.

RETURN VALUES  Returns 1 if a buffer of the requested size is available, and 0 if one is not.

CONTEXT  testb() can be called from user or interrupt context.

EXAMPLES  In a service routine, if copymsg(9F) fails (line 6), the message is put back on the queue (line 7) and a routine, tryagain, is scheduled to be run in one tenth of a second. Then the service routine returns.

When the timeout(9F) function runs, if there is no message on the front of the queue, it just returns. Otherwise, for each message block in the first message, check to see if an allocation would succeed. If the number of message blocks equals the number we can allocate, then enable the service procedure. Otherwise, reschedule tryagain to run again in another tenth of a second. Note that tryagain is merely an approximation. Its accounting may be faulty. Consider the case of a message comprised of two 1024-byte message blocks. If there is only one free 1024-byte message block and no free 2048-byte message blocks, then testb() will still succeed twice. If no message blocks are freed of these sizes before the service procedure runs again, then the copymsg(9F) will still fail. The reason testb() is used here is because it is significantly faster than calling copymsg. We must minimize the amount of time spent in a timeout routine.

```c
1  xxsrv(q)
2  queue_t *q;
3  {
4      mblk_t *mp;
5      mblk_t *nmp;
6      ...
7      if ((nmp = copymsg(mp)) == NULL) {
8          putbq(q, mp);
9          timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
10         return;
11      }
12      ...
```

modified 11 Nov 1996  SunOS 5.6  9F-543
tryagain(q)
queue_t *q;

register int can_alloc = 0;
register int num_blks = 0;
register mblk_t *mp;

if (!q->q_first)
    return;
for (mp = q->q_first; mp; mp = mp->b_cont) {
    num_blks++;
    can_alloc += testb((mp->b_datap->db_lim -
                      mp->b_datap->db_base), BPRI_MED);
}
if (num_blks == can_alloc)
    qenable(q);
else
    timeout(tryagain, (intptr_t)q, drv_usec_to_hz(100000));

SEE ALSO allocb(9F), bufcall(9F), copymsg(9F), timeout(9F)

Writing Device Drivers
STREAMS Programming Guide

NOTES The pri argument is provided for compatibility only. Its value is ignored.
NAME  timeout – execute a function after a specified length of time

SYNOPSIS  
#include <sys/types.h>
#include <sys/conf.h>

int timeout(void (*func)(), caddr_t arg, clock_t ticks);

INTERFACE LEVEL  Architecture independent level 1 (DDI/DKI).

ARGUMENTS  
func  Kernel function to invoke when the time increment expires.
arg  Argument to the function.
ticks  Number of clock ticks to wait before the function is called.

DESCRIPTION  The timeout() function schedules the specified function to be called after a specified time interval. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The function called by timeout() must adhere to the same restrictions as a driver soft interrupt handler.

The timeout() function returns an identifier that may be passed to the untimeout(9F) function to cancel a pending request.

timeout() can be called from user or interrupt context.

The function called by timeout() is run in interrupt context and must not sleep or call other functions which may sleep.

RETURN VALUES  Under normal conditions, timeout() returns an integer timeout identifier not equal to zero. If, however, the timeout table is full, the system will panic with the following panic message:

PANIC: Timeout table overflow

CONTEXT  timeout() can be called from user or interrupt context.

EXAMPLES  In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

static void
xxtimeout_handler(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->lock);
    cv_signal(&xsp->cv);
    xsp->flags |= TIMED_OUT;
    mutex_exit(&xsp->lock);
}

modified 19 Sep 1996

SunOS 5.6

9F-545
timeout (9F)

Kernel Functions for Drivers

```c

static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    ...
    mutex_enter(&xsp->lock);
    /* Service interrupt */
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->lock);
    if (xsp->timeout_id != 0) {
        (void) untimeout(xsp->timeout_id);
        xsp->timeout_id = 0;
    }
    return(DDI_INTRCLAIMED);
}

static void
xxcheckcond(struct xxstate *xsp)
{
    ...
    xsp->timeout_id = timeout(xxtimeout_handler,
                                (caddr_t)xsp, 5 * hd_drv_usec_to_hz(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), delay(9F), untimeout(9F)

Writing Device Drivers
uiomove (9F) Kernel Functions for Drivers

NAME

uiomove — copy kernel data using uio structure

SYNOPSIS

#include <sys/types.h>
#include <sys/uio.h>

int uiomove(caddr_t address, size_t nbytes, enum uio_rw rwflag, uio_t *uio_p);

INTERFACE

Architecture independent level 1 (DDI/DKI).

ARGUMENTS

address  Source/destination kernel address of the copy.
nbytes   Number of bytes to copy.
rwflag   Flag indicating read or write operation. Possible values are UIO_READ and UIO_WRITE.
uio_p    Pointer to the uio structure for the copy.

DESCRIPTION

The uiomove() function copies nbytes of data to or from the space defined by the uio structure (described in uio(9S)) and the driver.

The uio_segflag member of the uio(9S) structure determines the type of space to or from which the transfer is being made. If it is set to UIO_SYSSPACE, the data transfer is between addresses in the kernel. If it is set to UIO_USERSPACE, the transfer is between a user program and kernel space.

rwflag indicates the direction of the transfer. If UIO_READ is set, the data will be transferred from address to the buffer(s) described by uio_p. If UIO_WRITE is set, the data will be transferred from the buffer(s) described by uio_p to address.

In addition to moving the data, uiomove() adds the number of bytes moved to the iov_base member of the iovec(9S) structure, decreases the iov_len member, increases the uio_offset member of the uio(9S) structure, and decreases the uio_resid member.

This function automatically handles page faults. nbytes does not have to be word-aligned.

RETURN VALUES

uiomove() returns 0 upon success or EFAULT on failure.

CONTEXT

User context only, if uio_segflag is set to UIO_USERSPACE. User or interrupt context, if uio_segflag is set to UIO_SYSSPACE.

SEE ALSO

ureadc(9F), uwritec(9F), iovec(9S), uio(9S)
Writing Device Drivers

WARNINGS

If uio_segflag is set to UIO_SYSSPACE and address is selected from user space, the system may panic.
NAME
unbufcall – cancel a pending bufcall request

SYNOPSIS
#include <sys/stream.h>
void unbufcall(int id);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
id Identifier returned from bufcall(9F) or esbbcall(9F)

DESCRIPTION
unbufcall cancels a pending bufcall() or esbbcall() request. The argument id is a non-
zero identifier for the request to be cancelled. id is returned from the bufcall() or esbb-
call() 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 unbuf-
call() or deadlock may result.

RETURN VALUES
None.

CONTEXT
unbufcall() can be called from user or interrupt context.

SEE ALSO
bufcall(9F), esbbcall(9F)
Writing Device Drivers
STREAMS Programming Guide
NAME       unlinkb – remove a message block from the head of a message

SYNOPSIS  #include <sys/stream.h>
            mblk_t *unlinkb(mblk_t *mp);

INTERFACE LEVEL
ARGUMENTS  mp   Pointer to the message.

DESCRIPTION unlinkb() removes the first message block from the message pointed to by mp. A new message, minus the removed message block, is returned.

RETURN VALUES If successful, unlinkb() returns a pointer to the message with the first message block removed. If there is only one message block in the message, NULL is returned.

CONTEXT     unlinkb() can be called from user or interrupt context.

EXAMPLE     The routine expects to get passed an M_PROTO T_DATA_IND message. It will remove and free the M_PROTO header and return the remaining M_DATA portion of the message.
            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 }

SEE ALSO   linkb(9F)

Writing Device Drivers
STREAMS Programming Guide
NAME  
utimeout – cancel previous timeout function call

SYNOPSIS  
#include <sys/types.h>
#include <sys/conf.h>

int untimeout(int id);

INTERFACE  
LEVEL  
Architecture independent level 1 (DDI/DKI).

ARGUMENTS  
id  
Identification value generated by a previous timeout(9F) function call.

DESCRIPTION  
utimeout() cancels a pending timeout(9F) request. untimeout() will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to untimeout() or a deadlock may result.

Since no mutex should be held across the call to untimeout(), there is a race condition between the occurrence of an expected event and the execution of the timeout handler. In particular, it should be noted that no problems will result from calling untimeout() for a timeout which is either running on another CPU, or has already completed. Drivers should be structured with the understanding that the arrival of both an interrupt and a timeout for that interrupt can occasionally occur, in either order.

RETURN VALUES  
utimeout() returns -1 if the id is not found. Otherwise, it returns an integer value greater than or equal to 0.

CONTEXT  
utimeout() can be called from user or interrupt context.

EXAMPLES  
In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

static void
xxtimeout_handler(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->lock);
    cv_signal(&xsp->cv);
    xsp->flags |= TIMED_OUT;
    mutex_exit(&xsp->lock);
    xsp->timeout_id = 0;
}

static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
}
mutex_enter(&xsp->lock);
/* Service interrupt */
cv_signal(&xsp->cv);
mutex_exit(&xsp->lock);
if (xsp->timeout_id != 0) {
    (void) untimeout(xsp->timeout_id);
    xsp->timeout_id = 0;
}
return(DDI_INTR_CLAIMED);
}
static void
xxcheckcond(struct xxstate *xsp)
{
    mutex_enter(&xsp->lock);
    while (/* Waiting for interrupt or timeout */)  
        cv_wait(&xsp->cv, &xsp->lock);
    if (xsp->flags & TIMED_OUT)
        cmn_err(CE_WARN, "Device not responding");
    mutex_exit(&xsp->lock);
}

SEE ALSO  open(9E), cv_signal(9F), cv_wait_sig(9F), delay(9F), timeout(9F)

Writing Device Drivers
NAME
ureadc – add character to a uio structure

SYNOPSIS
#include <sys/uio.h>
#include <sys/types.h>

int ureadc(int c, uio_t *uio_p);

INTERFACE LEVEL
Architecture independent level 1 (DDI/DKI).

ARGUMENTS

| c | The character added to the uio (9S) structure. |
|   | Pointer to the uio(9S) structure. |

DESCRIPTION
ureadc() transfers the character c into the address space of the uio(9S) structure pointed to by uio_p, and updates the uio structure as for uiomove(9F).

RETURN VALUES
0 is returned on success and EFAULT on failure.

CONTEXT
ureadc() can be called from user or interrupt context.

SEE ALSO
uiomove(9F), uwritec(9F), iovec(9S), uio(9S)

Writing Device Drivers

modified 11 Apr 1991

SunOS 5.6

9F-553
<table>
<thead>
<tr>
<th><strong>NAME</strong></th>
<th>uwritec – remove a character from a uioc structure</th>
</tr>
</thead>
</table>
| **SYNOPSIS** | #include <sys/uioc.h>  
int uwritec (uioc_t *uioc_p); |
| **INTERFACE LEVEL** | Architecture independent level 1 (DDI/DKI). |
| **ARGUMENTS** | uioc_p Pointer to the uioc(9S) structure. |
| **DESCRIPTION** | uwritec() returns a character from the uioc structure pointed to by uioc_p, and updates the uioc structure as for uiomove(9F). |
| **RETURN VALUES** | The next character for processing is returned on success, and -1 is returned if uioc is empty or there is an error. |
| **CONTEXT** | uwritec() can be called from user or interrupt context. |
| **SEE ALSO** | uiomove(9F), ureadc(9F), iovec(9S), uioc(9S) |

*Writing Device Drivers*
NAME  va_arg, va_start, va_copy, va_end – handle variable argument list
SYNOPSIS  
#include <sys/varargs.h>

void va_start(va_list pvar, void parmN);
(type *) va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);

INTERFACE LEVEL  Solaris DDI specific (Solaris DDI).
ARGUMENTS  
va_start()  
pvar  Pointer to variable argument list.
name  Identifier of rightmost parameter in the function definition.

va_arg()  
pvar  Pointer to variable argument list.
type  Type name of the next argument to be returned.

va_copy()  
dest  Destination variable argument list.
src  Source variable argument list.

va_end()  
pvar  Pointer to variable argument list.

DESCRIPTION  This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists but do not use the varargs macros are inherently non-portable, as different machines use different argument-passing conventions. Routines that accept a variable argument list can use these macros to traverse the list.

va_list is the type defined for the variable used to traverse the list of arguments.

va_start() is called to initialize pvar to the beginning of the variable argument list.
va_start() must be invoked before any access to the unnamed arguments. The parameter name is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the “,...”). If this parameter is declared with the register storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

va_arg() expands to an expression that has the type and value of the next argument in the call. The parameter pvar must be initialized by va_start(). Each invocation of va_arg() modifies pvar so that the values of successive arguments are returned in turn. The parameter type is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply 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()**, and 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 should any of these restrictions not be 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**

The following example uses these routines to create a variable length command. This may be useful for a device which provides for a variable length command set.

**ncmdbbytes** is the number of bytes in the command. The new command is written to **cmdp**.

