man pages section 9: DDI and DKI Driver
Entry Points
# Contents

**Preface** ..................................................................................................................................................... 5

**Introduction** ........................................................................................................................................... 9

*Intro(9E)* ............................................................................................................................................... 10

**Driver Entry Points** ................................................................................................................................. 17

*aread(9E)* ............................................................................................................................................ 18
*attach(9E)* ........................................................................................................................................... 20
*awrite(9E)* ........................................................................................................................................... 22
*chpoll(9E)* ........................................................................................................................................... 24
*close(9E)* ............................................................................................................................................. 26
*csx_event_handler(9E)* .................................................................................................................... 29
*detach(9E)* ........................................................................................................................................... 36
*devmap(9E)* ........................................................................................................................................... 38
*devmap_access(9E)* .......................................................................................................................... 42
*devmap_contextmgt(9E)* .................................................................................................................... 45
*devmap_dup(9E)* ................................................................................................................................ 48
*devmap_map(9E)* ............................................................................................................................... 50
*devmap_unmap(9E)* ............................................................................................................................ 52
*dump(9E)* ............................................................................................................................................ 55
*_fini(9E)* ............................................................................................................................................. 56
*getinfo(9E)* ....................................................................................................................................... 59
*gld(9E)* ............................................................................................................................................... 61
*identify(9E)* ....................................................................................................................................... 66
*iocctl(9E)* ........................................................................................................................................... 67
*ks_snapshot(9E)* ............................................................................................................................... 71
*ks_update(9E)* ................................................................................................................................... 73
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac(9E)</td>
<td>75</td>
</tr>
<tr>
<td>mmap(9E)</td>
<td>82</td>
</tr>
<tr>
<td>open(9E)</td>
<td>86</td>
</tr>
<tr>
<td>power(9E)</td>
<td>89</td>
</tr>
<tr>
<td>print(9E)</td>
<td>91</td>
</tr>
<tr>
<td>probe(9E)</td>
<td>92</td>
</tr>
<tr>
<td>prop_op(9E)</td>
<td>93</td>
</tr>
<tr>
<td>put(9E)</td>
<td>95</td>
</tr>
<tr>
<td>read(9E)</td>
<td>97</td>
</tr>
<tr>
<td>segmap(9E)</td>
<td>99</td>
</tr>
<tr>
<td>srv(9E)</td>
<td>101</td>
</tr>
<tr>
<td>strategy(9E)</td>
<td>103</td>
</tr>
<tr>
<td>tran_abort(9E)</td>
<td>104</td>
</tr>
<tr>
<td>tran_bus_reset(9E)</td>
<td>105</td>
</tr>
<tr>
<td>tran_dmafree(9E)</td>
<td>106</td>
</tr>
<tr>
<td>tran_getcap(9E)</td>
<td>107</td>
</tr>
<tr>
<td>tran_init_pkt(9E)</td>
<td>109</td>
</tr>
<tr>
<td>tran_quiesce(9E)</td>
<td>112</td>
</tr>
<tr>
<td>tran_reset(9E)</td>
<td>113</td>
</tr>
<tr>
<td>tran_reset_notify(9E)</td>
<td>115</td>
</tr>
<tr>
<td>tran_start(9E)</td>
<td>116</td>
</tr>
<tr>
<td>tran_sync_pkt(9E)</td>
<td>119</td>
</tr>
<tr>
<td>tran_tgt_free(9E)</td>
<td>120</td>
</tr>
<tr>
<td>tran_tgt_init(9E)</td>
<td>121</td>
</tr>
<tr>
<td>tran_tgt_probe(9E)</td>
<td>122</td>
</tr>
<tr>
<td>write(9E)</td>
<td>123</td>
</tr>
</tbody>
</table>
Preface

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

Overview

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

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.
Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and man(1) for more information about man pages in general.

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

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

The following special characters are used in this section:

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

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

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

- **{}** Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

**PROTOCOL**
This section occurs only in subsection 3R to indicate the protocol description file.

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

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

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

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

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

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

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

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

Commands
Modifiers
Variables
Expressions
Input Grammar
EXAMPLES

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

ENVIRONMENT VARIABLES

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

EXIT STATUS

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

FILES

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

ATTRIBUTES

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

SEE ALSO

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

DIAGNOSTICS

This section lists diagnostic messages with a brief explanation of the condition causing the error.

WARNINGS

This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.

NOTES

This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and, wherever possible, suggests workarounds.
Introduction
This page provides an overview of device driver interfaces and all of the Section 9 man pages (9E, 9F, 9P, and 9S). This overview is followed by an introduction to Section 9E, the driver entry-point routines.

Section 9 provides reference information needed to write device drivers for the Solaris operating environment. It describes the interfaces provided by the Device Driver Interface and the Driver-Kernel Interface (DDI/DKI).

Porting

Software is usually considered portable if it can be adapted to run in a different environment more cheaply than it can be rewritten. The new environment may include a different processor, operating system, and even the language in which the program is written, if a language translator is available. Likewise the new environment might include multiple processors. More often, however, software is ported between environments that share an operating system, processor, and source language. The source code is modified to accommodate the differences in compilers or processors or releases of the operating system.

In the past, device drivers did not port easily for one or more of the following reasons:

- To enhance functionality, members had been added to kernel data structures accessed by drivers, or the sizes of existing members had been redefined.
- The calling or return syntax of kernel functions had changed.
- Driver developers did not use existing kernel functions where available, or relied on undocumented side effects that were not maintained in the next release.
- Architecture-specific code had been scattered throughout the driver when it could have been isolated.

Operating systems are periodically reissued to customers as a way to improve performance, fix bugs, and add new features. This is probably the most common threat to compatibility encountered by developers responsible for maintaining software. Another common problem is upgrading hardware. As new hardware is developed, customers occasionally decide to upgrade to faster, more capable computers of the same family. Although they may run the same operating system as those being replaced, architecture-specific code may prevent the software from porting.

Scope of Interfaces

Although application programs have all of the porting problems mentioned, developers attempting to port device drivers have special challenges. Before describing the DDI/DKI, it is necessary to understand the position of device drivers in operating systems.

Device drivers are kernel modules that control data transferred to and received from peripheral devices but are developed independently from the rest of the kernel. If the goal of achieving complete freedom in modifying the kernel is to be reconciled with the goal of binary
compatibility with existing drivers, the interaction between drivers and the kernel must be rigorously regulated. This driver/kernel service interface is the most important of the three distinguishable interfaces for a driver, summarized as follows:

- **Driver–Kernel.** I/O System calls result in calls to driver entry point routines. These make up the kernel-to-driver part of the service interface, described in Section 9E. Drivers may call any of the functions described in Section 9F. These are the driver-to-kernel part of the interface.

- **Driver–Hardware.** All drivers (except software drivers) must include code for interrupt handling, and may also perform direct memory access (DMA). These and other hardware-specific interactions make up the driver/hardware interface.

- **Driver–Boot/Configuration Software.** The interaction between the driver and the boot and configuration software is the third interface affecting drivers.

**Scope of the DDI/DKI**

The primary goal of the DDI/DKI is to facilitate both source and binary portability across successive releases of the operating systems on a particular machine. In addition, it promotes source portability across implementations of UNIX on different machines, and applies only to implementations based on System V Release 4. The DDI/DKI consists of several sections:

- **DDI/DKI Architecture Independent** - These interfaces are supported on all implementations of System V Release 4.

- **DKI-only** - These interfaces are part of System V Release 4, and may not be supported in future releases of System V. There are only two interfaces in this class, `segmap(9E)` and `hat_getkpfnum(9F)`.

- **Solaris DDI** - These interfaces specific to Solaris.

- **Solaris SPARC specific DDI** - These interfaces are specific to the SPARC processor, and may not be available on other processors supported by Solaris.

- **Solaris x86 specific DDI** - These interfaces are specific to the x86 processor, and may not be available on other processors supported by Solaris.

To achieve the goal of source and binary compatibility, the functions, routines, and structures specified in the DDI/DKI must be used according to these rules:

- Drivers cannot access system state structures (for example, `u` and `sysinfo`) directly.

- For structures external to the driver that may be accessed directly, only the utility functions provided in Section 9F should be used. More generally, these functions should be used wherever possible.

- The headers `<sys/ddi.h>` and `<sys/sunddi.h>` must be the last header files included by the driver.

**Audience**
Section 9 is for software engineers responsible for creating, modifying, or maintaining drivers that run on this operating system and beyond. It assumes that the reader is familiar with system internals and the C programming language.

**PCMCIA Standard**

The *PC Card 95 Standard* is listed under the SEE ALSO heading in some Section 9 reference pages. This refers to documentation published by the Personal Computer Memory Card International Association (PCMCIA) and the Japan Electronic Industry Development Association (JEIDA).

**How to Use Section 9**

Section 9 is divided into the following subsections:

- **9E** Driver Entry Points – contains reference pages for all driver entry point routines.
- **9F** Kernel Functions – contains reference pages for all driver support routines.
- **9P** Driver Properties – contains reference pages for driver properties.
- **9S** Data Structures – contains reference pages for driver-related structures.

**Compatibility Note**

Sun Microsystems’s implementation of the DDI/DKI was designed to provide binary compatibility for third-party device drivers across currently supported hardware platforms across minor releases of the operating system. However, unforeseen technical issues may force changes to the binary interface of the DDI/DKI. We cannot therefore promise or in any way assure that DDI/DKI-compliant device drivers will continue to operate correctly on future releases.

Section 9E describes the entry-point routines a developer can include in a device driver. These are called entry-point because they provide the calling and return syntax from the kernel into the driver. Entry-points are called, for instance, in response to system calls, when the driver is loaded, or in response to STREAMS events.

Kernel functions usable by the driver are described in section 9F.

In this section, reference pages contain the following headings:

- **NAME** describes the routine’s purpose.
- **SYNOPSIS** summarizes the routine’s calling and return syntax.
- **INTERFACE LEVEL** describes any architecture dependencies. It also indicates whether the use of the entry point is required, optional, or discouraged.
- **ARGUMENTS** describes each of the routine’s arguments.
- **DESCRIPTION** provides general information about the routine.
- **RETURN VALUES** describes each of the routine’s return values.
SEE ALSO gives sources for further information.

Overview of Driver Entry-Point Routines and Naming Conventions

By convention, a prefix string is added to the driver routine names. For a driver with the prefix `prefix`, the driver code may contain routines named `prefixopen`, `prefixclose`, `prefixread`, `prefixwrite`, and so forth. All global variables associated with the driver should also use the same prefix.

All routines and data should be declared as static.

Every driver MUST include `<sys/ddi.h>` and `<sys/sunddi.h>`, in that order, and after all other include files.

The following table summarizes the STREAMS driver entry points described in this section.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>srv</td>
<td>DDI/DKI</td>
</tr>
</tbody>
</table>

The following table summarizes the driver entry points described in this section.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_fini</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>_info</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>_init</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>aread</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>attach</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>awrite</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>chpoll</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>close</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>detach</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_access</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_contextmg</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_dup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>devmap_map</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>devmap_unmap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>dump</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>getinfo</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>identify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>ioctl</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>ks_update</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mapdev_access</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mapdev_dup</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mapdev_free</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>mmap</td>
<td>DKI only</td>
</tr>
<tr>
<td>open</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>power</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>print</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>probe</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>prop_op</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>read</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>segmap</td>
<td>DKI only</td>
</tr>
<tr>
<td>strategy</td>
<td>DDI/DKI</td>
</tr>
<tr>
<td>tran_abort</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_destroy_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_dmafree</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_getcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_init_pkt</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_reset</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_reset_notify</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_setcap</td>
<td>Solaris DDI</td>
</tr>
<tr>
<td>tran_start</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>
The following table lists the error codes returned by a driver routine when it encounters an error. The error values are listed in alphabetic order and are defined in `sys/errno.h`. In the driver `open(9E), close(9E), ioctl(9E), read(9E), and write(9E)` routines, errors are passed back to the user by calling `bioerror(9F)` to set `b_flags` to the proper error code. In the driver `strategy(9E)` routine, errors are passed back to the user by setting the `b_error` member of the `buf(9S)` structure to the error code. For `STREAMS ioctl` routines, errors should be sent upstream in an `M_IOCNAK` message. For `STREAMS read()` and `write()` routines, errors should be sent upstream in an `M_ERROR` message. The driver `print` routine should not return an error code because the function that it calls, `cmn_err(9F)`, is declared as `void` (no error is returned).

<table>
<thead>
<tr>
<th>Error Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>Kernel resources, such as the buf structure or cache memory, are not available at this time (device may be busy, or the system resource is not available). This is used in open, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>An invalid address has been passed as an argument; memory addressing error. This is used in open, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>EINTR</td>
<td>Sleep interrupted by signal. This is used in open, close, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>An invalid argument was passed to the routine. This is used in open, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>EIO</td>
<td>A device error occurred; an error condition was detected in a device status register (the I/O request was valid, but an error occurred on the device). This is used in open, close, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>An attempt was made to access a device or subdevice that does not exist (one that is not configured); an attempt was made to perform an invalid I/O operation; an incorrect minor number was specified. This is used in open, close, ioctl, read, write, and strategy.</td>
</tr>
<tr>
<td>EPERM</td>
<td>A process attempting an operation did not have required permission. This is used in open, ioctl, read, write, and strategy.</td>
</tr>
</tbody>
</table>
An attempt was made to open for writing a read-only device. This is used in `open`.

The table below cross references error values to the driver routines from which the error values can be returned.

<table>
<thead>
<tr>
<th>open</th>
<th>close</th>
<th>ioctl</th>
<th>read, write and strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>EFAULT</td>
<td>EAGAIN</td>
<td>EAGAIN</td>
</tr>
<tr>
<td>EFAULT</td>
<td>EINTR</td>
<td>EFAULT</td>
<td>EFAULT</td>
</tr>
<tr>
<td>EINTR</td>
<td>EIO</td>
<td>EINTR</td>
<td>EINTR</td>
</tr>
<tr>
<td>EINVAL</td>
<td>ENXIO</td>
<td>EINVAL</td>
<td>EINVAL</td>
</tr>
<tr>
<td>EIO</td>
<td>ENXIO</td>
<td>EIO</td>
<td>EIO</td>
</tr>
<tr>
<td>ENXIO</td>
<td>EPERM</td>
<td>ENXIO</td>
<td>ENXIO</td>
</tr>
<tr>
<td>EPERM</td>
<td></td>
<td>EPERM</td>
<td></td>
</tr>
<tr>
<td>EROFS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Also Intro(9F), Intro(9S)
REFERENCE

Driver Entry Points
**Name**
aread – asynchronous read from a device

**Synopsis**
```c
#include <sys/uio.h>
#include <sys/aio_req.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
int prefix

aread(dev_t dev, struct aio_req *aio_reqp, cred_t *cred_p);
```

**Interface Level**
Solaris DDI specific (Solaris DDI). This entry point is *optional*. Drivers that do not support an aread() entry point should use nodev(9F)

**Parameters**
- **dev**
  Device number.
- **aio_reqp**
  Pointer to the aio_req(9S) structure that describes where the data is to be stored.
- **cred_p**
  Pointer to the credential structure.

**Description**
The driver's aread() routine is called to perform an asynchronous read. getminor(9F) can be used to access the minor number component of the dev argument. aread() may use the credential structure pointed to by cred_p to check for superuser access by calling drv_priv(9F). The aread() routine may also examine the uio(9S) structure through the aio_req structure pointer, aio_reqp. aread() must call aphysio(9F) with the aio_req pointer and a pointer to the driver's strategy(9E) routine.