```c
static void
xx_write_cmd(u_char *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, u_char);
    }
    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.

It is non-portable to specify a second argument of **char** or **short** to **va_arg**, because arguments seen by the called function are not **char** or **short**. C converts **char** and **short** arguments to **int** before passing them to a function.
NAME
vsprintf – format characters in memory

SYNOPSIS
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

char *vsprintf(char *buf, const char *fmt, va_list ap);

INTERFACE
LEVEL
Solaris DDI specific (Solaris DDI).

ARGUMENTS
buf Pointer to a character string.
fmt Pointer to a character string.
ap Pointer to a variable argument list.

DESCRIPTION
vsprintf() builds a string in buf under the control of the format fmt. The format is a character string with either plain characters, which are simply copied into buf, or conversion specifications, each of which converts zero or more arguments, again copied into buf. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user’s responsibility to ensure that enough storage is available for buf.
ap contains the list of arguments used by the conversion specifications in fmt. ap is a variable argument list and must be initialized by calling va_start(9F). va_end(9F) is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by va_start(9F) and va_end(9F), are possible.
Each conversion specification is introduced by the % character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (ll) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:
d, D, o, O, x, X, u
The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X) or unsigned decimal (u), respectively, and copied. The letters abcdef are used for x and X conversion.
c
The character value of the argument is copied.
b
This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form <base>[<arg>…]. The base supplies the conversion base for the first argument as a binary

modified 6 May 1996

SunOS 5.6

9F-557
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

\texttt{reg=3<BitTwo,BitOne>\n in buf.}

\texttt{vsprintf(buf, "reg=\%b\n", 3, "\10\2BitTwo\1BitOne")}

The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is \texttt{NULL}, the string <null string> is used in its place.

\% Copy a \%; no argument is converted.

RETURN VALUES \texttt{vsprintf()} returns its first argument, \texttt{buf}.

CONTEXT \texttt{vsprintf()} can be called from user, kernel, or interrupt context.

EXAMPLES In this example, \texttt{xxerror()} accepts a pointer to a dev_info_t structure \texttt{dip}, an error level \texttt{level}, a format \texttt{fmt}, and a variable number of arguments. The routine uses \texttt{vsprintf()} to format the error message in \texttt{buf}. Note that \texttt{va_start(9F)} and \texttt{va_end(9F)} bracket the call to \texttt{vsprintf()}. instance, level, name, and \texttt{buf} are then passed to \texttt{cmn_err(9F)}.

\texttt{#include <sys/varargs.h>}
\texttt{#include <sys/ddi.h>}
\texttt{#include <sys/sunddi.h>}

\texttt{#define MAX_MSG 256}

\texttt{void}
\texttt{xxerror(dev_info_t \*dip, int level, const char \*fmt, \ldots )}
\{
    \texttt{va_list ap;}
    \texttt{int instance;}
    \texttt{char buf[MAX_MSG],}
\texttt{*name;}

    \texttt{instance = ddi_get_instance(dip);}
    \texttt{name = ddi_binding_name(dip);}

    \texttt{/* format buf using fmt and arguments contained in ap */}
    \texttt{va_start(ap, fmt);}
    \texttt{vsprintf(buf, fmt, ap);}
    \texttt{va_end(ap);}

Kernel Functions for Drivers

SEE ALSO

cmn_err(9F), ddi_binding_name(9F), ddi_get_instance(9F), va_arg(9F)

Writing Device Drivers

modified 6 May 1996 SunOS 5.6 9F-559
WR (9F) Kernel Functions for Drivers

NAME
WR, wr – get pointer to the write queue for this module or driver

SYNOPSIS
#include <sys/stream.h>
#include <sys/ddi.h>
queue_t *WR(queue_t *q);

INTERFACE
Architecture independent level 1 (DDI/DKI).

ARGUMENTS
q Pointer to the read queue whose write queue is to be returned.

DESCRIPTION
The WR() function accepts a read queue pointer as an argument and returns a pointer to the write queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a read queue. WR() will not check for queue type, and a system panic could result if the pointer is not to a read queue.

RETURN VALUES
The pointer to the write queue.

CONTEXT
WR() can be called from user or interrupt context.

EXAMPLES
In a STREAMS close(9E) routine, the driver or module is passed a pointer to the read queue. These usually are set to the address of the module-specific data structure for the minor device.

```c
1  xxxclose(q, flag)
2     queue_t *q;
3     int flag;
4 {
5     q->q_ptr = NULL;
6     WR(q)->q_ptr = NULL;
7     ...
```

SEE ALSO
close(9E), OTHERQ(9F), RD(9F)

Writing Device Drivers
STREAMS Programming Guide
Index

A
activate a new DMA window —
  ddi_dma_getwin, 9F-254
add a fully initialized kstat to the system —
  kstat_install, 9F-413
add a soft interrupt
  — ddi_add_softintr, 9F-204
add an interrupt handler
  — ddi_add_intr, 9F-201
address
  return mapped virtual address —
    csx_GetMappedAddr, 9F-106
adjmsg — trim bytes from a message, 9F-37
allocate and free a scsi_pkt structure —
  scsi_hba_pkt_alloc, 9F-507
  scsi_hba_pkt_free, 9F-507
allocate and free non-sequentially accessed memory
  — ddi_iopb_alloc, 9F-286
  — ddi_iopb_free, 9F-286
allocate and free transport structures —
  scsi_hba_tran Alloc, 9F-510
  scsi_hba_tran_free, 9F-510
allocate DMA handle —
  ddi_dma_alloc_handle, 9F-237
allocate kernel memory
  — ddi_umem_alloc, 9F-356
  — ddi_umem_free, 9F-356
allocate kernel memory, continued
  — ddi_umem_zalloc, 9F-356
  — kmem_alloc, 9F-408
  — kmem_free, 9F-408
  — kmem_zalloc, 9F-408
allocate memory for DMA transfer —
  ddi_dma_mem_alloc, 9F-257
allocate space — rmalloc, 9F-476
allocate space from a resource map —
  rmalloc_wait, 9F-481
allow 64 bit transfers on SBus —
  ddi_dma_set_sbus64, 9F-270
anocancel — prevent cancellation of asynchronous I/O request, 9F-40
aphysio — perform asynchronous physical I/O, 9F-41
assert — expression verification, 9F-43
asynchronous physical I/O — aphysio, 9F-41
asynchronous STREAMS perimeter upgrade —
  qwriter, 9F-474

B
bcopy — copy data between address locations in kernel, 9F-47
binds a system buffer to a DMA handle —
  ddi_dma_buf_bind_handle, 9F-240
binds an address to a DMA handle —
ddi_dma_addr_bind_handle, 9F-233
bioclone — clone another buffer, 9F-49
bioerror — indicate error in buffer header, 9F-54
biofini — uninitialized a buffer structure, 9F-55
bioinit — initialize a buffer structure, 9F-56
biomodified — check if a buffer is modified, 9F-57
bioreset — reuse a private buffer header after I/O is complete, 9F-58
biosize — returns size of a buffer structure, 9F-59
bufcall — call a function when a buffer becomes available, 9F-549, 9F-65
buffer header
indicate error — bioerror, 9F-54
reuse a private buffer header after I/O is complete — bioreset, 9F-58
busy-wait for specified interval — drv_usecwait, 9F-380
byte streams
compare two — bcmp, 9F-46
bytes, size
convert size in pages — ptob, 9F-450
convert to size in memory pages (round down) — btop, 9F-63
convert to size in memory pages (round up) — btopr, 9F-64
C
call a function when a buffer becomes available — qbufcall, 9F-463
call a STREAMS put procedure — put, 9F-453
cancel a pending qbufcall request — qunbufcall, 9F-470
cancel previous timeout function call — qun-timeout, 9F-471
cancellation of asynchronous I/O — anocancel, 9F-40
character strings
compare two null terminated strings —
strcmp, strncmp, 9F-537
convert between an integer and a decimal string — stoi, numtos, 9F-534
check for an available buffer — testb, 9F-543
check for the existence of a property —
ddi_prop_exists, 9F-323
check if a buffer is modified — biomodified, 9F-57
CIS tuple
first tuple — csx_GetFirstTuple, 9F-103
next tuple — csx_GetNextTuple, 9F-103
clear client event mask —
csx_ReleaseSocketMask, 9F-188
client
register client — csx_RegisterClient, 9F-169
client event mask
return client event mask —
csx_GetEventMask, 9F-196
set client event mask — csx_SetEventMask, 9F-196
client return
— csx_GetFirstClient, 9F-101
— csx_GetNextClient, 9F-101
copy a string from one location to another —
strncpy, 9F-537
determine the number of non-null bytes in a string —
strlen, 9F-538
find a character in a string —
strchr, 9F-535
format in memory —
sprintf, 9F-532
control device components’ availability for power management
— pm_busy_component, 9F-444
— pm_idle_component, 9F-444
configure PC Card and socket —
csx_RequestConfiguration, 9F-178
control device components' availability for power management
— pm_busy_component, 9F-444
— pm_idle_component, 9F-444
Index—2
control driver notification of user accesses —
  ddi_mapdev_intercept, 9F-296
  ddi_mapdev_nointercept, 9F-296
control the validation of memory address translations
  — devmap_load, 9F-372
  — devmap_unload, 9F-372
convert a DMA segment to a DMA address cookie
  — ddi_dma_segtocookie, 9F-268
convert clock ticks to microseconds —
  drv_hztousec, 9F-377
convert device sizes — csx_ConvertSize, 9F-90
convert device speeds — csx_ConvertSpeed, 9F-91
convert error return codes to text strings —
  csx_Error2Text, 9F-97
convert events to text strings — csx_Event2Text, 9F-98
convert microseconds to clock ticks —
  drv_usectohz, 9F-379
copy data from one device register to another device register — ddi_device_copy, 9F-220
create a minor node for this device —
  ddi_create_minor_node, 9F-218
create and initialize a new kstat — kstat_create, 9F-411
create driver-controlled mapping of device —
  ddi_mapdev, 9F-294
create minor nodes for client —
  csx_MakeDeviceNode, 9F-112
csx_AccessConfigurationRegister — read or write a PC Card Configuration Register, 9F-88
csx_ConvertSize — convert device sizes, 9F-90
csx_ConvertSpeed — convert device speeds, 9F-91
csx_CS_DDI_Info — obtain DDI information, 9F-92
csx_DeregisterClient — remove client from Card Services list, 9F-94
csx_DupHandle — duplicate access handle, 9F-95
csx_Error2Text — convert error return codes to text strings, 9F-97
csx_Event2Text — convert events to text strings, 9F-98
csx_FreeHandle — free access handle, 9F-99
csx_Get16 — read from device register, 9F-100
csx_Get32 — read from device register, 9F-100
csx_Get64 — read from device register, 9F-100
csx_Get8 — read from device register, 9F-100
csx_GetEventMask — return client event mask, 9F-196
csx_GetFirstClient — return first client, 9F-101
csx_GetFirstTuple — return first CIS tuple, 9F-103
csx_GetHandleOffset — return current access handle offset, 9F-105
csx_GetMappedAddr — return mapped virtual address, 9F-106
csx_GetNextClient — return next client, 9F-101
csx_GetNextTuple — return next CIS tuple, 9F-103
csx_GetStatus — return status of PC Card and socket, 9F-107
csx_GetTupleData — return data portion of tuple, 9F-110
csx_MakeDeviceNode — create minor nodes for client, 9F-112
csx_MapLogSocket — return physical socket number, 9F-114
csx_MapMemPage — map memory area on PC Card, 9F-115
csx_ModifyConfiguration — modify PC Card configuration, 9F-116
csx_ModifyWindow — modify window attributes, 9F-119
csx_Parse_CISTPL_BATTERY — parse Battery Replacement Date tuple, 9F-121
csx_Parse_CISTPL_BYTEORDER — parse Byte Order tuple, 9F-122
csx_Parse_CISTPL_CFTABLE_ENTRY — parse Card Configuration Table tuple, 9F-124
csx_Parse_CISTPL_CONFIG — parse Card Configuration tuple, 9F-130
csx_Parse_CISTPL_DATE — parse Card Initiali-
Interleaving tuple, 9F-162

Index−4
resources, 9F-190

*csx_ResetFunction* — reset a function on a PC card, 9F-195

*csx_SetEventMask* — set client event mask, 9F-196

*csx_SetHandleOffset* — set current access handle offset, 9F-198

*csx_ValidateCIS* — validate Card Information Structure (CIS), 9F-199

**D**

datamsg — test whether a message is a data message, 9F-200

DDI access credential structure
— *ddi_get_cred*, 9F-279

DDI announce a device
— *ddi_report_dev*, 9F-344

DDI device access
slave access only — *ddi_slaveonly*, 9F-350

DDI device critical region of control
enter — *ddi_enter_critical*, 9F-276
exit — *ddi_exit_critical*, 9F-276

DDI device information structure
find parent — *ddi_get_parent*, 9F-282
get the root of the dev_info tree —
  *ddi_root_node*, 9F-347
remove a minor node for this devinfo —
  *ddi_remove_minor_node*, 9F-341

DDI device instance number
get — *ddi_get_instance*, 9F-281

DDI device mapping
*ddi_mapdev* — create driver-controlled mapping of device, 9F-294
*ddi_mapdev_intercept* — control driver notification of user accesses, 9F-296
*ddi_mapdev_nointercept* — control driver notification of user accesses, 9F-296
*devmap_default_access* — device mapping access entry point, 9F-360

DDI device registers
map — *ddi_map_regs*, 9F-300
return the number of register sets —
  *ddi_dev_nregs*, 9F-230

DDI device registers, *continued*
return the size — *ddi_dev_regsize*, 9F-231
unmap — *ddi_unmap_regs*, 9F-300

DDI device virtual address
read 16 bit — *ddi_peek16*, 9F-315
read 32 bit — *ddi_peek32*, 9F-315
read 64 bit — *ddi_peek64*, 9F-315
read 8 bit — *ddi_peek8*, 9F-315
read a value — *ddi_peek*, 9F-315
write 16 bit — *ddi_poke16*, 9F-317
write 32 bit — *ddi_poke32*, 9F-317
write 64 bit — *ddi_poke64*, 9F-317
write 8 bit — *ddi_poke8*, 9F-317
write a value — *ddi_poke*, 9F-317

DDI device’s private data area
get the address —
  *ddi_get_driver_private*, 9F-280
set the address —
  *ddi_set_driver_private*, 9F-280

DDI devinfo node name
return — *ddi_binding_name*, 9F-210, 9F-314

DDI direct memory access
convert DMA handle to DMA addressing
  cookie — *ddi_dma_htoc*, 9F-256