No fields of the uio(9S) structure pointed to by aio_req, other than uio_offset or uio_loffset, may be modified for non-seekable devices.

**Return Values**
The aread() routine should return 0 for success, or the appropriate error number.

**Context**
This function is called from user context only.

**Examples**
**EXAMPLE 1**
The following is an example of an aread() routine:
```c
static int
xxaread(dev_t dev, struct aio_req *aio, cred_t *cred_p)
{
    int instance;
    struct xxstate *xsp;
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    /*Verify soft state structure has been allocated */
    if (xsp == NULL)
        return (ENXIO);
    return (aphysio(xxstrategy, anocancel,
                    dev, B_READ, xxminphys, aio));
```
EXAMPLE 1 The following is an example of an aread() routine:

(Continued)

}

See Also read(2), aioread(3AIO), awrite(9E), read(9E), strategy(9E), write(9E), anocancel(9F), aphysio(9F), ddi_get_soft_state(9F), drv_priv(9F), getminor(9F), minphys(9F), nodev(9F), aio_req(9S), cb_ops(9S), uio(9S)

Writing Device Drivers

Bugs There is no way other than calling aphysio(9F) to accomplish an asynchronous read.
attach(9E)

Name
attach – Attach a device to the system, or resume it

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

int prefixattach(dev_info_t *dip, ddi_attach_cmd_t cmd);
```

Interface Level
Solaris DDI specific (Solaris DDI)

Parameters
dip A pointer to the device’s dev_info structure.
cmd Attach type. Possible values are DDI_ATTACH and DDI_RESUME. Other values are reserved. The driver must return DDI_FAILURE if reserved values are passed to it.

Description
The attach(9E) function is the device-specific initialization entry point. This entry point is required and must be written.

DDI_ATTACH
The DDI_ATTACH command must be provided in the attach(9E) entry point. DDI_ATTACH is used to initialize a given device instance. When attach(9E) is called with cmd set to DDI_ATTACH, all normal kernel services (such as kmem_alloc(9F)) are available for use by the driver. Device interrupts are not blocked when attaching a device to the system.

The attach(9E) function is called once for each instance of the device on the system with cmd set to DDI_ATTACH. Until attach(9E) succeeds, the only driver entry point which may be called is getinfo(9E). See the Writing Device Drivers for more information. The instance number may be obtained using ddi_get_instance(9F).

At attach time, all components of a power-manageable device are assumed to be at unknown levels. Before using the device, the driver needs to bring the required component(s) to a known power level. The pm_raise_power(9F) function can be used to set the power level of a component. This function must not be called before data structures referenced in power(9E) have been initialized.

DDI_RESUME
The attach() function may be called with cmd set to DDI_RESUME after detach(9E) has been successfully called with cmd set to DDI_SUSPEND.

When called with cmd set to DDI_RESUME, attach() must restore the hardware state of a device (power may have been removed from the device), allow pending requests to continue, and service new requests. In this case, the driver must not make any assumptions about the state of the hardware, but must restore the state of the device except for the power level of components.
If the device driver uses the automatic device Power Management interfaces (driver exports the `pm-components(9P)` property), the Power Management framework sets its notion of the power level of each component of a device to `unknown` while processing a `DDI_RESUME` command.

The driver can deal with components during `DDI_RESUME` in one of the following ways:

1. If the driver can determine the power level of the component without having to power it up (for example, by calling `ddi.peek(9F)` or some other device-specific method) then it should notify the power level to the framework by calling `pm_power_has_changed(9F)`.
2. The driver must also set its own notion of the power level of the component to `unknown`. The system will consider the component idle or busy based on the most recent call to `pm_idle_component(9F)` or `pm_busy_component(9F)` for that component. If the component is idle for sufficient time, the framework will call into the driver’s `power(9E)` entry point to turn the component off. If the driver needs to access the device, then it must call `pm_raise_power(9F)` to bring the component up to the level needed for the device access to succeed. The driver must honor any request to set the power level of the component, since it cannot make any assumption about what power level the component has (or it should have called `pm_power_has_changed(9F)` as outlined above). As a special case of this, the driver may bring the component to a known state because it wants to perform an operation on the device as part of its `DDI_RESUME` processing (such as loading firmware so that it can detect hot-plug events).

**Return Values**

The `attach()` function returns:

- `DDI_SUCCESS` Successful completion
- `DDI_FAILURE` Operation failed

**Attributes**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

**See Also**

`cpr(7), pm(7D), pm-components(9P), detach(9E), getinfo(9E), identify(9E), open(9E), power(9E), probe(9E), ddi_add_intr(9F), ddi_create_minor_node(9F), ddi_get_instance(9F), ddi_map_REGS(9F), kmem_alloc(9F), pm_raise_power(9F)`

Writing Device Drivers
awrite(9E)

Name  awrite – asynchronous write to a device

Synopsis  
```c
#include <sys/uio.h>
#include <sys/aio_req.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefix awrite(dev_t dev, struct aio_req *aio_reqp, cred_t *cred_p);
```

Interface Level  Solaris DDI specific (Solaris DDI). This entry point is optional. Drivers that do not support an awrite() entry point should use nodev(9F)

Parameters  
- `dev`  Device number.
- `aio_reqp`  Pointer to the aio_req(9S) structure that describes where the data is stored.
- `cred_p`  Pointer to the credential structure.

Description  The driver's awrite() routine is called to perform an asynchronous write. getminor(9F) can be used to access the minor number component of the dev argument. awrite() may use the credential structure pointed to by cred_p to check for superuser access by calling drv_priv(9F). The awrite() routine may also examine the uio(9S) structure through the aio_req structure pointer, aio_reqp. awrite() must call aphysio(9F) with the aio_req pointer and a pointer to the driver's strategy(9E) routine.

No fields of the uio(9S) structure pointed to by aio_req, other than uio_offset or uio_loffset, may be modified for non-seekable devices.

Return Values  The awrite() routine should return 0 for success, or the appropriate error number.

Context  This function is called from user context only.

Examples  
**EXAMPLE 1** Using the awrite() routine:

The following is an example of an awrite() routine:
```c
static int
xxawrite(dev_t dev, struct aio_req *aio, cred_t *cred_p)
{
    int instance;
    struct xxstate *xsp;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    /*Verify soft state structure has been allocated */
    if (xsp == NULL)
```
EXAMPLE 1  Using the awrite() routine:  (Continued)

        return (ENXIO);
        return (aphysio(xxstrategy, anocancel, dev, B_WRITE, \
                xxminphys, aio));
    }

See Also  write(2), aiowrite(3AIO), aread(9E), read(9E), strategy(9E), write(9E), anocancel(9F),
aphysio(9F), ddi_get_soft_state(9F), drv_priv(9F), getminor(9F), minphys(9F),
nodev(9F), aio_req(9S), cb_ops(9S), uio(9S)

Writing Device Drivers

Bugs  There is no way other than calling aphysio(9F) to accomplish an asynchronous write.
chpoll – poll entry point for a non-STREAMS character driver

Synopsis

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

int prefixchpoll(dev_t dev, short events, int anyyet,
                 short *reventsp, struct pollhead **phpp);
```

Interface Level

This entry point is optional. Architecture independent level 1 (DDI/DKI).

Parameters

- `dev`: The device number for the device to be polled.
- `events`: The events that may occur. Valid events are:
  - `POLLIN`: Data other than high priority data may be read without blocking.
  - `POLLOUT`: Normal data may be written without blocking.
  - `POLLPRI`: High priority data may be received without blocking.
  - `POLLHUP`: A device hangup has occurred.
  - `POLLERR`: An error has occurred on the device.
  - `POLLRDNORM`: Normal data (priority band = 0) may be read without blocking.
  - `POLLRDBAND`: Data from a non-zero priority band may be read without blocking.
  - `POLLRWNRDNORM`: The same as POLLOUT.
  - `POLLRWNRBAND`: Priority data (priority band > 0) may be written.
- `anyyet`: A flag that is non-zero if any other file descriptors in the `pollfd` array have events pending. The `poll(2)` system call takes a pointer to an array of `pollfd` structures as one of its arguments. See the `poll(2)` reference page for more details.
- `reventsp`: A pointer to a bitmask of the returned events satisfied.
- `phpp`: A pointer to a pointer to a `pollhead` structure.

Description

The chpoll() entry point routine is used by non-STREAMS character device drivers that wish to support polling. The driver must implement the polling discipline itself. The following rules must be followed when implementing the polling discipline:

1. Implement the following algorithm when the chpoll() entry point is called:
if (events_are_satisfied_now) {
    *reventsp = satisfied_events & events;
} else {
    *reventsp = 0;
    if (!anyyet)
        *phpp = &my_local_pollhead_structure;
}
return (0);

2. Allocate an instance of the pollhead structure. This instance may be tied to the per-minor data structure defined by the driver. The pollhead structure should be treated as a “black box” by the driver. Initialize the pollhead structure by filling it with zeroes. The size of this structure is guaranteed to remain the same across releases.

3. Call the pollwakeup() function with events listed above whenever pollable events which the driver should monitor occur. This function can be called with multiple events at one time. The pollwakeup() can be called regardless of whether or not the chpoll() entry is called; it should be called every time the driver detects the pollable event. The driver must not hold any mutex across the call to pollwakeup(9F) that is acquired in its chpoll() entry point, or a deadlock may result.

Return Values  chpoll() should return 0 for success, or the appropriate error number.

See Also  poll(2), nochpoll(9F), pollwakeup(9F)

Writing Device Drivers
close(9E)

Name  
close – relinquish access to a device

Synopsis

Block and Character

```c
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixclose(dev_t dev, int flag, int otyp, cred_t *cred_p);
```

STREAMS

```c
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixclose(queue_t *q, int flag, cred_t *cred_p);
```

Interface Level  
Architecture independent level 1 (DDI/DKI). This entry point is required for block devices.

Parameters

Block and Character

- **dev**  
  Device number.

- **flag**  
  File status flag, as set by the open(2) or modified by the fcntl(2) system calls. The flag is for information only—the file should always be closed completely. Possible values are: FEXCL, FNDELAY, FREAD, FKLYR, and FWRITE. Refer to open(9E) for more information.

- **otyp**  
  Parameter supplied so that the driver can determine how many times a device was opened and for what reasons. The flags assume the open() routine may be called many times, but the close() routine should only be called on the last close() of a device.

  - **OTYP_BLK**  
    Close was through block interface for the device.

  - **OTYP_CHR**  
    Close was through the raw/character interface for the device.
OTYP_LYR Close a layered process (a higher-level driver called the close() routine of the device).

*cred_p Pointer to the user credential structure.

STREAMS *q Pointer to queue(9S) structure used to reference the read side of the driver. (A queue is the central node of a collection of structures and routines pointed to by a queue.)

flag File status flag.

*cred_p Pointer to the user credential structure.

**Description**

For STREAMS drivers, the close() routine is called by the kernel through the cb_ops(9S) table entry for the device. (Modules use the fmodsw table.) A non-null value in the d_str field of the cb_ops entry points to a streamtab structure, which points to a qinit(9S) containing a pointer to the close() routine. Non-STREAMS close() routines are called directly from the cb_ops table.

close() ends the connection between the user process and the device, and prepares the device (hardware and software) so that it is ready to be opened again.

A device may be opened simultaneously by multiple processes and the open() driver routine is called for each open. For all otyp values other than OTYP_LYR, the kernel calls the close() routine when the last-reference occurs. For OTYP_LYR each close operation will call the driver.

Kernel accounting for last-reference occurs at (dev, otyp) granularity. Note that a device is referenced once its associated open(9E) routine is entered, and thus open(9E)'s which have not yet completed will prevent close() from being called. The driver's close() call associated with the last-reference going away is typically issued as result of a close(2), exit(2), munmap(2), or umount(2). However, a failed open(9E) call can cause this last-reference close() call to be issued as a result of an open(2) or mount(2).

The kernel provides open() close() exclusion guarantees to the driver at the same devp, otyp granularity as last-reference accounting. The kernel delays new calls to the open() driver routine while the last-reference close() call is executing. For example, a driver that blocks in close() will not see new calls to open() until it returns from close(). This effectively delays invocation of other cb_ops(9S) driver entry points that also depend on an open(9E) established device reference. If the driver has indicated that an EINTR return is safe via the D_OPEN_RETURNS_EINTR cb_flag, then a delayed open() may be interrupted by a signal, resulting in an EINTR return from open() prior to calling open(9E).

Last-reference accounting and open() close() exclusion typically simplify driver writing. In some cases, however, they might be an impediment for certain types of drivers. To overcome any impediment, the driver can change minor numbers in open(9E), as described below, or
implement multiple minor nodes for the same device. Both techniques give the driver control over when close() calls occur and whether additional open() calls will be delayed while close() is executing.

In general, a close() routine should always check the validity of the minor number component of the dev parameter. The routine should also check permissions as necessary, by using the user credential structure (if pertinent), and the appropriateness of the flag and otyp parameter values.

close() could perform any of the following general functions:

- disable interrupts
- hang up phone lines
- rewind a tape
- deallocate buffers from a private buffering scheme
- unlock an unsharable device (that was locked in the open() routine)
- flush buffers
- notify a device of the close
- deallocate any resources allocated on open

The close() routines of STREAMS drivers and modules are called when a stream is dismantled or a module popped. The steps for dismantling a stream are performed in the following order. First, any multiplexor links present are unlinked and the lower streams are closed. Next, the following steps are performed for each module or driver on the stream, starting at the head and working toward the tail:

1. The write queue is given a chance to drain.
2. The close() routine is called.
3. The module or driver is removed from the stream.

Return Values close() should return 0 for success, or the appropriate error number. Return errors rarely occur, but if a failure is detected, the driver should decide whether the severity of the problem warrants either displaying a message on the console or, in worst cases, triggering a system panic. Generally, a failure in a close() routine occurs because a problem occurred in the associated device.

Notes If you use qwait_sig(9F), cv_wait_sig(9F) or cv_timedwait_sig(9F), you should note that close() may be called in contexts in which signals cannot be received. The ddi_can_receive_sig(9F) function is provided to determine when this hazard exists.

See Also close(2), fcntl(2), open(2), umount(2), detach(9E), open(9E), ddi_can_receive_sig(9F), cb_ops(9S), qinit(9S), queue(9S)

Writing Device Drivers

STREAMS Programming Guide
csx_event_handler(9E)

Name csx_event_handler – PC Card driver event handler

Synopsis #include <sys/pccard.h>

```c
int32_t prefixevent_handler(event_t event, int32_t priority,
     event_callback_args_t *args);
```

Interface Level Solaris architecture specific (Solaris DDI)

Parameters

- **event** The event.
- **priority** The priority of the event.
- **args** A pointer to the `event_callback_t` structure.

Description

Each instance of a PC Card driver must register an event handler to manage events associated with its PC Card. The driver event handler is registered using the `event_handler` field of the `client_req_t` structure passed to `csx_RegisterClient(9F)`. The driver may also supply a parameter to be passed to its event handler function using the `event_callback_args.client_data` field. Typically, this argument is the driver instance’s soft state pointer. The driver also registers which events it is interested in receiving through the `EventMask` field of the `client_req_t` structure.