DDI direct memory access services
allocate consistent memory—
  *ddi_iopb_alloc*, 9F-302
convert a DMA cookie — *ddi_dma_coff*, 9F-245
easier DMA setup — *ddi_dma_addr_setup*, 9F-236, 9F-243
find minimum alignment and transfer size for
device — *ddi_omin*, 9F-285
find post DMA mapping alignment and
  minimum effect properties —
  *ddi_dma_devalign*, 9F-247
free consistent memory — *ddi_iopb_free*, 9F-302
report current DMA window offset and size —
  *ddi_dma_curwin*, 9F-246
setup DMA mapping — *ddi_dma_setup*, 9F-264, 9F-265, 9F-268
setup DMA resources — *ddi_dma_setup*, 9F-271
shift current DMA window —
DDI direct memory access services, continued
9F-260
tear down DMA mapping — ddi_dma_free,
9F-252
DDI information — csx_CS_DDI_Info, 9F-92
DDI interrupt handling
add an interrupt — ddi_add_intr, 9F-201
get interrupt block cookie —
    ddi_get_iblock_cookie, 9F-201
indicate interrupt handler type —
    ddi_intr_hilevel, 9F-283
remove an interrupt — ddi_remove_intr, 9F-201
return the number of interrupt specifications —
    ddi_dev_nintrs, 9F-229
DDI memory mapping
map a segment — ddi_segmap, 9F-349,
    9F-371
DDI page size conversions
    — ddi_btop, 9F-211
    — ddi_btopr, 9F-211
    — ddi_ptob, 9F-211
DDI property management
create properties for leaf device drivers —
    ddi_prop_create, 9F-320
    — ddi_getlongprop, 9F-332
    — ddi_getlongprop_buf, 9F-332
    — ddi_getprop, 9F-332
    — ddi_getproplen, 9F-332
    — ddi_prop_op, 9F-332
modify properties for leaf device drivers —
    ddi_prop_modify, 9F-320
remove all properties for leaf device drivers —
    ddi_prop_remove_all, 9F-320
remove properties for leaf device drivers —
    ddi_prop_remove, 9F-320
DDI self identifying devices
tell whether a device is self-identifying —
    ddi_dev_is_sid, 9F-228
DDI soft interrupt handling
add a soft interrupt — ddi_add_softintr, 9F-204
get soft interrupt block cookie —
    ddi_get_soft_iblock_cookie,
9F-204
DDI soft interrupt handling, continued
remove a soft interrupt —
    ddi_remove_softintr, 9F-204
DDI soft state utility routines
allocate state structure —
    ddi_soft_state_zalloc, 9F-351
free soft state entry —
    ddi_soft_state_free, 9F-351
get pointer to soft state —
    ddi_get_soft_state, 9F-351
initialize state — ddi_soft_state_init,
    9F-351
remove all state info —
    ddi_soft_state_fini, 9F-351
ddi_add_intr — add an interrupt handler, 9F-201
ddi_add_softintr — add a soft interrupt, 9F-204
ddi_binding_name — return driver binding
    name, 9F-210
ddi_create_minor_node — create a minor node
    for this device, 9F-218
ddi_dev_is_needed — inform the system that a
    device’s component is required, 9F-226
ddi_device_copy — copy data from one device
    register to another device register, 9F-220
    ddi_device_zero — zero fill the device register,
    9F-222
ddi_device_compare — Kernel interfaces for dev-
    ice ids , 9F-223
ddi_device_free — Kernel interfaces for device
    ids , 9F-223
ddi_device_init — Kernel interfaces for device
    ids , 9F-223
ddi_device_register — Kernel interfaces for
    device ids , 9F-223
ddi_devidx_sizeof — Kernel interfaces for device
    ids , 9F-223
ddi_device_unregister — Kernel interfaces for
    device ids , 9F-223
ddi_devidx_valid — Kernel interfaces for device
    ids , 9F-223
ddi_dma_addr_bind_handle — binds an
    address to a DMA handle, 9F-233
ddi_dma_alloc_handle — allocate DMA handle, 9F-237
ddi_dma_buf_bind_handle — binds a system buffer to a DMA handle, 9F-240
ddi_dma_burstsizes — find out the allowed burst sizes for a DMA mapping, 9F-244
ddi_dma_free_handle — free DMA handle, 9F-253
ddi_dma_getwin — activate a new DMA window, 9F-254
ddi_dma_mem_alloc — allocate memory for DMA transfer, 9F-257
ddi_dma_mem_free — free previously allocated memory, 9F-259
ddi_dma_nextcookie — retrieve subsequent DMA cookie, 9F-262
ddi_dma_nextseg — get next DMA segment, 9F-264
ddi_dma_nextwin — get next DMA window, 9F-265
ddi_dma_numwin — retrieve number of DMA windows, 9F-267
ddi_dma_setsegcookie — convert a DMA segment to a DMA address cookie, 9F-268
ddi_dma_set_sbus64 — allow 64 bit transfers on SBus, 9F-270
ddi_dma_sync — synchronize CPU and I/O views of memory, 9F-273
ddi_dma_unbind_handle — unbinds the address in a DMA handle, 9F-275
ddi_dmae — system DMA engine functions, 9F-248
ddi_dmae_1stparty — system DMA engine functions, 9F-248
ddi_dmae_alloc — system DMA engine functions, 9F-248
ddi_dmae_disable — system DMA engine functions, 9F-248
ddi_dmae_enable — system DMA engine functions, 9F-248
ddi_dmae_getattr — system DMA engine functions, 9F-248
ddi_dmae_getcnt — system DMA engine functions, 9F-248
ddi_dmae_getlim — system DMA engine functions, 9F-248
ddi_dmae_prog — system DMA engine functions, 9F-248
ddi_dmae_release — system DMA engine functions, 9F-248
ddi_dmae_stop — system DMA engine functions, 9F-248
ddi_ffs — find first (last) bit set in a long integer, 9F-277
ddi_fls — find first (last) bit set in a long integer, 9F-277
ddi_get_iblock_cookie — get interrupt block cookie, 9F-201
ddi_get_name — return driver binding name, 9F-210
ddi_get_soft_iblock_cookie — get soft interrupt block cookie, 9F-204
ddi_get16 — read data from the device, 9F-278
ddi_get32 — read data from the device, 9F-278
ddi_get64 — read data from the device, 9F-278
ddi_get8 — read data from the device, 9F-278
ddi_io_get16 — read data from the mapped device register in I/O space, 9F-284
ddi_io_get32 — read data from the mapped device register in I/O space, 9F-284
ddi_io_get8 — read data from the mapped device register in I/O space, 9F-284
ddi_io_getb — read data from the mapped device register in I/O space, 9F-284
ddi_io_getl — read data from the mapped device register in I/O space, 9F-284
ddi_io_getw — read data from the mapped device register in I/O space, 9F-284
ddi_io_put16 — write data to the mapped device register in I/O space, 9F-288
ddi_io_put32 — write data to the mapped device register in I/O space, 9F-288
ddi_io_put8 — write data to the mapped device register in I/O space, 9F-288
ddi_io_putb — write data to the mapped device register in I/O space, 9F-288
register in I/O space, 9F-288
ddi_io_putl — write data to the mapped device
register in I/O space, 9F-288
ddi_io_putw — write data to the mapped device
register in I/O space, 9F-288
ddi_io_rep_get16 — read multiple data from
the mapped device register in I/O space,
9F-290
ddi_io_rep_get32 — read multiple data from
the mapped device register in I/O space,
9F-290
ddi_io_rep_get8 — read multiple data from the
mapped device register in I/O space, 9F-290
ddi_io_rep_getb — read multiple data from the
mapped device register in I/O space, 9F-290
ddi_io_rep_getl — read multiple data from the
mapped device register in I/O space, 9F-290
ddi_io_rep_put16 — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_io_rep_put32 — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_io_rep_put8 — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_io_rep_putb — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_io_rep_putl — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_io_rep_putw — write multiple data to the
mapped device register in I/O space, 9F-292
ddi_iopb_alloc — allocate and free non-
sequentially accessed memory, 9F-286
ddi_iopb_free — allocate and free non-
sequentially accessed memory, 9F-286
ddi_mapdev — create driver-controlled mapping
of device, 9F-294
ddi_mapdev_intercept — control driver
notification of user accesses, 9F-296
ddi_mapdev_intercept — control driver
notification of user accesses, 9F-296
ddi_mapdev_set_device_acc_attr — Set the
device attributes for the mapping, 9F-298
ddi_mem_get16 — read data from mapped device
in the memory space or allocated DMA
memory, 9F-304
ddi_mem_get32 — read data from mapped device
in the memory space or allocated DMA
memory, 9F-304
ddi_mem_get64 — read data from mapped device
in the memory space or allocated DMA
memory, 9F-304
ddi_mem_put16 — write data to mapped device in
the memory space or allocated DMA memory,
9F-305
ddi_mem_put32 — write data to mapped device in
the memory space or allocated DMA memory,
9F-305
ddi_mem_put64 — write data to mapped device in
the memory space or allocated DMA memory,
9F-305
ddi_mem_rep_get16 — read data from mapped
device in the memory space or allocated DMA
memory, 9F-306
ddi_mem_rep_get32 — read data from mapped
device in the memory space or allocated DMA
memory, 9F-306
ddi_mem_rep_get64 — read data from mapped
device in the memory space or allocated DMA
memory, 9F-306
ddi_mem_rep_get8 — read data from mapped
device in the memory space or allocated DMA
memory, 9F-306
ddi_mem_rep_put16 — write data to mapped
device in the memory space or allocated DMA
memory, 9F-308
ddi_mem_rep_put32 — write data to mapped
device in the memory space or allocated DMA
memory, 9F-308
ddi_mem_rep_put64 — write data to mapped
device in the memory space or allocated DMA
memory, 9F-308
ddi_mem_rep_put8 — write data to mapped device in
the memory space or allocated DMA
memory, 9F-308

ddi_mmap_get_model — return data model type of current thread, 9F-310

ddi_model_convert_from — Determine if there is a need to translate shared data structure contents, 9F-312

ddi_node_name — return the devinfo node name, 9F-314

ddi_prop_exists — check for the existence of a property, 9F-323

ddi_prop_get_int — look up integer property, 9F-325

ddi_prop_lookup — lookup property information, 9F-328

ddi_prop_lookup_byte_array — lookup property information, 9F-328

ddi_prop_lookup_int_array — lookup property information, 9F-328

ddi_prop_lookup_string — lookup property information, 9F-328

ddi_prop_lookup_string_array — lookup property information, 9F-328

ddi_prop_update — update property information, 9F-334

ddi_prop_update_byte_array — update property information, 9F-334

ddi_prop_update_int — update property information, 9F-334

ddi_prop_update_int_array — update property information, 9F-334

ddi_prop_update_string — update property information, 9F-334

ddi_prop_update_string_array — update property information, 9F-334

ddi_put16 — write data to the device, 9F-337

ddi_put32 — write data to the device, 9F-337

ddi_put64 — write data to the device, 9F-337

ddi_put8 — write data to the device, 9F-337

ddi_regs_map_free — free mapped register address space, 9F-338

ddi_regs_map_setup — set up a mapping for a register address space, 9F-339

ddi_remove_intr — remove an interrupt handler, 9F-201

ddi_remove_softintr — remove a soft interrupt, 9F-204

ddi_rep_get16 — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_get32 — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_get64 — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_get8 — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_getl — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_getll — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_getw — read data from the mapped memory address, device register or allocated DMA memory address, 9F-342

ddi_rep_put16 — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345

ddi_rep_put32 — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345

ddi_rep_put64 — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345

ddi_rep_put8 — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345

ddi_rep_putl — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345

Index
DMA memory address, 9F-345
ddi_rep_putl — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345
ddi_rep_putll — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345
ddi_rep_putw — write data to the mapped memory address, device register or allocated DMA memory address, 9F-345
ddi_trigger_softintr — trigger a soft interrupt, 9F-204
ddi_umem_alloc — allocate kernel memory, 9F-356
ddi_umem_free — allocate kernel memory, 9F-356
ddi_umem_zalloc — allocate kernel memory, 9F-356
default SCSI HBA probe function — scsi_hba_probe, 9F-509
delay — delay process execution for a specified number of clock ticks, 9F-358
deregister client from Card Services list — csx_DeregisterClient, 9F-94
determine data model type mismatch — ddi_model_convert_from, 9F-312
Device Driver Interface, See DDI
device mapping access entry point — devmap_default_access, 9F-360
device switch tables
  return function for insignificant entries — nulldev, 9F-436
devices
  get major device number — getmajor, 9F-397
  get minor device number — getminor, 9F-398
  make device number from major and minor numbers — makedevice, 9F-420
devices, non-pollable
  error return function — nochpoll, 9F-433
devmap_default_access — device mapping access entry point, 9F-360
devmap_devmem_setup — Set driver memory mapping parameters, continued
  devmap_devmem_setup(), 9F-363
devmap_umem_setup(), 9F-364
devmap_do_ctxmgt — perform device context switching on a mapping, 9F-366
devmap_load — control the validation of memory address translations, 9F-372
devmap_set_ctx_timeout — set context management timeout value, 9F-369
devmap_umem_setup — Set driver memory mapping parameters, 9F-364
devmap unload — control the validation of memory address translations, 9F-372
disksort — single direction elevator seek sort for buffers, 9F-374
DMA mapping, the allowed burst sizes for —
ddi_dma_burstsizes, 9F-244
driver buffers
  copy data — ddi_copyin, 9F-212
  copy data from driver — ddi_copyout, 9F-215
  copy data from driver to user program — copyout, 9F-86
  copy data from user program — copyin, 9F-82
driver error messages
  display an error message or panic the system — cmn_err, 9F-72
driver privilege — drv_priv, 9F-378
drv_getparm — retrieve kernel state information, 9F-375
drv_hztousec — convert clock ticks to microseconds, 9F-377
drv_priv — determine driver privilege, 9F-378
drv_usectohz — convert microseconds to clock ticks, 9F-379
drv_usecwait — busy-wait for specified interval, 9F-380
dupb — duplicate a message block descriptor, 9F-381
duplicate a message — dupmsg, 9F-384
duplicate a message block descriptor — dupb,
duplicate access handle — csx_DupHandle, 9F-95
dupmsg — duplicate a message, 9F-384

edible accesses to the PCI Local Bus
Configuration space.
— pci_config_setup, 9F-441
— pci_config_teardown, 9F-441
error return codes converted to text strings —
csx_Error2Text, 9F-97
error return function for illegal entries — nodev,
9F-434
event mask
return client event mask —
csx_GetEventMask, 9F-196
set client event mask — csx_SetEventMask,
9F-196
events converted to text strings —
csx_Event2Text, 9F-98
expression verification
— assert, 9F-43

find first (last) bit set in a long integer — ddi_ffs,
9F-277
ddi_fils, 9F-277
first CIS tuple — csx_GetFirstTuple, 9F-103
flushband — flush messages for specified priority
band, 9F-389
free access handle — csx_FreeHandle, 9F-99
free DMA handle
— ddi_dma_free_handle, 9F-253
free mapped register address space —
ddi_regs_map_free, 9F-338
free previously allocated memory —
ddi_dma_mem_free, 9F-259
free space — rmfree, 9F-482
freerbuf — free a raw buffer header, 9F-394
freeze, thaw the state of a stream — freezestr,
9F-395
unfreezestr, 9F-395
freeze str — freeze, thaw the state of a stream,
9F-395
I/O, physical, continued
   — minphys, 9F-442
   — physio, 9F-442
inb — read from an I/O port, 9F-404
inform the system that a device’s component is
required. — ddi_dev_is_needed, 9F-226
initialize a buffer structure — bioinit, 9F-56
initialize a named kstat — kstat_named_init,
9F-414
inl — read from an I/O port, 9F-404
interrupt handling
   add an interrupt — ddi_add_intr, 9F-201
   get interrupt block cookie —
      ddi_get_iblock_cookie, 9F-201
   remove an interrupt — ddi_remove_intr,
      9F-201
inw — read from an I/O port, 9F-404
IOC_CONVERT_FROM — Determine if there is a
need to translate M_IOCTL contents, 9F-407
IRQ resource
   release IRQ resource — csx_ReleaseIRQ,
      9F-186
   request IRQ resource — csx_RequestIRQ,
      9F-186
K
kernel address locations
   between locations — bcopy, 9F-47
kernel addresses
   get page frame number — hat_getkpfnum,
      9F-403
Kernel interfaces for device ids
   — ddi_devid_compare, 9F-223
   — ddi_devid_free, 9F-223
   — ddi_devid_init, 9F-223
   — ddi_devid_register, 9F-223
   — ddi_devid_sizeof, 9F-223
   — ddi_devid_unregister, 9F-223
   — ddi_devid_valid, 9F-223
kernel modules, dynamic loading
   add loadable module — mod_install, 9F-426
   query loadable module — mod_info, 9F-426
   remove loadable module — mod_remove,
      9F-426
kmem_alloc — allocate kernel memory, 9F-408
kmem_free — allocate kernel memory, 9F-408
kmem_zalloc — allocate kernel memory, 9F-408
kstat_create — create and initialize a new kstat,
      9F-411
kstat_delete — remove a kstat from the system,
      9F-412
kstat_install — add a fully initialized kstat to
      the system, 9F-413
kstat_named_init — initialize a named kstat,
      9F-414
kstat_queue — update I/O kstat statistics, 9F-415
kstat_runq_back_to_waitq — update I/O
      kstat statistics, 9F-415
kstat_runq_enter — update I/O kstat statistics,
      9F-415
kstat_runq_exit — update I/O kstat statistics,
      9F-415
kstat_waitq_enter — update I/O kstat statistics,
      9F-415
kstat_waitq_exit — update I/O kstat statistics,
      9F-415
kstat_waitq_to_runq — update I/O kstat statistics,
      9F-415
L
look up integer property — ddi_prop_get_int,
      9F-325
lookup property information
   — ddi_prop_lookup, 9F-328
   — ddi_prop_lookup_byte_array, 9F-328
   — ddi_prop_lookup_int_array, 9F-328
   — ddi_prop_lookup_string, 9F-328
   — ddi_prop_lookup_string_array,
      9F-328
M
makedevice — make device number from major
      and minor numbers, 9F-420
map memory area on PC Card —
   csx_MapMemPage, 9F-115
max — return the larger of two integers, 9F-421
memory
clear for a given number of bytes — bzero, 9F-68
min — return the lesser of two integers, 9F-422
minor node for device
create — ddi_create_minor_node, 9F-218
modify PC Card configuration —
csx_ModifyConfiguration, 9F-116
modify window attributes — csx_ModifyWindow, 9F-119
mt-streams — STREAMS multithreading, 9F-429
mutex routines
— mutex, 9F-431
— mutex_destroy, 9F-431
— mutex_enter, 9F-431
— mutex_exit, 9F-431
— mutex_init, 9F-431
— mutex_owned, 9F-431
— mutex_tryenter, 9F-431
mutual exclusion lock, See mutex