Each event is delivered to the driver with a priority, `priority`. High priority events with `CS_EVENT_PRI_HIGH` set in `priority` are delivered above lock level, and the driver must use its high-level event mutex initialized with the `iblk_cookie` returned by `csx_RegisterClient(9F)` to protect such events. Low priority events with `CS_EVENT_PRI_LOW` set in `priority` are delivered below lock level, and the driver must use its low-level event mutex initialized with a NULL interrupt cookie to protect these events.

`csx_RegisterClient(9F)` registers the driver’s event handler, but no events begin to be delivered to the driver until after a successful call to `csx_RequestSocketMask(9F)`.

In all cases, Card Services delivers an event to each driver instance associated with a function on a multiple function PC Card.

Event Indications

The events and their indications are listed below; they are always delivered as low priority unless otherwise noted:

- **CS_EVENT_REGISTRATION_COMPLETE** A registration request processed in the background has been completed.
- **CS_EVENT_CARD_INSERTION** A PC Card has been inserted in a socket.
- **CS_EVENT_CARD_READY** A PC Card’s READY line has transitioned from the busy to ready state.
CS_EVENT_CARD_REMOVAL  A PC Card has been removed from a socket. This event is delivered twice; first as a high priority event, followed by delivery as a low priority event. As a high priority event, the event handler should only note that the PC Card is no longer present to prevent accesses to the hardware from occurring. As a low priority event, the event handler should release the configuration and free all I/O, window and IRQ resources for use by other PC Cards.

CS_EVENT_BATTERY_LOW  The battery on a PC Card is weak and is in need of replacement.

CS_EVENT_BATTERY_DEAD  The battery on a PC Card is no longer providing operational voltage.

CS_EVENT_PM_RESUME  Card Services has received a resume notification from the system’s Power Management software.

CS_EVENT_PM_SUSPEND  Card Services has received a suspend notification from the system’s Power Management software.

CS_EVENT_CARD_LOCK  A mechanical latch has been manipulated preventing the removal of the PC Card from the socket.

CS_EVENT_CARD_UNLOCK  A mechanical latch has been manipulated allowing the removal of the PC Card from the socket.

CS_EVENT_EJECTION_REQUEST  A request that the PC Card be ejected from a socket using a motor-driven mechanism.

CS_EVENT_EJECTION_COMPLETE  A motor has completed ejecting a PC Card from a socket.

CS_EVENT_ERASE_COMPLETE  A queued erase request that is processed in the background has been completed.

CS_EVENT_INSERTION_REQUEST  A request that a PC Card be inserted into a socket using a motor-driven mechanism.

CS_EVENT_INSERTION_COMPLETE  A motor has completed inserting a PC Card in a socket.

CS_EVENT_CARD_RESET  A hardware reset has occurred.

CS_EVENT_RESET_REQUEST  A request for a physical reset by a client.

CS_EVENT_RESET_COMPLETE  A reset request that is processed in the background has been completed.
CS_EVENT_RESET_PHYSICAL
A reset is about to occur.

CS_EVENT_CLIENT_INFO
A request that the client return its client information data. If
GET_CLIENT_INFO_SUBSVC(args->client_info.Attributes)
is equal to CS_CLIENT_INFO_SUBSVC_CS, the driver
should fill in the other fields in the client_info structure as described below, and return
CS_SUCCESS. Otherwise, it should return
CS_UNSUPPORTED_EVENT.

args->client_data.Attributes                        Must be
Must be OR'ed with
CS_CLIENT_INFO_VALID.

args->client_data.Revision                           Must be set
to a
driver-private version
number.

args->client_data.CSLevel                           Must be set
to
CS_VERSION.

args->client_data.RevDate                           Must be set
to the
revision date
of the PC
Card driver,
using
CS_CLIENT_INFO_MAKE_DATE(day, month, year).

day must be
the day of the
month,
month must be
the month of the year,
and year
must be the
year, offset
from a base
of 1980. For
every field, this
could be
set to a revision date of July 4 1997 with
CS_CLIENT_INFO_MAKE_DATE(4, 7, 17).

args->client_data.ClientName A string describing the PC Card driver should be copied into this space.

args->client_data.VendorName A string supplying the name of the PC Card driver vendor should be copied into this space.

args->client_data.DriverName A string supplying the name of the PC Card driver will be copied into this space by Card Services after the PC Card driver has successfully processed this event; the driver does not need to initialize this field.
The write protect status of the PC Card in the indicated socket has changed. The current write protect state of the PC Card is in the `args->info` field:

- **CS_EVENT_WRITE_PROTECT_WPOFF**: Card is not write protected.
- **CS_EVENT_WRITE_PROTECT_WPON**: Card is write protected.

The structure members of `event_callback_args_t` are:

- `void *info; /* event-specific information */`
- `void *client_data; /* driver-private data */`
- `client_info_t client_info; /* client information*/`

The structure members of `client_info_t` are:

- `unit32_t Attributes; /* attributes */`
- `unit32_t Revisions; /* version number */`
- `uint32_t CSLevel; /* Card Services version */`
- `uint32_t RevDate; /* revision date */`
- `char ClientName[CS_CLIENT_INFO_MAX_NAME_LEN]; /* PC Card driver description */`
- `char VendorName[CS_CLIENT_INFO_MAX_NAME_LEN]; /* PC Card driver vendor name */`
- `char DriverName[MODMAXNAMELEN]; /* PC Card driver name */`

**Return Values**

- **CS_SUCCESS**: The event was handled successfully.
- **CS_UNSUPPORTED_EVENT**: Driver does not support this event.
- **CS_FAILURE**: Error occurred while handling this event.

**Examples**

```c
static int
xx_event(event_t event, int priority, event_callback_args_t *args)
{
    int rval;
    struct xxx *xxx = args->client_data;
    client_info_t *info = &args->client_info;

    switch (event) {
    case CS_EVENT_REGISTRATION_COMPLETE:
        ASSERT(priority & CS_EVENT_PRI_LOW);
```
mutex_enter(&xxx->event_mutex);
xxx->card_state |= XX_REGISTRATION_COMPLETE;
mutex_exit(&xxx->event_mutex);
rval = CS_SUCCESS;
break;

case CS_EVENT_CARD_READY:
    ASSERT(priority & CS_EVENT_PRI_LOW);
rval = xx_card_ready(xxx);
mutex_exit(&xxx->event_mutex);
break;

case CS_EVENT_CARD_INSERTION:
    ASSERT(priority & CS_EVENT_PRI_LOW);
mutex_enter(&xxx->event_mutex);
rval = xx_card_insertion(xxx);
mutex_exit(&xxx->event_mutex);
break;

case CS_EVENT_CARD_REMOVAL:
    if (priority & CS_EVENT_PRI_HIGH) {
        mutex_enter(&xxx->hi_event_mutex);
        xxx->card_state &= ~XX_CARD_PRESENT;
mutex_exit(&xxx->hi_event_mutex);
    } else {
        mutex_enter(&xxx->event_mutex);
rval = xx_card_removal(xxx);
mutex_exit(&xxx->event_mutex);
    }
break;

case CS_EVENT_CLIENT_INFO:
    ASSERT(priority & CS_EVENT_PRI_LOW);
    if (GET_CLIENT_INFO_SUBSVC_CS(info->Attributes) ==
        CS_CLIENT_INFO_SUBSVC_CS) {
        info->Attributes |= CS_CLIENT_INFO_VALID;
        info->Revision = 4;
        info->CSLevel = CS_VERSION;
        info->RevDate = CS_CLIENT_INFO_MAKE_DATE(4, 7, 17);
        (void)strncpy(info->ClientName,
            "WhizBang Ultra Zowie PC card driver",
            CS_CLIENT_INFO_MAX_NAME_LEN)
        "ACME PC card drivers, Inc.",
        CS_CLIENT_INFO_MAX_NAME_LEN);
rval = CS_SUCCESS;
    } else {

case CS_EVENT_WRITE_PROTECT:
    ASSERT(priority & CS_EVENT_PRI_LOW);
    mutex_enter(&xxx->event_mutex);
    if (args->info == CS_EVENT_WRITE_PROTECT_WPOFF) {
        xxx->card_state &= ~XX_WRITE_PROTECTED;
    } else {
        xxx->card_state |= XX_WRITE_PROTECTED;
    }
    mutex_exit(&xxx->event_mutex);
    rval = CS_SUCCESS;
    break;

default:
    rval = CS_UNSUPPORTED_EVENT;
    break;

return (rval);

See Also  csx_Event2Text(9F), csx_RegisterClient(9F), csx_RequestSocketMask(9F)

*PC Card 95 Standard, PCMCIA/JEIDA*
**Name**
detach – detach or suspend a device

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

int prefix detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

**Interface Level**
Solaris DDI specific (Solaris DDI)

**Parameters**
- **dip**
  A pointer to the device's dev_info structure.
- **cmd**
  Type of detach; the driver should return DDI_FAILURE if any value other than DDI_DETACH or DDI_SUSPEND is passed to it.

**Description**
The detach() function complements the attach(9E) routine.

**DDI_DETACH**
If **cmd** is set to DDI_DETACH, detach() is used to remove the state associated with a given instance of a device node prior to the removal of that instance from the system.

The detach() function will be called once for each instance of the device for which there has been a successful attach(), once there are no longer any opens on the device. An attached instance of a driver can be successfully detached only once. The detach() function should clean up any per instance data initialized in attach(9E) and call kmem_free(9F) to free any heap allocations. For information on how to unregister interrupt handlers, see ddi_add_intr(9F). This should also include putting the underlying device into a quiescent state so that it will not generate interrupts.

Drivers that set up timeout(9F) routines should ensure that they are cancelled before returning DDI_SUCCESS from detach().

If detach() determines a particular instance of the device cannot be removed when requested because of some exceptional condition, detach() must return DDI_FAILURE, which prevents the particular device instance from being detached. This also prevents the driver from being unloaded. A driver instance failing the detach must ensure that no per instance data or state is modified or freed that would compromise the system or subsequent driver operation.

The system guarantees that the function will only be called for a particular dev_info node after (and not concurrently with) a successful attach(9E) of that device. The system also guarantees that detach() will only be called when there are no outstanding open(9E) calls on the device.

**DDI_SUSPEND**
The DDI_SUSPEND **cmd** is issued when the entire system is being suspended and power removed from it or when the system must be made quiescent. It will be issued only to devices which have a reg property or which export a pm-hardware-state property with the value needs-suspend-resume.
If `cmd` is set to `DDI_SUSPEND`, `detach()` is used to suspend all activity of a device before power is (possibly) removed from the device. The steps associated with suspension must include putting the underlying device into a quiescent state so that it will not generate interrupts or modify or access memory. Once quiescence has been obtained, `detach()` can be called with outstanding `open(9E)` requests. It must save the hardware state of the device to memory and block incoming or existing requests until `attach()` is called with `DDI_RESUME`.

If the device is used to store file systems, then after `DDI_SUSPEND` is issued, the device should still honor `dump(9E)` requests as this entry point may be used by suspend-resume operation (see `cpr(7)`) to save state file. It must do this, however, without disturbing the saved hardware state of the device.

If the device driver uses automatic device Power Management interfaces (driver exports `pm-components(9P)` property), it might need to call `pm_raise_power(9F)` if the current power level is lower than required to complete the `dump(9E)` request.

Before returning successfully from a call to `detach()` with a command of `DDI_SUSPEND`, the driver must cancel any outstanding timeouts and make any driver threads quiescent.

If `DDI_FAILURE` is returned for the `DDI_SUSPEND cmd`, either the operation to suspend the system or to make it quiescent will be aborted.

**Return Values**

- **DDI_SUCCESS**  
  For `DDI_DETACH`, the state associated with the given device was successfully removed. For `DDI_SUSPEND`, the driver was successfully suspended.

- **DDI_FAILURE**  
  The operation failed or the request was not understood. The associated state is unchanged.

**Context**  
This function is called from user context only.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

**See Also**  
`cpr(7), pm(7D), pm(9P), pm-components(9P), attach(9E), dump(9E), open(9E), power(9E), ddi_add_intr(9F), ddi_dev_is_needed(9F), ddi_map_regs(9F), kmem_free(9F), pm_raise_power(9F), timeout(9F)`

*Writing Device Drivers*
Name  devmap – validate and translate virtual mapping for memory mapped device

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

int prefixdevmap(dev_t dev, devmap_cookie_t dhp, offset_t off,
    size_t len, size_t *maplen, uint_t model);

Interface Level  Solaris DDI specific (Solaris DDI).

Parameters  

dev  Device whose memory is to be mapped.
dhp  An opaque mapping handle that the system uses to describe the mapping.
off  User offset within the logical device memory at which the mapping begins.
len  Length (in bytes) of the mapping to be mapped.
maplen  Pointer to length (in bytes) of mapping that has been validated. maplen is less
        than or equal to len.
model  The data model type of the current thread.

Description  devmap() is a required entry point for character drivers supporting memory-mapped devices
if the drivers use the devmap framework to set up the mapping. A memory mapped device has
memory that can be mapped into a process's address space. The mmap(2) system call, when
applied to a character special file, allows this device memory to be mapped into user space for
direct access by the user applications.

As a result of a mmap(2) system call, the system calls the devmap() entry point during the
mapping setup when _D_DEVMAP is set in the cb_flag field of the cb_ops(9S) structure, and any
of the following conditions apply:
  ■ ddi_devmap_segmap(9F) is used as the segmap(9E) entry point.
  ■ segmap(9E) entry point is set to NULL.
  ■ mmap(9E) entry point is set to NULL.

Otherwise EINVAL will be returned to mmap(2).

Device drivers should use devmap() to validate the user mappings to the device, to translate
the logical offset, off, to the corresponding physical offset within the device address space, and
to pass the mapping information to the system for setting up the mapping.

dhp is a device mapping handle that the system uses to describe a mapping to a memory that is
either contiguous in physical address space or in kernel virtual address space. The system may
create multiple mapping handles in one mmap(2) system call (for example, if the mapping
contains multiple physically discontiguous memory regions).
model returns the C Language Type Model which the current thread expects. It is set to
DDI_MODEL_TLP32 if the current thread expects 32-bit (ILP32) semantics, or DDI_MODEL_LP64
if the current thread expects 64-bit (LP64) semantics. model is used in combination with
ddi_model_convert_from(9F) 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.

devmap() should return EINVAL if the logical offset, off, is out of the range of memory exported
by the device to user space. If off + len exceeds the range of the contiguous memory, devmap ()
should return the length from off to the end of the contiguous memory region. The system will
repeatedly call devmap() until the original mapping length is satisfied. The driver sets *maplen
to the validated length which must be either less than or equal to len.

The devmap() entry point must initialize the mapping parameters before passing them to the
system through either devmap_devmem_setup(9F) (if the memory being mapped is device
memory) or devmap_umem_setup(9F) (if the memory being mapped is kernel memory). The
devmap() entry point initializes the mapping parameters by mapping the control callback
structure (see devmap_callback_ctl(9S)), the device access attributes, mapping length,
maximum protection possible for the mapping, and optional mapping flags. See
devmap_devmem_setup(9F) and devmap_umem_setup(9F) for further information on
initializing the mapping parameters.

The system will copy the driver’s devmap_callback_ctl(9S) data into its private memory so
the drivers do not need to keep the data structure after the return from either
devmap_devmem_setup(9F) or devmap_umem_setup(9F).

For device mappings, the system establishes the mapping to the physical address that
corresponds to off by passing the register number and the offset within the register address
space to devmap_devmem_setup(9F).

For kernel memory mapping, the system selects a user virtual address that is aligned with the
kernel address being mapped for cache coherence.