N
next CIS tuple — csx_GetNextTuple, 9F-103
nodes
create minor nodes for client —
csx_MakeDeviceNode, 9F-112
notify target driver of bus resets —
csci_reset_notify, 9F-525

O
obtain DDI information — csx_CS_DDI_Info, 9F-92
OTHERQ — get pointer to queue’s partner queue, 9F-437
outb — write to an I/O port, 9F-438
outl — write to an I/O port, 9F-438
outw — write to an I/O port, 9F-438

P
parse Battery Replacement Date tuple —
csx_Parse_CISTPL_BATTERY, 9F-121
parse Byte Order tuple —
csx_Parse_CISTPL_BYTEORDER, 9F-122
parse Card Configuration Table tuple —
csx_Parse_CISTPL_CFTABLE_ENTRY, 9F-124
parse Card Initialization Date tuple —
csx_Parse_CISTPL_DATE, 9F-132
parse Configuration tuple —
csx_Parse_CISTPL_CONFIG, 9F-130
parse Data Organization tuple —
csx_Parse_CISTPL_ORG, 9F-159
parse Data Recording Format tuple —
csx_Parse_CISTPL_FORMAT, 9F-140
parse Device Geo A tuple —
csx_Parse_CISTPL_DEVICEGEO_A, 9F-138
parse Device Geo tuple —
csx_Parse_CISTPL_DEVICEGEO, 9F-136
parse Device Information tuple
for Attribute Memory —
csx_Parse_CISTPL_DEVICE_A, 9F-133
for Common Memory —
csx_Parse_CISTPL_DEVICE, 9F-133
parse Function Extension tuple —
csx_Parse_CISTPL_FUNCE, 9F-142
parse Function Identification tuple —
csx_Parse_CISTPL_FUNCID, 9F-148
parse Geometry tuple —
csx_Parse_CISTPL_GEOMETRY, 9F-150
parse JEDEC Identifier tuple
for Attribute Memory —
csx_Parse_CISTPL_JEDEC_A, 9F-151
for Common Memory —
csx_Parse_CISTPL_JEDEC_C, 9F-151
parse Level-1 Version/Product Information tuple —
csx_Parse_CISTPL_VERS_1, 9F-163
parse Level-2 Version and Information tuple —
csx_Parse_CISTPL_VERS_2, 9F-164
parse Link Target tuple —
csx_Parse_CISTPL_LINKTARGET, 9F-153
parse Long Link A tuple
— csx_Parse_CISTPL_LONGLINK_A, 9F-154
parse Long Link C tuple
— csx_Parse_CISTPL_LONGLINK_C, 9F-154
Index–13
parse Manufacturer Identification tuple —
csx_Parse_CISTPL_MANFID, 9F-158
parse Multi-Function tuple —
csx_Parse_CISTPL_LONGLINK_MFC, 9F-156
parse Other Condition Device Information tuple
for Attribute Memory —
csx_Parse_CISTPL_DEVICE_OA, 9F-133
for Common Memory —
csx_Parse_CISTPLDEVICE_OC, 9F-133
parse Software Interleaving tuple —
csx_Parse_CISTPL_SWIL, 9F-162
parse Special Purpose tuple —
csx_Parse_CISTPL_SPCL, 9F-160
parser, for tuples (generic) — csx_ParseTuple, 9F-166
pci_config_get16 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_get32 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_get64 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_getb — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_getl — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_getw — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_put16 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_put32 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_put64 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_putb — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_putl — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_put11 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_put11 — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_putw — read or write single datum
of various sizes to the PCI Local Bus
Configuration space, 9F-439
pci_config_setup — enable/disable accesses to
the PCI Local Bus Configuration space., 9F-441
pci_config_teardown — enable/disable
accesses to the PCI Local Bus Configuration space., 9F-441
perform device context switching on a mapping —
devmap_do_ctxmgt, 9F-366
pm_busy_component — control device components’ availability for power management, 9F-444
pm_idle_component — control device components’ availability for power management, 9F-444
pollwakeup — inform a process that an event has
occurred, 9F-447
proc_ref — send a signal to a process, 9F-448
proc_signal — send a signal to a process, 9F-448
proc_unref — send a signal to a process, 9F-448
put — call a STREAMS put procedure, 9F-453

Index–14
Q
qbufcall — call a function when a buffer becomes available, 9F-463
qtimeout — execute a function after a specified length of time, 9F-469
qunbufcall — cancel a pending qbufcall request, 9F-470
quntimeout — cancel previous timeout function call, 9F-471
qwait — STREAMS wait routines, 9F-472
qwait_sig — STREAMS wait routines, 9F-472
qwriter — asynchronous STREAMS perimeter upgrade, 9F-474

R
raw buffer
  free a raw buffer header — freerbuf, 9F-394
  get a raw buffer header — getrbuf, 9F-402
RD — get pointer to the read queue, 9F-475
read data from mapped device in the memory space or allocated DMA memory
  — ddi_mem_get16, 9F-304
  — ddi_mem_get32, 9F-304
  — ddi_mem_get64, 9F-304
  — ddi_mem_get8, 9F-304
  — ddi_mem_rep_get16, 9F-306
  — ddi_mem_rep_get32, 9F-306
  — ddi_mem_rep_get64, 9F-306
  — ddi_mem_rep_get8, 9F-306
read data from the device
  — ddi_get16, 9F-278
  — ddi_get32, 9F-278
  — ddi_get64, 9F-278
  — ddi_get8, 9F-278
read data from the mapped device register in I/O space
  — ddi_io_get16, 9F-284
  — ddi_io_get32, 9F-284
  — ddi_io_get8, 9F-284
read data from the mapped memory address, device register or allocated DMA memory address
  — ddi_rep_get16, 9F-342
  — ddi_rep_get32, 9F-342
  — ddi_rep_get64, 9F-342
  — ddi_rep_get8, 9F-342
  — ddi_rep_getb, 9F-342
  — ddi_rep_getl, 9F-342
  — ddi_rep_getll, 9F-342
  — ddi_rep_getw, 9F-342
read from an I/O port — inb, 9F-404
inl, 9F-404
inw, 9F-404
repinsb, 9F-404
repinsd, 9F-404
repinsw, 9F-404
read from device register
  — csx_Get16, 9F-100
  — csx_Get32, 9F-100
  — csx_Get64, 9F-100
  — csx_Get8, 9F-100
read multiple data from the mapped device register in I/O space
  — ddi_io_rep_get16, 9F-290
  — ddi_io_rep_get32, 9F-290
  — ddi_io_rep_get8, 9F-290
  — ddi_io_rep_getb, 9F-290
  — ddi_io_rep_getl, 9F-290
  — ddi_io_rep_getll, 9F-290
read or write a PC Card Configuration Register —
csx_AccessConfigurationRegister, 9F-88
read or write single datum of various sizes to the PCI Local Bus Configuration space
  — pci_config_get16, 9F-439
  — pci_config_get32, 9F-439
  — pci_config_get64, 9F-439
  — pci_config_get8, 9F-439
  — pci_config_getb, 9F-439
  — pci_config_getl, 9F-439
  — pci_config_getll, 9F-439
  — pci_config_getw, 9F-439
  — pci_config_get16, 9F-439
  — pci_config_get32, 9F-439
  — pci_config_get64, 9F-439
read or write single datum of various sizes to the
PCI Local Bus Configuration space,
continued
— pci_config_put8, 9F-439
— pci_config_putchar, 9F-439
— pci_config_put1, 9F-439
— pci_config_put1t, 9F-439
— pci_config_put_l, 9F-439
— pci_config_putll, 9F-439
read repetitively from device register
— csx_RepGet16, 9F-174
— csx_RepGet32, 9F-174
— csx_RepGet64, 9F-174
— csx_RepGet8, 9F-174
readers/writer lock functions
— rw_destroy, 9F-487
— rw_downgrade, 9F-487
— rw_enter, 9F-487
— rw_exit, 9F-487
— rw_init, 9F-487
— rw_read_locked, 9F-487
— rw_tryenter, 9F-487
— rw_tryupgrade, 9F-487
— rwlock, 9F-487
register client — csx_RegisterClient, 9F-169
release client event mask —
— csx_ReleaseSocketMask, 9F-188
release configuration on PC Card —
— csx_ReleaseConfiguration, 9F-172
release I/O resources — csx_ReleaseIO, 9F-182
release IRQ resource — csx_ReleaseIRQ, 9F-186
release window resources —
— csx_ReleaseWindow, 9F-190
remove a kstat from the system — kstat_delete, 9F-412
remove a soft interrupt
— ddi_remove_softintra, 9F-204
remove an interrupt handler
— ddi_remove_intr, 9F-201
remove client from Card Services list —
— csx_DeregisterClient, 9F-94
repinsb — read from an I/O port, 9F-404
repinsd — read from an I/O port, 9F-404
repinsw — read from an I/O port, 9F-404
repoutsb — write to an I/O port, 9F-438
repoutsd — write to an I/O port, 9F-438
repouts w — write to an I/O port, 9F-438
request client event mask —
— csx_RequestSocketMask, 9F-188
request I/O resources — csx_RequestIO, 9F-182
request IRQ resource — csx_RequestIRQ, 9F-186
request window resources —
— csx_RequestWindow, 9F-190
reset a function on a PC card —
— csx_ResetFunction, 9F-195
resource map
allocate resource maps — rmallocmap, 9F-479
free resource maps — rmallocmap, 9F-479
retrieve kernel state information — drv_getparm, 9F-375
retrieve number of DMA windows —
— ddi_dma_numwin, 9F-267
retrieve subsequent DMA cookie —
— ddi_dma_nextcookie, 9F-262
return client —
— csx_GetFirstClient, 9F-101
— csx_GetNextClient, 9F-101
return client event mask — csx_GetEventMask, 9F-196
return current access handle offset —
— csx_GetHandleOffset, 9F-105
return data model type of current thread —
— ddi_mmap_get_model, 9F-310
return data portion of tuple —
— csx_GetTupleData, 9F-110
return driver binding name —
— ddi_binding_name, 9F-210
— ddi_get_name, 9F-210
return index matching capability string —
— scsi_hba_lookup_capstr, 9F-506
return physical socket number —
— csx_MapLogSocket, 9F-114
return status of PC Card and socket —
— csx_GetStatus, 9F-107
return the devinfo node name — ddi_node_name, 9F-314
return the larger of two integers — max, 9F-421