Return Values  0          Successful completion.
               Non-zero   An error occurred.

Examples  EXAMPLE 1  Implementing the devmap() Entry Point

The following is an example of the implementation for the devmap() entry point. For mapping
device memory, devmap() calls devmap_devmem_setup(9F) with the register number,
runumber, and the offset within the register, roff. For mapping kernel memory, the driver must
first allocate the kernel memory using ddi_umem_alloc(9F). For example,
ddi_umem_alloc(9F) can be called in the attach(9E) routine. The resulting kernel memory
cookie is stored in the driver soft state structure, which is accessible from the devmap() entry
point. See ddi_soft_state(9F). devmap() passes the cookie obtained from
ddi_umem_alloc(9F) and the offset within the allocated kernel memory to
devmap_umem_setup(9F). The corresponding ddi_umem_free(9F) can be made in the
detach(9E) routine to free up the kernel memory.

```
... 
#define MAPPING_SIZE 0x2000 /* size of the mapping */
#define MAPPING_START 0x70000000 /* logical offset at beginning of the mapping */
static
struct devmap_callback_ctl xxmap_ops = {
    DEVMAP_OPS_REV, /* devmap_ops version number */
    xxmap_map, /* devmap_ops map routine */
    xxmap_access, /* devmap_ops access routine */
    xxmap_dup, /* devmap_ops dup routine */
    xxmap_unmap, /* devmap_ops unmap routine */
};

static int
xxdevmap(dev_t dev, devmap_cookie_t dhp, offset_t off, size_t len,
        size_t *maplen, uint_t model)
{
    int instance;
    struct xxstate *xsp;
    struct ddi_device_acc_attr *endian_attr;
    struct devmap_callback_ctl *callbackops = NULL;
    ddi_umem_cookie_t cookie;
    dev_info_t *dip;
    offset_t roff;
    offset_t koff;
    uint_t rnumber;
    uint_t maxprot;
    uint_t flags = 0;
    size_t length;
    int err;
    /* get device soft state */
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (-1);
    dip = xsp->dip;
    /* check for a valid offset */
    if (off is invalid)
```
EXAMPLE 1  Implementing the devmap() Entry Point  (Continued)

    return (-1);
    /* check if len is within the range of contiguous memory */
    if ( (off + len) is contiguous.)
        length = len;
    else
        length = MAPPING_START + MAPPING_SIZE - off;

    /* device access attributes */
    endian_attr = xsp->endian_attr;

    if ( off is referring to a device memory |
        |
        |
        |
        |
        |
        |
        rnumber = XXX; /* index to register set at off */
        roff = XXX; /* offset of rnumber at local bus */
        callbacks = &xxmap_ops; /* do all callbacks for this mapping */
        maxprot = PROT_ALL; /* allowing all access */
        if ((err = devmap_devmem_setup(dhp, dip, callbacks, rnumber, roff,
            length, maxprot, flags, endian_attr)) < 0)
            return (err);
    )

    else if ( off is referring to a kernel memory: |
        |
        |
        |
        |
        |
        |
        cookie = xsp->cookie; /* cookie is obtained from |
            ddi_umem_alloc(9F) */
        koff = XXX; /* offset within the kernel memory. */
        callbacks = NULL; /* don’t do callback for this mapping */
        maxprot = PROT_ALL; /* allowing all access */
        if ((err = devmap_umem_setup(dhp, dip, callbacks, cookie, koff,
            length, maxprot, flags, endian_attr)) < 0)
            return (err);
    )

    *maplen = length;
    return (0);
}

See Also  mmap(2), attach(9E), detach(9E), mmap(9E), segmap(9E), ddi_devmap_segmap(9F),
    ddi_model_convert_from(9F), ddi_soft_state(9F), ddi_umem_alloc(9F),
    ddi_umem_free(9F), devmap_devmem_setup(9F), devmap_setup(9F),
    devmap_umem_setup(9F), cb_ops(9S), devmap_callback_ctl(9S)

    Writing Device Drivers
# devmap_access(9E)

## Name
devmap_access – device mapping access entry point

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

int prefixdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len, uint_t type, uint_t rw);
```

## Interface Level
Solaris DDI specific (Solaris DDI).

## 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.
- **type**
  Type of access operation. Possible values are:
  - DEVMAP_ACCESS: Memory access.
  - DEVMAP_LOCK: Lock the memory being accessed.
  - DEVMAP_UNLOCK: Unlock the memory being accessed.
- **rw**
  Direction of access. Possible values are:
  - DEVMAP_READ: Read access attempted.
  - DEVMAP_WRITE: Write access attempted.
  - DEVMAP_EXEC: Execution access attempted.

## Description
The devmap_access() entry point is an optional routine. It notifies drivers whenever an access is made to a mapping described by dhp that has not been validated or does not have sufficient protection for the access. The system expects devmap_access() to call either devmap_do_ctxmgt(9F) or devmap_default_access(9F) to load the memory address translations before it returns. For mappings that support context switching, device drivers should call devmap_do_ctxmgt(9F). For mappings that do not support context switching, the drivers should call devmap_default_access(9F).

In devmap_access(), drivers perform memory access related operations such as context switching, checking the availability of the memory object, and locking and unlocking the memory object being accessed. The devmap_access() entry point is set to NULL if no operations need to be performed.

pvtp is a pointer to the driver’s private mapping data that was allocated and initialized in the devmap_map(9E) entry point.
off and len define the range to be affected by the operations in devmap_access(). type defines the type of operation that device drivers should perform on the memory object. If type is either DEVMAP_LOCK or DEVMAP_UNLOCK, the length passed to either devmap_do_ctxmgt(9F) or devmap_default_access(9F) must be same as len. rw specifies the direction of access on the memory object.

A non-zero return value from devmap_access() may result in a SIGSEGV or SIGBUS signal being delivered to the process.

**Return Values**

devmap_access() returns the following values:

0 Successful completion.

Non-zero An error occurred. The return value from devmap_do_ctxmgt(9F) or devmap_default_access(9F) should be returned.

**Examples**

**EXAMPLE 1** devmap_access() entry point

The following is an example of the devmap_access() entry point. If the mapping supports context switching, devmap_access() calls devmap_do_ctxmgt(9F). Otherwise, devmap_access() calls devmap_default_access(9F).

```c
#define OFF_DO_CTXMGT 0x40000000
#define OFF_NORMAL 0x40100000
#define CTXMTGT_SIZE 0x100000
#define NORMAL_SIZE 0x100000

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

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, uint_t type, uint_t rw)
{
```
EXAMPLE 1  devmap_access() entrypoint  (Continued)

offset_t diff;
int err;

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

    return (err);
}

See Also  devmap_map(9E),devmap_default_access(9F),devmap_do_ctxmgt(9F),
          devmap_callback_ctl(9S)

Writing Device Drivers
# devmap_contextmgt

**Name**

devmap_contextmgt – driver callback function for context management

**Synopsis**

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

int devmap_contextmgt(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, uint_t type, uint_t rw);
```

**Interface Level**

Solaris DDI specific (Solaris DDI).

**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.

- **type**
  Type of access operation. Possible values are:
  - `DEVMAP_ACCESS`: Memory access.
  - `DEVMAP_LOCK`: Lock the memory being accessed.
  - `DEVMAP_UNLOCK`: Unlock the memory being accessed.

- **rw**
  Direction of access. Possible values are:
  - `DEVMAP_READ`: Read access attempted.
  - `DEVMAP_WRITE`: Write access attempted.

**Description**

`devmap_contextmgt()` is a driver-supplied function that performs device context switching on a mapping. Device drivers pass `devmap_contextmgt()` as an argument to `devmap_do_ctxmgt(9F)` in the `devmap_access(9E)` entry point. The system will call `devmap_contextmgt()` when memory is accessed. The system expects `devmap_contextmgt()` to load the memory address translations of the mapping by calling `devmap_load(9F)` before returning.

`dhp` uniquely identifies the mapping and is used as an argument to `devmap_load(9F)` to validate the mapping. `off` and `len` define the range to be affected by the operations in `devmap_contextmgt()`.

The driver must check if there is already a mapping established at `off` that needs to be unloaded. If a mapping exists at `off`, `devmap_contextmgt()` must call `devmap_unload(9F)` on the current mapping. `devmap_unload(9F)` must be followed by `devmap_load()` on the mapping that generated this call to `devmap_contextmgt()`. `devmap_unload(9F)` unloads the current mapping so that a call to `devmap_access(9E)`, which causes the system to call `devmap_contextmgt()`, will be generated the next time the mapping is accessed.
pvtp is a pointer to the driver’s private mapping data that was allocated and initialized in the devmap_map(9E) entry point. type defines the type of operation that device drivers should perform on the memory object. If type is either DEVMAP_LOCK or DEVMAP_UNLOCK, the length passed to either devmap_unload(9F) or devmap_load(9F) must be same as len. rw specifies the access direction on the memory object.

A non-zero return value from devmap_contextmgt() will be returned to devmap_access(9E) and will cause the corresponding operation to fail. The failure may result in a SIGSEGV or SIGBUS signal being delivered to the process.

Return Values

0 Successful completion.
Non-zero An error occurred.

Examples

EXAMPLE 1 managing a device context

The following shows an example of managing a device context.

```c
struct xxcontext cur_ctx;
static int
xxdevmap_contextmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, uint_t type, uint_t rw)
{
    devmap_cookie_t cur_dhp;
    struct xxpvtdata *p;
    struct xxpvtdata *pvp = (struct xxpvtdata *)pvtp;
    struct xx_softc *softc = pvp->softc;
    int err;

    mutex_enter(&softc->mutex);

    /*
    * invalidate the translations of current context before
    * switching context.
    */
    if (cur_ctx != NULL && cur_ctx != pvp->ctx) {
        p = cur_ctx->pvt;
        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;

    /*
    * Load the address translations of the calling context.
    */
```
EXAMPLE 1  managing a device context  (Continued)

    */
    err = devmap_load(pvp->dhp, off, len, type, rw);

    mutex_exit(&sof tc->mutex);

    return (err);
}

See Also  devmap_access(9E), devmap_do_ctxmgt(9F) devmap_load(9F), devmapUnload(9F)

Writing Device Drivers
# devmap_dup – device mapping duplication entry point

## Synopsis

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

int prefixdevmap_dup(devmap_cookie_t dhp, void *pvtp,
                      devmap_cookie_t new_dhp, void **new_pvtp);
```

## Interface Level
Solaris DDI specific (Solaris DDI).

## Arguments

- **dhp** An opaque mapping handle that the system uses to describe the mapping currently being duplicated.
- **pvtp** Driver private mapping data for the mapping currently being duplicated.
- **new_dhp** An opaque data structure that the system uses to describe the duplicated device mapping.
- **new_pvtp** A pointer to be filled in by device drivers with the driver private mapping data for the duplicated device mapping.

## Description

The system calls `devmap_dup()` when a device mapping is duplicated, such as during the execution of the `fork(2)` system call. The system expects `devmap_dup()` to generate new driver private data for the new mapping, and to set `new_pvtp` to point to it. `new_dhp` is the handle of the new mapped object.

A non-zero return value from `devmap_dup()` will cause a corresponding operation such as `fork()` to fail.

## Return Values

- **devmap_dup()** returns the following values:
  - **0** Successful completion.
  - Non-zero An error occurred.

## Examples

```c
static int
xxdevmap_dup(devmap_cookie_t dhp, void *pvtp, \
              devmap_cookie_t new_dhp, \
              void **new_pvtp)
{
    struct xxpvtdata *prvtdata;
    struct xxpvtdata *p = (struct xxpvtdata *)pvtp;
    struct xx_softc *softc = p->softc;
    mutex_enter(&softc->mutex);
    /* Allocate a new private data structure */
    prvtdata = kmalloc(sizeof (struct xxpvtdata), KM_SLEEP);
    /* Return the new data */
    prvtdata->off = p->off;
    /* ...
```
prvtdata->len = p->len;
prvtdata->ctx = p->ctx;
prvtdata->dhp = new_dhp;
prvtdata->softc = p->softc;
*new_pvtp = prvtdata;
mutex_exit(&softc->mutex);
return (0);
}

See Also  fork(2), devmap_callback_ctl(9S)

Writing Device Drivers
### Synopsis

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

int prefixdevmap_map(devmap_cookie_t dhp, dev_t dev,
          uint_t flags, offset_t off, size_t len, void **pvtp);
```

### Interface Level
Solaris DDI specific (Solaris DDI).

### Arguments

- **dhp**: An opaque mapping handle that the system uses to describe the mapping currently being created.
- **dev**: The device whose memory is to be mapped.
- **flags**: Flags indicating type of mapping. Possible values are:
  - `MAP_PRIVATE`: Changes are private.
  - `MAP_SHARED`: Changes should be shared.
- **off**: User offset within the logical device memory at which the mapping begins.
- **len**: Length (in bytes) of the memory to be mapped.
- **pvtp**: A pointer to be filled in by device drivers with the driver private mapping data.

### Description
The `devmap_map()` entry point is an optional routine that allows drivers to perform additional processing or to allocate private resources during the mapping setup time. For example, in order for device drivers to support context switching, the drivers allocate private mapping data and associate the private data with the mapping parameters in the `devmap_map()` entry point.

The system calls `devmap_map()` after the user mapping to device physical memory has been established. (For example, after the `devmap(9E)` entry point is called.)

`devmap_map()` receives a pointer to the driver private data for this mapping in `pvtp`. The system expects the driver to allocate its private data and set `*pvtp` to the allocated data. The driver must store `off` and `len`, which define the range of the mapping, in its private data. Later, when the system calls `devmap_unmap(9E)`, the driver will use the `off` and `len` stored in `pvtp` to check if the entire mapping, or just a part of it, is being unmapped. If only a part of the mapping is being unmapped, the driver must allocate a new private data for the remaining mapping before freeing the old private data. The driver will receive `*pvtp` in subsequent event notification callbacks.

If the driver supports context switching, it should store the mapping handle `dhp` in its private data `*pvtp` for later use in `devmap_unload(9F)`.
For a driver that supports context switching, \textit{flags} indicates whether or not the driver should allocate a private context for the mapping. For example, a driver may allocate a memory region to store the device context if \textit{flags} is set to MAP\textsubscript{PRIVATE}.

\textbf{Return Values} \texttt{devmap\_map()} returns the following values:

- 0 \hspace{1em} Successful completion.
- Non-zero \hspace{1em} An error occurred.