Index—16
return the lesser of two integers — min, 9F-422
return tuple
  first CIS tuple — csx_GetFirstTuple, 9F-103
  next CIS tuple — csx_GetNextTuple, 9F-103
returns size of a buffer structure — biosize, 9F-59
rmalloc — allocate space from a resource map, 9F-476
rmalloc_wait — allocate space from a resource map, 9F-481
rmfree — free space back into a resource map, 9F-482

S
SAMESTR — test if next queue is in the same stream, 9F-489
SCSI commands, make packet
  — makecom, 9F-418
  — makecom_g0, 9F-418
  — makecom_g0_s, 9F-418
  — makecom_g1, 9F-418
  — makecom_g5, 9F-418
SCSI dma utility routines
  — scsi_dmafree, 9F-496
  — scsi_dmaget, 9F-496
SCSI HBA attach and detach routines
  — scsi_hba_attach, 9F-502
  — scsi_hba_attach_setup, 9F-502
  — scsi_hba_detach, 9F-502
SCSI Host Bus Adapter system initialization and completion routines
  — scsi_hba_init, 9F-505
  — scsi_hba_init, 9F-505
SCSI packet
  allocate a SCSI packet in iopb map —
    get_pktiopb, 9F-399
  free a packet in iopb map — free_pktiopb, 9F-399
  free an allocated SCSI packet and its DMA
    resource — scsi_destroy_pkt, 9F-495
SCSI packet utility routines
  — scsi_pktalloc, 9F-519
  — scsi_pktfree, 9F-519
SCSI packet utility routines, continued
  — scsi_resalloc, 9F-519
  — scsi_resfree, 9F-519
scsi_abort — abort a SCSI command, 9F-490
scsi_alloc_consistent_buf — scsi dma utility
  for allocating an I/O buffer for SCSI DMA, 9F-491
scsi_cname — decode SCSI commands, 9F-493
scsi_destroy_pkt — free an allocated SCSI packet and its DMA
  resource, 9F-495
scsi_dname — decode SCSI peripheral device
  type, 9F-493
scsi_errmsg — display a SCSI request sense message, 9F-498
scsi_free_consistent_buf — free a previously allocated SCSI DMA I/O buffer, 9F-501
scsi_hba_attach — SCSI HBA attach and detach
  routines, 9F-502
scsi_hba_attach_setup — SCSI HBA attach
  and detach routines, 9F-502
scsi_hba_detach — SCSI HBA attach and detach
  routines, 9F-502
scsi_hba_fini — SCSI Host Bus Adapter system
  completion routines, 9F-505
scsi_hba_init — SCSI Host Bus Adapter system
  initialization routines, 9F-505
scsi_hba_lookup_capstr — return index
  matching capability string, 9F-506
scsi_hba_pkt_alloc — allocate and free a
  scsi_pkt structure, 9F-507
scsi_hba_pkt_free — allocate and free a
  scsi_pkt structure, 9F-507
scsi_hbaProbe — default SCSI HBA probe
  function, 9F-509
scsi_hba_tran_alloc — allocate and free trans-
  port structures, 9F-510
scsi_hba_tran_free — allocate and free trans-
  port structures, 9F-510
scsi_ifgetcap — get SCSI transport capability,
  9F-511
scsi_ifsetcap — set SCSI transport capability,
  9F-511

Index–17
scsi_init_pkt — prepare a complete SCSI packet, 9F-514

scsi_log — display a SCSI-device-related message, 9F-517

scsi_mname — decode SCSI messages, 9F-493

scsi_poll — run a polled SCSI command on behalf of a target driver, 9F-521

scsi_probe — utility for probing a scsi device, 9F-522

scsi_reset — reset a SCSI bus or target, 9F-524

scsi_reset_notify — notify target driver of bus resets, 9F-525

scsi_rname — decode SCSI packet completion reasons, 9F-493

scsi_slave — utility for SCSI target drivers to establish the presence of a target, 9F-526

scsi_sname — decode SCSI sense keys, 9F-493

scsi_sync_pkt — synchronize CPU and I/O views of memory, 9F-527

scsi_transport — request by a target driver to start a SCSI command, 9F-528

scsi_unprobe — free resources allocated during initial probing, 9F-529

scsi_unslave — free resources allocated during initial probing, 9F-529

Semaphore functions
  — sema_destroy, 9F-530
  — sema_init, 9F-530
  — sema_p, 9F-530
  — sema_p_sig, 9F-530
  — sema_tryp, 9F-530
  — sema_v, 9F-530
  — semaphore, 9F-530

Send a signal to a process
  — proc_ref, 9F-448
  — proc_signal, 9F-448
  — proc_unref, 9F-448

Set client event mask —
  csx_RequestSocketMask, 9F-188, 9F-196

Set current access handle offset —
  csx_SetHandleOffset, 9F-198

Set driver memory mapping parameters, continued
  — devmap_devmem_setup, 9F-364
  — devmap_umem_setup, 9F-364

Set the device attributes for the mapping —
  ddi_mapdev_set_device_acc_attr, 9F-298

Set up a mapping for a register address space —
  ddi_regs_map_setup, 9F-339

Single direction elevator seek sort for buffers —
  disksort, 9F-374

Size in bytes
  convert size in pages — ptob, 9F-450
  convert to size in memory pages (round down) — btop, 9F-63
  convert to size in memory pages (round up) — btopr, 9F-64

Socket number
  return physical socket number —
  csx_MapLogSocket, 9F-114

Soft interrupt handling
  add a soft interrupt — ddi_add_softintr, 9F-204
  get soft interrupt block cookie —
  ddi_get_soft_iblock_cookie, 9F-204
  remove a soft interrupt —
  ddi_remove_softintr, 9F-204
  trigger a soft interrupt —
  ddi_trigger_softintr, 9F-204

Spyprintf — format characters in memory, 9F-532

Status of PC Card and socket — csx_GetStatus, 9F-107

STREAMS ioctl blocks
  allocate — mkiocb, 9F-423

STREAMS message blocks
  allocate — allocb, 9F-38
  attach a user-supplied data buffer in place —
  esballoc, 9F-386
  call a function when a buffer becomes available —
  bufcall, 9F-549, 9F-65, 9F-463, 9F-470
  call function when buffer is available — esbbcall, 9F-388
  concatenate bytes in a message — msgpullup, 9F-428, 9F-451

Index—18
STREAMS message blocks, continued
concatenate two — linkb, 9F-417
copy — copyb, 9F-80
erase the contents of a buffer — clrbuf, 9F-71
free all message blocks in a message — freemsg, 9F-393
free one — freeb, 9F-392
remove from head of message — unlinkb, 9F-550
remove one from a message — rmvb, 9F-483
STREAMS message queue
insert a message into a queue — insq, 9F-405
STREAMS message queues, 9F-44
STREAMS Message queues
get next message — getq, 9F-401
STREAMS message queues
reschedule a queue for service — enableok, 9F-385
test for room — canput, 9F-69, 9F-70
STREAMS messages
copy a message — copymsg, 9F-84
flush for specified priority band — flushband, 9F-389
remove form queue — flushq, 9F-390, 9F-485
return the number of bytes in a message — msgdsize, 9F-427
submit messages to the log driver — strlog, 9F-475
test whether a message is a data message — datamsg, 9F-200
trim bytes — adjmsg, 9F-37
STREAMS multithreading
— mt-streams, 9F-429
qbufcall — call a function when a buffer becomes available, 9F-463
qtimeout — execute a function after a specified length of time, 9F-469
qunbufcall — cancel a pending qbufcall request, 9F-470
quntimeout — cancel previous timeout function call, 9F-471
qwait, qwait_sig — STREAMS wait routines, 9F-472
qwriter — asynchronous STREAMS perimeter upgrade, 9F-474