\textbf{Examples} \texttt{EXAMPLE1 devmap\_map()} implementation

The following shows an example implementation for \texttt{devmap\_map()}.

\begin{verbatim}
static int xxdevmap_map(devmap_cookie_t dhp, dev_t dev, uint_t flags, 
    offset_t off, size_t len, void **pvtp)
{
    struct xx_resources *pvt;
    struct xx_context *this_context;
    struct xx_softc *softc;
    softc = ddi_get_soft_state(statep, getminor(dev));
    this_context = get_context(softc, off, len);
    /* allocate resources for the mapping - Device dependent */
    pvt = kmem_zalloc(sizeof (struct xx_resources), KM_SLEEP);
    pvt->off = off;
    pvt->len = len;
    pvt->dhp = dhp;
    pvt->ctx = this_context;
    *pvtp = pvt;
}
\end{verbatim}

\textbf{See Also} \texttt{devmap\_unmap(9E), devmap\_unload(9F), devmap\_callback\_ctl(9S)}

\textit{Writing Device Drivers}
#include <sys/ddi.h>
#include <sys/sunddi.h>

void prefixdevmap_unmap(devmap_cookie_t dhp, void *pvtp, offset_t off, size_t len, devmap_cookie_t new_dhp1, void **new_pvtp1, devmap_cookie_t new_dhp2, void **new_pvtp2);

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 unmapping begins.
- **len**: Length (in bytes) of the memory being unmapped.
- **new_dhp1**: The opaque mapping handle that the system uses to describe the new region that ends at (off - 1). **new_dhp1** may be NULL.
- **new_pvtp1**: A pointer to be filled in by the driver with the driver private mapping data for the new region that ends at (off - 1); ignored if **new_dhp1** is NULL.
- **new_dhp2**: The opaque mapping handle that the system uses to describe the new region that begins at (off + len); **new_dhp2** may be NULL.
- **new_pvtp2**: A pointer to be filled in by the driver with the driver private mapping data for the new region that begins at (off + len); ignored if **new_dhp2** is NULL.

**Description**

devmap_unmap() is called when the system removes the mapping in the range [off, off + len], such as in the munmap(2) or exit(2) system calls. Device drivers use devmap_unmap() to free up the resources allocated in devmap_map(9E).

dhp is the mapping handle that uniquely identifies the mapping. The driver stores the mapping attributes in the driver’s private data, pvtp, when the mapping is created. See devmap_map(9E) for details.

off and len define the range to be affected by devmap_unmap(). This range is within the boundary of the mapping described by dhp.

If the range [off, off + len] covers the entire mapping, the system passes NULL to new_dhp1, new_pvtp1, new_dhp2, and new_pvtp2. The system expects device drivers to free all resources allocated for this mapping.

If off is at the beginning of the mapping and len does not cover the entire mapping, the system sets NULL to new_dhp1 and to new_pvtp1. The system expects the drivers to allocate new driver
private data for the region that starts at \( off + len \) and to set \(*new_pvtp2\) to point to it. \( new_dhp2 \) is the mapping handle of the newly mapped object.

If \( off \) is not at the beginning of the mapping, but \( off + len \) is at the end of the mapping the system passes NULL to \( new_dhp2 \) and \( new_pvtp2 \). The system then expects the drivers to allocate new driver private data for the region that begins at the beginning of the mapping (for example, stored in \( pvtp \)) and to set \(*new_pvtp1\) to point to it. \( new_dhp1 \) is the mapping handle of the newly mapped object.

The drivers should free up the driver private data, \( pvtp \), previously allocated in \texttt{devmap_map(9E)} before returning to the system.

**Examples**

**EXAMPLE1** \texttt{devmap_unmap()} implementation

```c
static void
xxdevmap_unmap(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, devmap_cookie_t new_dhp1, void **new_pvtp1,
    devmap_cookie_t new_dhp2, void **new_pvtp2)
{
    struct xxpvtdata *ptmp;
    struct xxpvtdata *p = (struct xxpvtdata *)pvtp;
    struct xx_softc *softc = p->softc;
    mutex_enter(&softc->mutex);
    /*
    * If new_dhp1 is not NULL, create a new driver private data
    * for the region from the beginning of old mapping to off.
    */
    if (new_dhp1 != NULL) {
        ptmp = kmem_zalloc(sizeof (struct xxpvtdata), KM_SLEEP);
        ptmp->dhp = new_dhp1;
        ptmp->off = pvtp->off;
        ptmp->len = off - pvtp->off;
        *new_pvtp1 = ptmp;
    }
    /*
    * If new_dhp2 is not NULL, create a new driver private data
    * for the region from off+len to the end of the old mapping.
    */
    if (new_dhp2 != NULL) {
        ptmp = kmem_zalloc(sizeof (struct xxpvtdata), KM_SLEEP);
        ptmp->off = off + len;
        ptmp->len = pvtp->len - (off + len - pvtp->off);
        ptmp->dhp = new_dhp2;
        *new_pvtp2 = ptmp;
    }
}
```

Driver Entry Points 53
EXAMPLE 1  devmap_unmap() implementation  (Continued)

    /* Destroy the driver private data - Device dependent */
    ...
    kmem_free(pvtp, sizeof (struct xxpvtdata));
    mutex_exit(&softc->mutex);
}

See Also  exit(2), munmap(2), devmap_map(9E), devmap_callback_ctl(9S)

Writing Device Drivers
dump – dump memory to device during system failure

Synopsis

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

int dump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk);
```

Interface Level

Solaris specific (Solaris DDI). This entry point is required. For drivers that do not implement `dump()` routines, `nodev(9F)` should be used.

Arguments

- `dev`: Device number.
- `addr`: Address for the beginning of the area to be dumped.
- `blkno`: Block offset to dump memory.
- `nblk`: Number of blocks to dump.

Description

dump() is used to dump a portion of virtual address space directly to a device in the case of system failure. It can also be used for checking the state of the kernel during a checkpoint operation. The memory area to be dumped is specified by `addr` (base address) and `nblk` (length). It is dumped to the device specified by `dev` starting at offset `blkno`. Upon completion `dump()` returns the status of the transfer.

When the system is panicking, the calls of functions scheduled by `timeout(9F)` and `ddi_trigger_softintr(9F)` will never occur. Neither can `delay(9F)` be relied upon, since it is implemented via `timeout()`. See `ddi_in_panic(9F)`.

dump() is called at interrupt priority.

Return Values

dump() returns 0 on success, or the appropriate error number.

See Also

cpr(7), nodev(9F)

Writing Device Drivers
### _fini(9E)

**Name**  
_fini, _info, _init – loadable module configuration entry points

**Synopsis**  
#include <sys/modctl.h>

```c
int _fini(void)
int _info(struct modinfo *modinfop);
int _init(void);
```

**Interface Level**  
Solaris DDI specific (Solaris DDI). These entry points are required. You must write them.

**Parameters**

- **_info()**  
  modinfop  
  A pointer to an opaque modinfo structure.

- **_init()**  
  Initializes a loadable module. It is called before any other routine in a loadable module. _init() returns the value returned by `mod_install(9F)`. The module may optionally perform some other work before the `mod_install(9F)` call is performed. If the module has done some setup before the `mod_install(9F)` function is called, then it should be prepared to undo that setup if `mod_install(9F)` returns an error.

  _info() returns information about a loadable module. _info() returns the value returned by `mod_info(9F)`.

  _fini() prepares a loadable module for unloading. It is called when the system wants to unload a module. If the module determines that it can be unloaded, then _fini() returns the value returned by `mod_remove(9F)`. Upon successful return from _fini() no other routine in the module will be called before _init() is called.

**Return Values**

- **_init()** should return the appropriate error number if there is an error, otherwise it should return the return value from `mod_install(9F)`.

  _info() should return the return value from `mod_info(9F)`

  _fini() should return the return value from `mod_remove(9F)`. _fini() is permitted to return EBUSY prior to calling `mod_remove(9F)` if the driver should not be unloaded. Driver global resources, such as mutexes and calls to `ddi_soft_state_fini(9F)`, should only be destroyed in _fini() after `mod_remove()` returns successfully.

**Examples**

- **EXAMPLE 1**  
  Initializing and Freeing a Mutex

  The following example demonstrates how to initialize and free a `mutex(9F)`.

  ```c
  #include <sys/modctl.h>
  #include <sys/ddi.h>
  #include <sys/sunddi.h>
  ```
EXAMPLE 1 Initializing and Freeing a Mutex  (Continued)

static struct dev_ops drv_ops;
/*
 * Module linkage information for the kernel.
 */
static struct modldr
static struct modldrv modldrv = {
    &mod_driverops, /* Type of module. This one is a driver */
    "Sample Driver",
    &drv_ops /* driver ops */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldr,
    NULL
};

/* Global driver mutex */
static kmutex_t xx_global_mutex;

int
_init(void)
{
    int i;

    /*
     * Initialize global mutex before mod_install'ing driver.
     * If mod_install() fails, must clean up mutex initialization
     */
    mutex_init(&xx_global_mutex, NULL,
               MUTEX_DRIVER, (void *)NULL);

    if ((i = mod_install(&modlinkage)) != 0) {
        mutex_destroy(&xx_global_mutex);
    }

    return (i);
}

int
_info(struct modinfo *modinfop)
{
EXAMPLE 1  Initializing and Freeing a Mutex  (Continued)

    return (mod_info(&modlinkage, modinfop));
  }

int
  _fini(void)
{
  int   i;

    /*
     * If mod_remove() is successful, we destroy our global mutex
     */
  if ((i = mod_remove(&modlinkage)) == 0) {
      mutex_destroy(&xx_global_mutex);
  }
  return (i);
}

See Also  add_drv(1M), mod_info(9F), mod_install(9F), mod_remove(9F), mutex(9F), modldr(9S),
           modlinkage(9S), modlstmod(9S)

Writing Device Drivers

Warnings  Do not change the structures referred to by the modlinkage structure after the call to
           mod_install(), as the system may copy or change them.

Notes  Even though the identifiers _fini(), _info(), and _init() appear to be declared as globals,
       their scope is restricted by the kernel to the module that they are defined in.

Bugs  On some implementations _info() may be called before _init().
## getinfo(9E)

<table>
<thead>
<tr>
<th>Name</th>
<th>getinfo – get device driver information</th>
</tr>
</thead>
</table>
| Synopsis | ```c
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixgetinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **resultp);
``` |
| Interface Level | Solaris DDI specific (Solaris DDI). This entry point is required for drivers which export `cb_ops(9S)` entry points. |
| Arguments | |
| `dip` | Do not use. |
| `cmd` | Command argument – valid command values are `DDI_INFO_DEV2DEVINFO` and `DDI_INFO_DEV2INSTANCE`. |
| `arg` | Command specific argument. |
| `resultp` | Pointer to where the requested information is stored. |
| Description | When `cmd` is set to `DDI_INFO_DEV2DEVINFO`, `getinfo()` should return the `dev_info_t` pointer associated with the `dev_t arg`. The `dev_info_t` pointer should be returned in the field pointed to by `resultp`. When `cmd` is set to `DDI_INFO_DEV2INSTANCE`, `getinfo()` should return the instance number associated with the `dev_t arg`. The instance number should be returned in the field pointed to by `resultp`. Drivers which do not export `cb_ops(9S)` entry points are not required to provide a `getinfo()` entry point, and may use `nodev(9F)` in the `devo_getinfo` field of the `dev_ops(9S)` structure. A SCSI HBA driver is an example of a driver which is not required to provide `cb_ops(9S)` entry points. |
| Return Values | `getinfo()` should return: |
| `DDI_SUCCESS` | on success. |
| `DDI_FAILURE` | on failure. |
| Examples | **EXAMPLE1** getinfo() implementation |
| | ```c
/*ARGSUSED*/
static int
rd_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, 
void **resultp)
{
    /* Note that in this simple example
     * the minor number is the instance
``` |
EXAMPLE 1  getinfo() implementation  (Continued)

    /* number. */
    devstate_t *sp;
    int error = DDI_FAILURE;
    switch (infocmd) {
        case DDI_INFO_DEVT2DEVINFO:
            if ((sp = ddi_get_soft_state(statep,
                getminor((dev_t) arg))) != NULL) {
                *resultp = sp->devi;
                error = DDI_SUCCESS;
            } else
                *result = NULL;
            break;

        case DDI_INFO_DEVT2INSTANCE:
            *resultp = (void *) (uintptr_t) getminor((dev_t) arg);
            error = DDI_SUCCESS;
            break;
    }

    return (error);
}

See Also  ddi_no_info(9F), nodev(9F), cb_ops(9S), dev_ops(9S)

Writing Device Drivers

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

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

int prefix_reset(gld_mac_info_t *macinfo);
int prefix_start(gld_mac_info_t *macinfo);
int prefix_stop(gld_mac_info_t *macinfo);
int prefix_set_mac_addr(gld_mac_info_t *macinfo, unsigned char *macaddr);
int prefix_set_multicast(gld_mac_info_t *macinfo, unsigned char *multicastaddr, int multiflag);
int prefix_set_promiscuous(gld_mac_info_t *macinfo, int promiscflag);
int prefix_send(gld_mac_info_t *macinfo, mblk_t *mp);
uint_t prefix_intr(gld_mac_info_t *macinfo);
int prefix_get_stats(gld_mac_info_t *macinfo, struct gld_stats *stats);
int prefix_ioctl(gld_mac_info_t *macinfo, queue_t *q, mblk_t *mp);
```

**Interface Level**

Solaris architecture specific (Solaris DDI).

**Parameters**

- **macinfo**
  - Pointer to a `gld_mac_info(9S)` structure.

- **macaddr**
  - Pointer to the beginning of a character array containing a valid MAC address. The array will be of the length specified by the driver in the `glm_addrlen` element of the `gld_mac_info(9S)` structure.

- **multicastaddr**
  - Pointer to the beginning of a character array containing a multicast, group, or functional address. The array will be of the length specified by the driver in the `glm_addrlen` element of the `gld_mac_info(9S)` structure.

- **multiflag**
  - A flag indicating whether reception of the multicast address is to be enabled or disabled. This argument is specified as `GLD_MULTI_ENABLE` or `GLD_MULTI_DISABLE`.

- **promiscflag**
  - A flag indicating what type of promiscuous mode, if any, is to be enabled. This argument is specified as `GLD_MAC_PROMISC_PHYS`, `GLD_MAC_PROMISC_MULTI`, or `GLD_MAC_PROMISC_NONE`. 
mp Pointer to a STREAMS message block containing the packet to be transmitted or the ioctl to be executed.

stats Pointer to a gld_stats(9S) structure to be filled in with the current values of statistics counters.

q Pointer to the queue(9S) structure to be used in the reply to the ioctl.

**Description**
These entry points must be implemented by a device-specific network driver designed to interface with the Generic LAN Driver (GLD).

As described in gld(7D), the main data structure for communication between the device-specific driver and the GLD module is the gld_mac_info(9S) structure. Some of the elements in that structure are function pointers to the entry points described here. The device-specific driver must, in its attach(9E) routine, initialize these function pointers before calling gld_register().

- **gldm_reset()** resets the hardware to its initial state.
- **gldm_start()** enables the device to generate interrupts and prepares the driver to call gld_recv() for delivering received data packets to GLD.
- **gldm_stop()** disables the device from generating any interrupts and stops the driver from calling gld_recv() for delivering data packets to GLD. GLD depends on the gldm_stop() routine to ensure that the device will no longer interrupt, and it must do so without fail.
- **gldm_set_mac_addr()** sets the physical address that the hardware is to use for receiving data. This function should program the device to the passed MAC address macaddr.
- **gldm_set_multicast()** enables and disables device-level reception of specific multicast addresses. If the third argument multiflag is set to GLD_MULTI_ENABLE, then the function sets the interface to receive packets with the multicast address pointed to by the second argument; if multiflag is set to GLD_MULTI_DISABLE, the driver is allowed to disable reception of the specified multicast address.

This function is called whenever GLD wants to enable or disable reception of a multicast, group, or functional address. GLD makes no assumptions about how the device does multicast support and calls this function to enable or disable a specific multicast address. Some devices may use a hash algorithm and a bitmask to enable collections of multicast addresses; this is allowed, and GLD will filter out any superfluous packets that are not required. If disabling an address could result in disabling more than one address at the device level, it is the responsibility of the device driver to keep whatever information it needs to avoid disabling an address that GLD has enabled but not disabled.

**gldm_set_multicast()** will not be called to enable a particular multicast address that is already enabled, nor to disable an address that is not currently enabled. GLD keeps track of
multiple requests for the same multicast address and only calls the driver’s entry point when the first request to enable, or the last request to disable a particular multicast address is made.

`gldm_set_promiscuous()` enables and disables promiscuous mode. This function is called whenever GLD wants to enable or disable the reception of all packets on the medium, or all multicast packets on the medium. If the second argument `promiscflag` is set to the value of `GLD_MAC_PROMISC_PHYS`, then the function enables physical-level promiscuous mode, resulting in the reception of all packets on the medium. If `promiscflag` is set to `GLD_MAC_PROMISC_MULTI`, then reception of all multicast packets will be enabled. If `promiscflag` is set to `GLD_MAC_PROMISC_NONE`, then promiscuous mode is disabled.

In the case of a request for promiscuous multicast mode, drivers for devices that have no multicast-only promiscuous mode must set the device to physical promiscuous mode to ensure that all multicast packets are received. In this case the routine should return `GLD_SUCCESS`. The GLD software will filter out any superfluous packets that are not required.

For forward compatibility, `gldm_set_promiscuous()` routines should treat any unrecognized values for `promiscflag` as though they were `GLD_MAC_PROMISC_PHYS`.

`gldm_send()` queues a packet to the device for transmission. This routine is passed a STREAMS message containing the packet to be sent. The message may comprise multiple message blocks, and the send routine must chain through all the message blocks in the message to access the entire packet to be sent. The driver should be prepared to handle and skip over any zero-length message continuation blocks in the chain. The driver should check to ensure that the packet does not exceed the maximum allowable packet size, and must pad the packet, if necessary, to the minimum allowable packet size. If the send routine successfully transmits or queues the packet, it should return `GLD_SUCCESS`.

The send routine should return `GLD_NORESOURCES` if it cannot immediately accept the packet for transmission; in this case GLD will retry it later. If `gldm_send()` ever returns `GLD_NORESOURCES`, the driver must, at a later time when resources have become available, call `gld_sched()` to inform GLD that it should retry packets that the driver previously failed to queue for transmission. (If the driver’s `gldm_stop()` routine is called, the driver is absolved from this obligation until it later again returns `GLD_NORESOURCES` from its `gldm_send()` routine; however, extra calls to `gld_sched()` will not cause incorrect operation.)

If the driver’s send routine returns `GLD_SUCCESS`, then the driver is responsible for freeing the message when the driver and the hardware no longer need it. If the send routine copied the message into the device, or into a private buffer, then the send routine may free the message after the copy is made. If the hardware uses DMA to read the data directly out of the message data blocks, then the driver must not free the message until the hardware has completed reading the data. In this case the driver will probably free the message in the interrupt routine, or in a buffer-reclaim operation at the beginning of a future send operation. If the send routine returns anything other than `GLD_SUCCESS`, then the driver must not free the message.
gldm_intr() is called when the device might have interrupted. Since it is possible to share
interrupts with other devices, the driver must check the device status to determine whether it
actually caused an interrupt. If the device that the driver controls did not cause the interrupt,
then this routine must return DDI_INTR_UNCLAIMED. Otherwise it must service the interrupt
and should return DDI_INTR_CLAIMED. If the interrupt was caused by successful receipt of a
packet, this routine should put the received packet into a STREAMS message of type M_DATA
and pass that message to gld_recv().

gld_recv() will pass the inbound packet upstream to the appropriate next layer of the
network protocol stack. It is important to correctly set the b_rptr and b_wptr members of the
STREAMS message before calling gld_recv().

The driver should avoid holding mutex or other locks during the call to gld_recv(). In
particular, locks that could be taken by a transmit thread may not be held during a call to
gld_recv(): the interrupt thread that calls gld_recv() may in some cases carry out
processing that includes sending an outgoing packet, resulting in a call to the driver's
gldm_send() routine. If the gldm_send() routine were to try to acquire a mutex being held by
the gldm_intr() routine at the time it calls gld_recv(), this could result in a panic due to
recursive mutex entry.

The interrupt code should increment statistics counters for any errors. This includes failure to
allocate a buffer needed for the received data and any hardware-specific errors such as CRC
errors or framing errors.

gldm_get_stats() gathers statistics from the hardware and/or driver private counters, and
updates the gld_stats(9S) structure pointed to by stats. This routine is called by GLD when it
gets a request for statistics, and provides the mechanism by which GLD acquires device
dependent statistics from the driver before composing its reply to the statistics request. See
gld_stats(9S) and gld(7D) for a description of the defined statistics counters.

gldm_ioctl() implements any device-specific ioctl commands. This element may be specified
as NULL if the driver does not implement any ioctl functions. The driver is responsible for
converting the message block into an ioctl reply message and calling the qreply(9F) function
before returning GLD_SUCCESS. This function should always return GLD_SUCCESS; any errors
the driver may wish to report should be returned via the message passed to qreply(9F). If the
gldm_ioctl() element is specified as NULL, GLD will return a message of type M_IOCNAK with an
error of EINVAL.

**Return Values**

- **gldm_intr()** must return:
  - DDI_INTR_CLAIMED if and only if the device definitely interrupted.
  - DDI_INTR_UNCLAIMED if the device did not interrupt.

The other functions must return:
GLD_SUCCESS  on success, gldm_stop() and gldm_ioctl() should always return this value.

GLD_NORESOURCES  if there are insufficient resources to carry out the request at this time. Only gldm_set_mac_addr(), gldm_set_multicast(), gldm_set_promiscuous(), and gldm_send() may return this value.

GLD_NOLINK  if gldm_send() is called when there is no physical connection to a network or link partner.

GLD_NOTSUPPORTED  if the requested function is not supported. Only gldm_set_mac_addr(), gldm_set_multicast(), and gldm_set_promiscuous() may return this value.

GLD_BADARG  if the function detected an unsuitable argument, for example, a bad multicast address, a bad MAC address, or a bad packet or packet length.

GLD_FAILURE  on hardware failure.

See Also  gld(7D), gld(9F), gld_mac_info(9S), gld_stats(9S), dlpi(7P), attach(9E), ddi_add_intr(9F)

Writing Device Drivers
identify(9E)

**Name**  identify – determine if a driver is associated with a device

**Interface Level**  Solaris DDI specific (Solaris DDI). This entry point is no longer supported. `nulldev(9F)` must be specified in the `dev_ops(9S)` structure.

**See Also**  `nulldev(9F), dev_ops(9S)`

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

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

**Warning**  For Solaris 10 and later versions, drivers must remove the `identify(9E)` implementation to recompile. Otherwise, the compiler generates errors about DDI_IDENTIFIED and DDI_NOT_IDENTIFIED.
Name  ioctl – control a character device

Synopsis  #include <sys/cred.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixioctl(dev_t dev, int cmd, intptr_t arg, int mode,
               cred_t *cred_p, int *rval_p);

Interface Level  Architecture independent level 1 (DDI/DKI). This entry point is optional.

Arguments  

  dev    Device number.
  cmd    Command argument the driver ioctl() routine interprets as the operation to be performed.
  arg    Passes parameters between a user program and the driver. When used with terminals, the argument is the address of a user program structure containing driver or hardware settings. Alternatively, the argument may be a value that has meaning only to the driver. The interpretation of the argument is driver dependent and usually depends on the command type; the kernel does not interpret the argument.
  mode   A bit field that contains:
         ■ Information set when the device was opened. The driver may use it to determine if the device was opened for reading or writing. The driver can make this determination by checking the FREAD or FWRITE flags. See the flag argument description of the open() routine for further values.
         ■ Information on whether the caller is a 32-bit or 64-bit thread.
         ■ In some circumstances address space information about the arg argument. See below.
  cred_p  Pointer to the user credential structure.
  rval_p  Pointer to return value for calling process. The driver may elect to set the value which is valid only if the ioctl() succeeds.

Description  ioctl() provides character-access drivers with an alternate entry point that can be used for almost any operation other than a simple transfer of characters in and out of buffers. Most often, ioctl() is used to control device hardware parameters and establish the protocol used by the driver in processing data.
The kernel determines that this is a character device, and looks up the entry point routines in `cb_ops(9S)`. The kernel then packages the user request and arguments as integers and passes them to the driver’s `ioctl()` routine. The kernel itself does no processing of the passed command, so it is up to the user program and the driver to agree on what the arguments mean.

I/O control commands are used to implement the terminal settings passed from `ttymon(1M)` and `stty(1)`, to format disk devices, to implement a trace driver for debugging, and to clean up character queues. Since the kernel does not interpret the command type that defines the operation, a driver is free to define its own commands.

Drivers that use an `ioctl()` routine typically have a command to “read” the current `ioctl()` settings, and at least one other that sets new settings. Drivers can use the `mode` argument to determine if the device unit was opened for reading or writing, if necessary, by checking the `FREAD` or `FWRITE` setting.

If the third argument, `arg`, is a pointer to a user buffer, the driver can call the `copyin(9F)` and `copyout(9F)` functions to transfer data between kernel and user space.

Other kernel subsystems may need to call into the drivers `ioctl()` routine. Drivers that intend to allow their `ioctl()` routine to be used in this way should publish the `ddi-kernel_ioctl` property on the associated devinfo node(s).

When the `ddi-kernel_ioctl` property is present, the `mode` argument is used to pass address space information about `arg` through to the driver. If the driver expects `arg` to contain a buffer address, and the FKIOCCTL flag is set in `mode`, then the driver should assume that it is being handed a kernel buffer address. Otherwise, `arg` may be the address of a buffer from a user program. The driver can use `ddi_copyin(9F)` and `ddi_copyout(9F)` perform the correct type of copy operation for either kernel or user address spaces. See the example on `ddi_copyout(9F)`.

Drivers have to interact with 32-bit and 64-bit applications. If a device driver shares data structures with the application (for example, through exported kernel memory) and the driver gets recompiled for a 64-bit kernel but the application remains 32-bit, binary layout of any data structures will be incompatible if they contain longs or pointers. The driver needs to know whether there is a model mismatch between the current thread and the kernel and take necessary action. The `mode` argument has additional bits set to determine the C Language Type Model which the current thread expects. `mode` has `FILP32` set if the current thread expects 32-bit (ILP32) semantics, or `FLP64` if the current thread expects 64-bit (LP64) semantics. `mode` is used in combination with `ddi_model_convert_from(9F)` and the `FMODELS` mask to determine whether there is a data model mismatch between the current thread and the device driver (see the example below). 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.

To implement I/O control commands for a driver the following two steps are required:
1. Define the I/O control command names and the associated value in the driver’s header and comment the commands.

2. Code the `ioctl()` routine in the driver that defines the functionality for each I/O control command name that is in the header.

The `ioctl()` routine is coded with instructions on the proper action to take for each command. It is commonly a `switch` statement, with each case definition corresponding to an `ioctl()` name to identify the action that should be taken. However, the command passed to the driver by the user process is an integer value associated with the command name in the header.

**Return Values**

`ioctl()` should return 0 on success, or the appropriate error number. The driver may also set the value returned to the calling process through `rval_p`.

**Examples**

**EXAMPLE 1  ioctl() entry point**

The following is an example of the `ioctl()` entry point and how to support 32-bit and 64-bit applications with the same device driver.

```c
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;

    #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),
```
EXAMPLE 1 ioctl() entry point  (Continued)

    mode);
    break;
}
#else /* _MULTI_DATAMODEL */
    ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
#endif /* _MULTI_DATAMODEL */

    do_ioctl(&pa);
    ...

See Also  stty(1), ttymon(1M), dkio(7I), fbio(7I), termio(7I), open(9E), put(9E), srv(9E),
           copyin(9F), copyout(9F), ddi_copyin(9F), ddi_copyout(9F),
           ddi_model_convert_from(9F), cb_ops(9S)

Writing Device Drivers

Warnings  Non-STREAMS driver ioctl() routines must make sure that user data is copied into or out of
the kernel address space explicitly using copyin(9F), copyout(9F), ddi_copyin(9F), or
ddi_copyout(9F), as appropriate.

It is a severe error to simply dereference pointers to the user address space, even when in user
context.

Failure to use the appropriate copying routines can result in panics under load on some
platforms, and reproducible panics on others.

Notes  STREAMS drivers do not have ioctl() routines. The stream head converts I/O control
commands to M_IOCTL messages, which are handled by the driver's put(9E) or srv(9E)
routine.
Name  ks_snapshot – take a snapshot of kstat data

Synopsis  #include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefix_ks_snapshot(kstat_t *ksp, void *buf, int rw);

Interface Level  Solaris DDI specific (Solaris DDI).

Parameters  ksp  Pointer to a kstat(9S) structure.
            buf  Pointer to a buffer to copy the snapshot into.
            rw  Read/Write flag. Possible values are:
            KSTAT_READ  Copy driver statistics from the driver to the buffer.
            KSTAT_WRITE Copy statistics from the buffer to the driver.

Description  The kstat mechanism allows for an optional ks_snapshot() function to copy kstat data. This is the routine that is called to marshall the kstat data to be copied to user-land. A driver can opt to use a custom snapshot routine rather than the default snapshot routine; to take advantage of this feature, set the ks_snapshot field before calling kstat_install(9F).

The ks_snapshot() function must have the following structure:

    static int
    xx_kstat_snapshot(kstat_t *ksp, void *buf, int rw)
    {
        if (rw == KSTAT_WRITE) {
            /* set the native stats to the values in buf */
            /* return EACCES if you don't support this */
        } else {
            /* copy the kstat-specific data into buf */
        }
        return (0);
    }

In general, the ks_snapshot() routine might need to refer to provider-private data; for example, it might need a pointer to the provider's raw statistics. The ks_private field is available for this purpose. Its use is entirely at the provider's discretion.

No kstat locking should be done inside the ks_update() routine. The caller will already be holding the kstat’s ks_lock (to ensure consistent data) and will prevent the kstat from being removed.
1. `ks_snaptime` must be set (via `gethrtime(9F)`) to timestamp the data.
2. Data gets copied from the `kstat` to the buffer on `KSTAT_READ`, and from the buffer to the `kstat` on `KSTAT_WRITE`.

**Return Values**

- **0** Success
- **EACCES** If `KSTAT_WRITE` is not allowed
- **EIO** For any other error

**Context**

This function is called from user context only.

**Examples**

**EXAMPLE 1** Named kstats with Long Strings (`KSTAT_DATA_STRING`)

```c
static int
xxx_kstat_snapshot(kstat_t *ksp, void *buf, int rw)
{
  if (rw == KSTAT_WRITE) {
    return (EACCES);
  } else {
    kstat_named_t *knp = buf;
    char *end = knp + ksp->ks_ndata;
    uint_t i;

    bcopy(ksp->ks_data, buf,
          sizeof (kstat_named_t) * ksp->ks_ndata);
    /*
     * Now copy the strings to the end of the buffer, and
     * update the pointers appropriately.
     */
    for (i = 0; i < ksp->ks_ndata; i++, knp++)
      if (knp->data_type == KSTAT_DATA_STRING &&
          KSTAT_NAMED_STR_PTR(knp) != NULL) {
        bcopy(KSTAT_NAMED_STR_PTR(knp), end,
              KSTAT_NAMED_STR_BUFLEN(knp));
        KSTAT_NAMED_STR_PTR(knp) = end;
        end += KSTAT_NAMED_STR_BUFLEN(knp);
      }
  }
  return (0);
}
```

**See Also**

`ks_update(9E), kstat_create(9F), kstat_install(9F), kstat(9S)`

**Writing Device Drivers**
Name  
ks_update – dynamically update kstats

Synopsis 
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefix_ks_update(kstat_t *ksp, int rw);

Interface Level  
Solaris DDI specific (Solaris DDI)

Parameters  

ksp  
Pointer to a kstat(9S) structure.

rw  
Read/Write flag. Possible values are

KSTAT_READ  
Update kstat structure statistics from the driver.