STREAMS put and service procedures
disable — qprocsoff, 9F-465
enable — qprocson, 9F-465
STREAMS queues
change information about a queue or band of the queue — strqset, 9F-541
enable a queue — qenable, 9F-464
get information about a queue or band of the queue — strqget, 9F-540
get pointer to queue’s partner queue — OTHERQ, 9F-437
get pointer to the read queue — RD, 9F-475
number of messages on a queue — qsize, 9F-468
place a message at the head of a queue — putbq, 9F-454
prevent a queue from being scheduled — noenable, 9F-435
put a message on a queue — putq, 9F-462
send a control message to a queue — putctl, 9F-456, 9F-460
send a control message with a one-byte parameter to a queue — putctl1, 9F-455, 9F-459
send a message on a stream in the reverse direction — qreply, 9F-466
send a message to the next queue — putnext, 9F-458
test for flow control in specified priority band — bcanput, 9F-45
test if next queue is in the same stream — SAMESTR, 9F-489
STREAMS wait routines — qwait, qwait_sig, 9F-472
STREAMS write queues
get pointer for this module or driver — WR, 9F-560
swab — swap bytes in 16-bit halfwords, 9F-542
synchronize CPU and I/O views of memory — ddi_dma_sync, 9F-273, 9F-527
system DMA engine functions
— ddi_dmae, 9F-248
— ddi_dmae_1stparty, 9F-248
— ddi_dmae_alloc, 9F-248
— ddi_dmae_disable, 9F-248

Index—19
system DMA engine functions, continued
— ddi_dmae_enable, 9F-248
— ddi_dmae_getattr, 9F-248
— ddi_dmae_getcnt, 9F-248
— ddi_dmae_getlim, 9F-248
— ddi_dmae_prog, 9F-248
— ddi_dmae_release, 9F-248
— ddi_dmae_stop, 9F-248

T
testb — check for an available buffer, 9F-543
timeout — execute a function after a specified
temporary length of time, 9F-545
cancel previous timeout function call —
untimout, 9F-551
trigger a soft interrupt
— ddi_trigger_softintri, 9F-204
tuple
first CIS tuple — csx_GetFirstTuple, 9F-103
next CIS tuple — csx_GetNextTuple, 9F-103
return data portion of tuple —
    csx_GetTupleData, 9F-110
tuple entry
generic tuple parser — csx_ParseTuple, 9F-166
parse Battery Replacement Date tuple —
    csx_Parse_CISTPL_BATTERY, 9F-121
parse Byte Order tuple —
    csx_Parse_CISTPL_BYTEORDER, 9F-122
parse Card Configuration Table tuple —
    csx_Parse_CISTPL_CFTABLE_ENTRY, 9F-124
parse Card Initialization Date tuple —
    csx_Parse_CISTPL_DATE, 9F-132
parse Configuration tuple —
    csx_Parse_CISTPL_CONFIG, 9F-130
parse Data Organization tuple —
    csx_Parse_CISTPL_ORG, 9F-159
parse Data Recording Format tuple —
    csx_Parse_CISTPL_FORMAT, 9F-140
parse Device Geo A tuple —
    csx_Parse_CISTPL_DEVICE_A, 9F-138
parse Device Geo tuple —
    csx_Parse_CISTPLDEVICEGEO, 9F-136
parse Device Information tuple for Attribute
    Memory —
    csx_Parse_CISTPLDEVICE_A, 9F-133
parse Device Information tuple for Common
    Memory —
    csx_Parse_CISTPLDEVICE, 9F-133
parse Function Extension tuple —
    csx_Parse_CISTPL_FUNC, 9F-142
parse Function Identification tuple —
    csx_Parse_CISTPL_FUNCID, 9F-148
parse Geometry tuple —
    csx_Parse_CISTPL_GEOMETRY, 9F-150
parse JEDEC Identifier tuple for Attribute
    Memory —
    csx_Parse_CISTPLJEDEC_A, 9F-151
parse JEDEC Identifier tuple for Common
    Memory —
    csx_Parse_CISTPLJEDEC_C, 9F-151
parse Level-1 Version/Product Information
tuple —
    csx_Parse_CISTPLVERS_1, 9F-163
parse Level-2 Version and Information tuple —
    csx_Parse_CISTPLVERS_2, 9F-164
parse Link Target tuple —
    csx_Parse_CISTPL_LINKTARGET, 9F-153
parse Long Link A tuple —
    csx_Parse_CISTPL_LONGLINK_A, 9F-154
parse Long Link C tuple —
    csx_Parse_CISTPL_LONGLINK_C, 9F-154
parse Manufacturer Identification tuple —
    csx_Parse_CISTPL_MANFID, 9F-158
parse Multi-Function tuple —
    csx_Parse_CISTPL_LONGLINK_MFC, 9F-158

Index—20
tuple entry, continued
9F-156
parse Other Condition Device Information
tuple for Attribute Memory —
csx_Parse_CISTPL.DEVICE_OA, 9F-133
parse Other Condition Device Information
tuple for Common Memory —
csx_Parse_CISTPL.DEVICE_OC, 9F-133
parse Software Interleaving tuple —
csx_Parse_CISTPL.SWIL, 9F-162
parse Special Purpose tuple —
csx_Parse_CISTPL.SPCL, 9F-160

U
uio structure
  add character — ureadc, 9F-553
  remove a character — uwritec, 9F-554
uiomove — copy kernel data using uio structure, 9F-548
unbinds the address in a DMA handle —
ddi_dma_unbind_handle, 9F-275
unfreezestr — freeze, thaw the state of a stream, 9F-395
uninitialize a buffer structure — biofini, 9F-55
update I/O kstat statistics
  — kstat_queue, 9F-415
  — kstat_runq_back_to_waitq, 9F-415
  — kstat_runq_enter, 9F-415
  — kstat_runq_exit, 9F-415
  — kstat_waitq_enter, 9F-415
  — kstat_waitq_exit, 9F-415
  — kstat_waitq_to_runq, 9F-415
update property information.
  — ddi_prop_update, 9F-334
  — ddi_prop_update_byte_array, 9F-334
  — ddi_prop_update_int, 9F-334
  — ddi_prop_update_int_array, 9F-334
  — ddi_prop_update_string, 9F-334
  — ddi_prop_update_string_array, 9F-334

V
va_arg — handle variable argument list, 9F-555
va_copy — handle variable argument list, 9F-555
va_end — handle variable argument list, 9F-555
va_start — handle variable argument list, 9F-555
validate Card Information Structure (CIS) —
csx_ValidateCIS, 9F-199
virtual address
  return mapped virtual address —
csx_GetMappedAddr, 9F-106
vsprintf — format characters in memory, 9F-557

W
window resources
  release window resources —
csx_ReleaseWindow, 9F-190
  request window resources —
csx_RequestWindow, 9F-190
write data to mapped device in the memory space
  or allocated DMA memory
    — ddi_mem_put16, 9F-305
    — ddi_mem_put32, 9F-305
    — ddi_mem_put64, 9F-305
    — ddi_mem_put8, 9F-305
    — ddi_mem_rep_put16, 9F-308
    — ddi_mem_rep_put32, 9F-308
    — ddi_mem_rep_put64, 9F-308
    — ddi_mem_rep_put8, 9F-308
write data to the device
    — ddi_put16, 9F-337
    — ddi_put32, 9F-337
    — ddi_put64, 9F-337
    — ddi_put8, 9F-337
write data to the mapped device register in I/O
  space
    — ddi_io_put16, 9F-288
    — ddi_io_put32, 9F-288
    — ddi_io_put8, 9F-288
    — ddi_io_putb, 9F-288
    — ddi_io_putl, 9F-288
    — ddi_io_putw, 9F-288
write data to the mapped memory address, device
  register or allocated DMA memory address
    — ddi_rep_put16, 9F-345
write data to the mapped memory address, device
register or allocated DMA memory
address, continued
— ddi_rep_put32, 9F-345
— ddi_rep_put64, 9F-345
— ddi_rep_put8, 9F-345
— ddi_rep_putb, 9F-345
— ddi_rep_putl, 9F-345
— ddi_rep_putll, 9F-345
— ddi_rep_putw, 9F-345
write multiple data to the mapped device register in
I/O space
— ddi_io_rep_put16, 9F-292
— ddi_io_rep_put32, 9F-292
— ddi_io_rep_put8, 9F-292
— ddi_io_rep_putb, 9F-292
— ddi_io_rep_putl, 9F-292
— ddi_io_rep_putw, 9F-292
write or read a PC Card Configuration Register —
csx_AccessConfigurationRegister, 9F-88
write repetitively to device register
— csx_RepPut16, 9F-176
— csx_RepPut32, 9F-176
— csx_RepPut64, 9F-176
— csx_RepPut8, 9F-176
write to an I/O port
— outb, 9F-438
— outl, 9F-438
— outw, 9F-438
— repoutsb, 9F-438
— repoutsd, 9F-438
— repoutsw, 9F-438
write to device register
— csx_Put16, 9F-168
— csx_Put32, 9F-168
— csx_Put64, 9F-168
— csx_Put8, 9F-168
Z
zero fill the device register — ddi_device_zero,
9F-222