KSTAT_WRITE  
Update driver statistics from the kstat structure.

Description  
The kstat mechanism allows for an optional ks_update() function to update kstat data. This is useful for drivers where the underlying device keeps cheap hardware statistics, but extraction is expensive. Instead of constantly keeping the kstat data section up to date, the driver can supply a ks_update() function which updates the kstat’s data section on demand. To take advantage of this feature, set the ks_update field before calling kstat_install(9F).

The ks_update() function must have the following structure:

static int
xx_kstat_update(kstat_t *ksp, int rw)
{
    if (rw == KSTAT_WRITE) {
        /* update the native stats from ksp->ks_data */
        /* return EACCES if you don’t support this */
    } else {
        /* update ksp->ks_data from the native stats */
    }
    return (0);
}

In general, the ks_update() routine may need to refer to provider-private data; for example, it may need a pointer to the provider’s raw statistics. The ks_private field is available for this purpose. Its use is entirely at the provider’s discretion.

No kstat locking should be done inside the ks_update() routine. The caller will already be holding the kstat’s ks_lock (to ensure consistent data) and will prevent the kstat from being removed.


Return Values  ks_update() should return

0        For success.
EACCES   If KSTAT_WRITE is not allowed.
EIO      For any other error.

See Also  kstat_create(9F), kstat_install(9F), kstat(9S)

Writing Device Drivers
### Name
mac, mc_getstat, mc_start, mc_stop, mc_setpromisc, mc_multicst, mc_unicst, mc_tx, mc_ioctl, mc_getcapab, mc_setprop, mc_getprop, mc_propinfo – MAC driver entry points

### Synopsis
```
#include <sys/mac_provider.h>
#include <sys/mac_ether.h>

int prefix_getstat(void *driver_handle, uint_t stat, uint64_t *stat_value);
int prefix_start(void *driver_handle);
void prefix_stop(void *driver_handle);
int prefix_setpromisc(void *driver_handle, boolean_t promisc_mode);
int prefix_multicst(void *driver_handle, boolean_t add, const uint8_t *mcast_addr);
int prefix_unicst(void *driver_handle, const uint8_t *ucast_addr);
mblk_t *prefix_tx(void *driver_handle, mblk_t *mp_chain);
void prefix_ioctl(void *driver_handle, queue_t *q, mblk_t *mp);
boolean_t prefix_getcapab(void *driver_handle, mac_capab_t cap, void *cap_data);
int prefix_setprop(void *driver_handle, const char *prop_name, mac_prop_id_t prop_id, uint_t prop_val_size, const void *prop_val);
int prefix_getprop(void *driver_handle, const char *prop_name, mac_prop_id_t prop_id, uint_t prop_val_size, void *prop_val);
void prefix_propinfo(void *driver_handle, const char *prop_name, mac_prop_id_t prop_id, mac_prop_info_handle_t prop_handle);
```

### Parameters
- **driver_handle**: pointer to the driver-private handle that was specified by the device driver through the \textit{m_driver} field of the \texttt{mac_register(9S)} structure during registration.
- **stat**: statistic being queried
- **stat\_val**: value of statistic being queried
- **promisc\_mode**: promiscuous mode to be set
- **add**: whether to add or delete the multicast address
- **mcast\_addr**: value of the multicast address to add or remove
- **ucast\_addr**: value of the unicast address to set
- **q**: STREAMS queue for ioctl operation
- **mp**: message block for ioctl operation
**Interface Level**  
Solaris architecture specific (Solaris DDI)

**Description**  
The entry points described below are implemented by a MAC device driver and passed to the MAC layer through the `mac_register` structure as part of the registration process using `mac_register(9F)`.

The `mc_getstat()` entry point returns through the 64 bit unsigned integer pointed to by `stat_value` the value of the statistic specified by the `stat` argument. Supported statistics are listed in the Statistics section below. The device driver `mc_getstat()` entry point should return 0 if the statistics is successfully passed back to the caller, or `ENOTSUP` if the statistic is not supported by the device driver.

The `mc_start()` entry point starts the device driver instance specified by `driver_handle`.

The `mc_stop()` entry point stops the device driver instance specified by `driver_handle`. The MAC layer will invoke the stop entry point before the device is detached.

The `mc_setpromisc()` entry point is used to change the promiscuous mode of the device driver instance specified by `driver_handle`. Promiscuous mode should be turned on if the `promisc_mode` is set to `B_TRUE` and off if the `promisc_mode` is set to `B_FALSE`.

The `mc_multicast()` entry point adds or remove the multicast address pointed to by `mcast_addr` to or from the device instance specified by `driver_handle`.

The `mc_unicast()` entry point sets the primary unicast address of the device instance specified by `driver_handle` to the value specified by `ucast_addr`. The device must start passing back through `mac_rx()` the packets with a destination MAC address which matches the new unicast address.

The `mc_tx()` entry point is used to transmit message blocks, chained using the `b_next` pointer, on the device driver instance specified by `driver_instance`. If all the message blocks could be

---

### Interface Level: Solaris architecture specific (Solaris DDI)

#### Description

The entry points described below are implemented by a MAC device driver and passed to the MAC layer through the `mac_register` structure as part of the registration process using `mac_register(9F)`.

The `mc_getstat()` entry point returns through the 64 bit unsigned integer pointed to by `stat_value` the value of the statistic specified by the `stat` argument. Supported statistics are listed in the Statistics section below. The device driver `mc_getstat()` entry point should return 0 if the statistics is successfully passed back to the caller, or `ENOTSUP` if the statistic is not supported by the device driver.

The `mc_start()` entry point starts the device driver instance specified by `driver_handle`.

The `mc_stop()` entry point stops the device driver instance specified by `driver_handle`. The MAC layer will invoke the stop entry point before the device is detached.

The `mc_setpromisc()` entry point is used to change the promiscuous mode of the device driver instance specified by `driver_handle`. Promiscuous mode should be turned on if the `promisc_mode` is set to `B_TRUE` and off if the `promisc_mode` is set to `B_FALSE`.

The `mc_multicast()` entry point adds or remove the multicast address pointed to by `mcast_addr` to or from the device instance specified by `driver_handle`.

The `mc_unicast()` entry point sets the primary unicast address of the device instance specified by `driver_handle` to the value specified by `ucast_addr`. The device must start passing back through `mac_rx()` the packets with a destination MAC address which matches the new unicast address.

The `mc_tx()` entry point is used to transmit message blocks, chained using the `b_next` pointer, on the device driver instance specified by `driver_instance`. If all the message blocks could be
submitted to the hardware for processing, the entry point returns NULL. If the hardware resources were exhausted, the entry point returns a chain containing the message blocks which could not be sent. In that case, the driver is responsible to invoke the `mac_tx_update(9F)` entry point once more hardware transmit resources are available to resume transmission. The driver is responsible to free the message blocks once the packets have been consumed by the hardware.

The `mc_ioctl()` entry point is a generic facility which can be used to pass arbitrary ioctl to a driver from STREAMs clients. This facility is intended to be used only for debugging purposes only. The STREAMs `M_IOCTL` messages can be generated by a user-space application and passed done to the device `libdpif(3LIB)`.

The `mc_getcapab()` entry point queries a specific capability from the driver. The cap argument specifies the type of capability being queried, and `cap_data` is used by the driver to return the capability data to the framework, if any. It the driver does not support the capability specified by the framework, it must return `B_FALSE`, otherwise the driver must return `B_TRUE`. The following capabilities are supported:

- **MAC_CAPAB_HCKSUM**
  - The `cap_data` argument points to a `uint32_t` location. The driver must return in `cap_data` a combination of one of the following flags:
    - `HCKSUM_INET_PARTIAL`
      - Partial 1's complement checksum ability.
    - `HCKSUM_INET_FULL_V4`
      - Full 1's complement checksum ability for IPv4 packets.
    - `HCKSUM_INET_FULL_V6`
      - Full 1's complement checksum ability for IPv6 packets.
    - `HCKSUM_IPHDRCKSUM`
      - IPv4 Header checksum offload capability.

  These flags indicate the level of hardware checksum offload that the driver is capable of performing for outbound packets.

  When hardware checksumming is enabled, the driver must use the `mac_hcksum_get(9F)` function to retrieve the per-packet hardware checksumming metadata.

- **MAC_CAPAB_LSO**
  - The `cap_data` argument points to a `mac_capab_lso_t` structure which describes the LSO capabilities of the driver, and is described in details in `mac_capab_lso(9S)`.

  The `mc_setprop()` and `mc_getprop()` entry points set and get, respectively, the value of a property for the device driver instance specified by `driver_handle`. The property is specified by the `prop_id` argument, and is one of the properties identifier listed in section `Properties` below. The value of the property is stored in a buffer at `prop_val`, and the size of that buffer is...
specified by \textit{prop_val_size}. The MAC layer ensures that the buffer is large enough to store the property specified by \textit{prop_id}. The type of each property is listed in the \texttt{Properties} section below.

The \texttt{mc\_prop\_info()} entry point returns immutable attributes of a property for the device driver instance specified by \textit{driver\_handle}. The property is specified by the \textit{prop\_id} argument, and is one of the properties identifier listed in section \texttt{Properties} below. The entry point invokes the \texttt{mac\_prop\_info\_set\_perm()}, \texttt{mac\_prop\_info\_set\_default()}, or \texttt{mac\_prop\_info\_set\_range()} functions to associate specific attributes of the property being queried. The opaque property handle passed to the \texttt{mc\_prop\_info()} entry point must be passed as-is to these routines.

In addition to the properties listed in the \texttt{Properties} section below, drivers can also expose driver-private properties. These properties are identified by property names strings. Private property names always start with an underscore \texttt{(_)} character and must be no longer than 256 characters, including a null-terminating character. Driver-private properties supported by a device driver are specified by the \texttt{m\_priv\_props} field of the \texttt{mac\_register} data structure. During a call to \texttt{mc\_setprop()}, \texttt{mc\_getprop()}, or \texttt{mc\_prop\_info()}, a private property is specified by a property id of \texttt{MAC\_PROP\_PRIVATE}, and the driver property name is passed through the \texttt{prop\_name} argument. Private property values are always specified by a string. The driver is responsible to encode and parse private properties value strings.

\textbf{Return Values} The \texttt{mc\_getstat()} entry point returns 0 on success, or \texttt{ENOTSUP} if the specific statistic is not supported by the device driver.

The \texttt{mc\_start()}, \texttt{mc\_setpromisc()}, \texttt{mc\_multicst()}, and \texttt{mc\_unicst()} entry points return 0 on success and one of the error values specified by \textit{Intro(2)} on failure.

The \texttt{mc\_getcapab()} entry point returns \texttt{B\_TRUE} if the capability is supported by the device driver, \texttt{B\_FALSE} otherwise.

The \texttt{mc\_tx()} entry point returns \texttt{NULL} if all packets could be posted on the hardware to be sent. The entry point returns a chain of unsent message blocks if the transmit resources were exhausted.

The \texttt{mc\_setprop()} and \texttt{mc\_getprop()} entry points return 0 on success, \texttt{ENOTSUP} if the property is not supported by the device driver, or an error value specified by \textit{Intro(2)} for other failures.

\textbf{Context} The \texttt{mc\_tx()} entry point can be called from interrupt context. The other entry points can be called from user or kernel context.

\textbf{Statistics} The stat argument value of the \texttt{mc\_getstat()} entry point is used by the framework to specify the specific statistic being queried. The following statistics are supported by all media types:

- MIB-II stats (RFC 1213 and RFC 1573):
MAC_STAT_IFSPEED
MAC_STAT_MULTIRCV
MAC_STAT_BRDCSTRCV
MAC_STAT_MULTIXMT
MAC_STAT_BRDCSTXMT
MAC_STAT_NORCVBUF
MAC_STAT_IERRORS
MAC_STAT_UNKNOWNS
MAC_STAT_NOXMTBUF
MAC_STAT_OERRORS
MAC_STAT_COLLISIONS
MAC_STAT_RBYTES
MAC_STAT_IPACKETS
MAC_STAT_OBYTES
MAC_STAT_OPACKETS
MAC_STAT_UNDERFLOWS
MAC_STAT_OVERFLOWS

The following statistics are specific to Ethernet device drivers:

RFC 1643 stats:

ETHER_STAT_ALIGN_ERRORS
ETHER_STAT_FCS_ERRORS
ETHER_STAT_FIRST_COLLISIONS
ETHER_STAT_MULTI_COLLISIONS
ETHER_STAT_SQE_ERRORS
ETHER_STAT_DEFER_XMTS
ETHER_STAT_TX_LATE_COLLISIONS
ETHER_STAT_EX_COLLISIONS
ETHER_STAT_MACXMT_ERRORS
ETHER_STAT_CARRIER_ERRORS
ETHER_STAT_TOOLONG_ERRORS
ETHER_STAT_MACRCV_ERRORS

MII/GMII stats:

ETHER_STAT_XCVR_ADDR
ETHER_STAT_XCVR_ID
ETHER_STAT_XCVR_INETUSE
ETHER_STAT_CAP_1000FDX
ETHER_STAT_CAP_1000HDX
ETHER_STAT_CAP_100FDX
ETHER_STAT_CAP_100HDX
ETHER_STAT_CAP_10FDX
ETHER_STAT_CAP_10HDX
ETHER_STAT_CAP_ASMPAUSE
ETHER_STAT_CAP_PAUSE
ETHER_STAT_CAP_AUTONEG
ETHER_STAT_ADV_CAP_1000FDX
<table>
<thead>
<tr>
<th>Property</th>
<th>PropertyType</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC_PROP_DUPLEX</td>
<td>link_duplex_t</td>
</tr>
<tr>
<td>MAC_PROP_SPEED</td>
<td>uint64_t</td>
</tr>
<tr>
<td>MAC_PROP_STATUS</td>
<td>link_state_t</td>
</tr>
<tr>
<td>MAC_PROP_AUTONEG</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_MGU</td>
<td>uint32_t</td>
</tr>
<tr>
<td>MAC_PROP_FLOWCTRL</td>
<td>link_flowctrl_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_10GFDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_10GFDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_1000FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_1000FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_1000HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_1000HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_100FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_100FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_LINK_AUTONEG</td>
<td></td>
</tr>
<tr>
<td>MAC_PROP_LINK_DUPLEX</td>
<td></td>
</tr>
</tbody>
</table>
## Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Property Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC_PROP_ADV_100HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_100HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_10FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_10FDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_ADV_10HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_EN_10HDX_CAP</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MAC_PROP_PRIVATE</td>
<td>char[]</td>
</tr>
</tbody>
</table>

### 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>Availability</td>
<td>SUNWhea</td>
</tr>
<tr>
<td>Interface Stability</td>
<td>Committed</td>
</tr>
</tbody>
</table>

### See Also

libdlpi(3LIB), attributes(5), mac_hcksum_get(9F), mac_prop_info_set_perm(9F), mac_register(9F), mac_tx_update(9F), mac_capab_lso(9S), mac_register(9S)
## mmap(9E)

### Name
mmap – check virtual mapping for memory mapped device

### Synopsis
```
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/mman.h>
#include <sys/ddi.h>

int prefixmmap(dev_t dev, off_t off, int prot);
```

### Interface Level
This interface is obsolete. devmap(9E) should be used instead.

### Parameters
- **dev**: Device whose memory is to be mapped.
- **off**: Offset within device memory at which mapping begins.
- **prot**: A bit field that specifies the protections this page of memory will receive. Possible settings are:
  - PROT_READ: Read access will be granted.
  - PROT_WRITE: Write access will be granted.
  - PROT_EXEC: Execute access will be granted.
  - PROT_USER: User-level access will be granted.
  - PROT_ALL: All access will be granted.

### 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 mmap() entry point is a required entry point for character drivers supporting memory-mapped devices. A memory mapped device has memory that can be mapped into a process’s address space. The mmap(2) system call, when applied to a character special file, allows this device memory to be mapped into user space for direct access by the user application.

The mmap() entry point is called as a result of an mmap(2) system call, and also as a result of a page fault. mmap() is called to translate the offset `off` in device memory to the corresponding physical page frame number.

The mmap() entry point checks if the offset `off` is within the range of pages exported by the device. For example, a device that has 512 bytes of memory that can be mapped into user space should not support offsets greater than 512. If the offset does not exist, then -1 is returned. If the offset does exist, mmap() returns the value returned by hat_getkpfnum(9F) for the physical page in device memory containing the offset `off`.

---

82  man pages section 9: DDI and DKI Driver Entry Points • Last Revised 27 Sep 2002

hat_getkpfnum(9F) accepts a kernel virtual address as an argument. A kernel virtual address can be obtained by calling ddi_regs_map_setup(9F) in the driver's attach(9E) routine. The corresponding ddi_regs_map_free(9F) call can be made in the driver's detach(9E) routine. Refer to the example below mmap Entry Point for more information.

mmap() should only be supported for memory-mapped devices. See segmap(9E) for further information on memory-mapped device drivers.

If a device driver shares data structures with the application, for example through exported kernel memory, and the driver gets recompiled for a 64-bit kernel but the application remains 32-bit, the binary layout of any data structures will be incompatible if they contain longs or pointers. The driver needs to know whether there is a model mismatch between the current thread and the kernel and take necessary action. ddi_mmap_get_model(9F) can be use to get the C Language Type Model which the current thread expects. In combination with ddi_model_convert_from(9F) the driver can 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. See ddi_mmap_get_model(9F) for an example.

Return Values
If the protection and offset are valid for the device, the driver should return the value returned by hat_getkpfnum(9F), for the page at offset off in the device's memory. If not, -1 should be returned.

Examples
EXAMPLE 1 mmap() Entry Point

The following is an example of the mmap() entry point. If offset off is valid, hat_getkpfnum(9F) is called to obtain the page frame number corresponding to this offset in the device's memory. In this example, xsp->regp->csr is a kernel virtual address which maps to device memory. ddi_regs_map_setup(9F) can be used to obtain this address. For example, ddi_regs_map_setup(9F) can be called in the driver's attach(9E) routine. The resulting kernel virtual address is stored in the xxstate structure, which is accessible from the driver's mmap() entry point. See ddi_soft_state(9F). The corresponding ddi_regs_map_free(9F) call can be made in the driver's detach(9E) routine.

```c
struct reg {
    uint8_t csr;
    uint8_t data;
};
struct xxstate {
    . .
    struct reg *regp
    . .
};
struct xxstate *xsp;
```
EXAMPLE 1  mmap() Entry Point  (Continued)

static int
xmmmap(dev_t dev, off_t off, int prot)
{
    int     instance;
    struct xxstate *xsp;

    /* No write access */
    if (prot & PROT_WRITE)
        return (-1);

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (-1);

    /* check for a valid offset */
    if ( off is invalid )
        return (-1);
    return (hat_getkpfnum (xsp->regp->csr + off));
}

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

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability Level</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>

See Also  mmap(2), attributes(5), attach(9E), detach(9E), devmap(9E), segmap(9E), ddi_btop(9F),
ddi_get_soft_state(9F), ddi_mmap_get_model(9F), ddi_model_convert_from(9F),
ddi_regs_map_free(9F), ddi_regs_map_setup(9F), ddi_soft_state(9F),
devmap_setup(9F), getminor(9F), hat_getkpfnum(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.

One alternative is to create a mapping for only the first page of device memory in attach(9E).
If the device memory is contiguous, a kernel page frame number may be obtained by calling
hat_getkpfnum(9F) with the kernel virtual address of the first page of device memory and
adding the desired page offset to the result. The page offset may be obtained by converting the
byte offset off to pages. See ddi_btop(9F).
Another alternative is to call `ddi_regs_map_setup(9F)` and `ddi_regs_map_free(9F)` in `mmap()`.

These function calls would bracket the call to `hat_getkpfnum(9F)`.

However, note that the above alternatives may not work in all cases. The existence of intermediate nexus devices with memory management unit translation resources that are not locked down may cause unexpected and undefined behavior.
open(9E)

**Name**  open – gain access to a device

**Synopsis**

Block and Character

```c
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixopen(dev_t *devp, int flag, int otyp,
               cred_t *cred_p);
```

STREAMS

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

int prefixopen(queue_t *q, dev_t *devp, int oflag, int sflag,
               cred_t *cred_p);
```

**Interface Level**  Architecture independent level 1 (DDI/DKI). This entry point is required, but it can be `nulldev(9F)`

**Parameters**

Block and Character

- **`devp`**  Pointer to a device number.
- **`flag`**  A bit field passed from the user program `open(2)` system call that instructs the driver on how to open the file. Valid settings are:
  - **FEXCL**  Open the device with exclusive access; fail all other attempts to open the device.
  - **FNDELAY**  Open the device and return immediately. Do not block the open even if something is wrong.
  - **FREAD**  Open the device with read-only permission. If ORed with `FWRITE`, allow both read and write access.
  - **FWRITE**  Open a device with write-only permission. If ORed with `FREAD`, allow both read and write access.
- **`otyp`**  Parameter supplied for driver to determine how many times a device was opened and for what reasons. For `OTYP_BLK` and `OTYP_CHR`, the `open()` function can be
called many times, but the close(9E) function is called only when the last reference to a device is removed. If the device is accessed through file descriptors, it is done by a call to close(2) or exit(2). If the device is accessed through memory mapping, it is done by a call to munmap(2) or exit(2). For OTYP_LYR, there is exactly one close(9E) for each open() operation that is called. This permits software drivers to exist above hardware drivers and removes any ambiguity from the hardware driver regarding how a device is used.

- **OTYP_BLK**: Open occurred through block interface for the device.
- **OTYP_CHR**: Open occurred through the raw/character interface for the device.
- **OTYP_LYR**: Open a layered process. This flag is used when one driver calls another driver's open() or close(9E) function. The calling driver ensures that there is one-layered close for each layered open. This flag applies to both block and character devices.

- **cred_p**: Pointer to the user credential structure.
- **q**: A pointer to the read queue.
- **devp**: Pointer to a device number. For STREAMS modules, devp always points to the device number associated with the driver at the end (tail) of the stream.
- **oflag**: Valid oflag values are FEXCL, FNDELAY, FREAD, and FWRITE — the same as those listed above for flag. For STREAMS modules, oflag is always set to 0.
- **sflag**: Valid values are as follows:
  - **CLONEOPEN**: Indicates that the open() function is called through the clone driver. The driver should return a unique device number.
  - **MODOPEN**: Modules should be called with sflag set to this value. Modules should return an error if they are called with sflag set to a different value. Drivers should return an error if they are called with sflag set to this value.
  - **0**: Indicates a driver is opened directly, without calling the clone driver.

- **cred_p**: Pointer to the user credential structure.

**Description**: The driver’s open() function is called by the kernel during an open(2) or a mount(2) on the special file for the device. A device can be opened simultaneously by multiple processes and the open() driver operation is called for each open. Note that a device is referenced once its associated open(9E) function is entered, and thus open(9E) operations which have not yet completed will prevent close(9E) from being called. The function should verify that the minor number component of *devp is valid, that the type of access requested by oftyp and flag is appropriate for the device, and, if required, check permissions using the user credentials pointed to by cred_p.
The kernel provides open() close() exclusion guarantees to the driver at *devp, otyp granularity. This delays new open() calls to the driver while a last-reference close() call is executing. If the driver has indicated that an EINTR returns safe via the D_OPEN_RETURNS_EINTR cb_ops(9S) cb_flag, a delayed open() may be interrupted by a signal that results in an EINTR return.

Last-reference accounting and open() close() exclusion typically simplify driver writing. In some cases, however, they might be an impediment for certain types of drivers. To overcome any impediment, the driver can change minor numbers in open(9E), as described below, or implement multiple minor nodes for the same device. Both techniques give the driver control over when close() calls occur and whether additional open() calls will be delayed while close() is executing.

The open() function is passed a pointer to a device number so that the driver can change the minor number. This allows drivers to dynamically create minor instances of the device. An example of this might be a pseudo-terminal driver that creates a new pseudo-terminal whenever it is opened. A driver that chooses the minor number dynamically, normally creates only one minor device node in attach(9E) with ddi_create_minor_node(9F). It then changes the minor number component of *devp using makedevice(9F) and getmajor(9F). The driver needs to keep track of available minor numbers internally. A driver that dynamically creates minor numbers might want to avoid returning the original minor number since returning the original minor will result in postponed dynamic opens when original minor close() call occurs.

*devp = makedevice(getmajor(*devp), new_minor);

Return Values The open() function should return 0 for success, or the appropriate error number.

See Also close(2), exit(2), mmap(2), mount(2), munmap(2), open(2), Intro(9E), attach(9E), close(9E), ddi_create_minor_node(9F), getmajor(9F), getminor(9F), makedevice(9F), nulldev(9F), cb_ops(9S)

Writing Device Drivers

STREAMS Programming Guide

Warnings Do not attempt to change the major number.

When a driver modifies the device number passed in, it must not change the major number portion of the device number. Unless CLONEOPEN is specified, the modified device number must map to the same driver instance indicated by the driver’s getinfo(9e) implementation. In other words, cloning across different drivers is not supported. Cloning across different instances of the same driver in only permitted if the driver specified in CLONE_DEV in ddi_create_minor_node(9F) is not supported.
The `power()` function is the device-specific Power Management entry point. This function is called when the system wants the driver to set the power level of `component` to `level`.

The `level` argument is the driver-defined power level to which the component needs to be set. Except for power level 0, which is interpreted by the framework to mean "powered off," the interpretation of `level` is entirely up to the driver.

The `component` argument is the component of the device to be power-managed. The interpretation of `component` is entirely up to the driver.

When a requested power transition would cause the device to lose state, the driver must save the state of the device in memory. When a requested power transition requires state to be restored, the driver must restore that state.

If a requested power transition for one component requires another component to change power state before it can be completed, the driver must call `pm_raise_power()` to get the other component changed, and the `power()` entry point must support being re-entered.

If the system requests an inappropriate power transition for the device (for example, a request to power down a device which has just become busy), then the power level should not be changed and power should return `DDI_FAILURE`.

The `power()` function returns:

- `DDI_SUCCESS` Successfully set the power to the requested `level`.
- `DDI_FAILURE` Failed to set the power to the requested `level`.

The `power()` function is called from user or kernel context only.
Attributes  See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

See Also  attach(9E), detach(9E), pm_busy_component(9F), pm_idle_component(9F), pm_raise_power(9F), cb_ops(9S)

Writing Device Drivers

Using Power Management
### Name
print – display a driver message on system console

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

int prefixprint(dev_t dev, char *str);
```

### Interface Level
Architecture independent level 1 (DDI/DKI). This entry point is required for block devices.

### Parameters
- **dev**: Device number.
- **str**: Pointer to a character string describing the problem.

### Description
The `print()` routine is called by the kernel when it has detected an exceptional condition (such as out of space) in the device. To display the message on the console, the driver should use the `cmn_err(9F)` kernel function. The driver should print the message along with any driver specific information.

### Return Values
The `print()` routine should return 0 for success, or the appropriate error number. The `print()` routine can fail if the driver implemented a non-standard `print()` routine that attempted to perform error logging, but was unable to complete the logging for whatever reason.

### See Also
- `cmn_err(9F)`
- *Writing Device Drivers*
probe

probe -- determine if a non-self-identifying device is present

Synopsis

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

static int prefixprobe(dev_info_t *dip);
```

Interface Level

Solaris DDI specific (Solaris DDI). This entry point is required for non-self-identifying devices. You must write it for such devices. For self-identifying devices, `nulldev(9F)` should be specified in the `dev_ops(9S)` structure if a probe routine is not necessary.

Arguments

dip Pointer to the device's `dev_info` structure.

Description

`probe()` determines whether the device corresponding to `dip` actually exists and is a valid device for this driver. `probe()` is called after `identify(9E)` and before `attach(9E)` for a given `dip`. For example, the `probe()` routine can map the device registers using `ddi_map_regs(9F)` then attempt to access the hardware using `ddi.peek(9F)` or `ddi.poke(9F)` and determine if the device exists. Then the device registers should be unmapped using `ddi_unmap_regs(9F)`.

To probe a device that was left powered off after the last `detach()`, it might be necessary to power it up. If so, the driver must power up the device by accessing device registers directly. `pm_raise_power(9F)` will not be available until `attach(9E)`. The framework ensures that the ancestors of the node being probed and all relevant platform-specific power management hardware is at full power at the time that `probe()` is called.

`probe()` should only probe the device. It should not change any software state and should not create any software state. Device initialization should be done in `attach(9E)`.

For a self-identifying device, this entry point is not necessary. However, if a device exists in both self-identifying and non-self-identifying forms, a `probe()` routine can be provided to simplify the driver. `ddi_dev_is_sid(9F)` can then be used to determine whether `probe()` needs to do any work. See `ddi_dev_is_sid(9F)` for an example.

Return Values

- `DDI_PROBE_SUCCESS` If the probe was successful.
- `DDI_PROBE_FAILURE` If the probe failed.
- `DDI_PROBE_DONTCARE` If the probe was unsuccessful, yet `attach(9E)` should still be called.
- `DDI_PROBE_PARTIAL` If the instance is not present now, but may be present in the future.

See Also

`attach(9E), identify(9E), ddi_dev_is_sid(9F), ddi_map_regs(9F), ddi.peek(9F), ddi.poke(9F), nulldev(9F), dev_ops(9S)`

Writing Device Drivers
### Name
prop_op – report driver property information

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

int prefixprop_op(dev_t dev, dev_info_t *dip,
                  ddi_prop_op_t prop_op, int flags, char *name,
                  caddr_t valuep, int *lengthp);
```

### Interface Level
Solaris DDI specific (Solaris DDI). This entry point is required, but it can be `ddi_prop_op(9F)`.

### Arguments
- **dev**  
  Device number associated with this device.
- **dip**  
  A pointer to the device information structure for this device.
- **prop_op**  
  Property operator. Valid operators are:
  - `PROP_LEN`  
    Get property length only. (`valuep` unaffected).
  - `PROP_LEN_AND_VAL_BUF`  
    Get length and value into caller’s buffer. (`valuep` used as input).
  - `PROP_LEN_AND_VAL_ALLOC`  
    Get length and value into allocated buffer. (`valuep` returned as pointer to pointer to allocated buffer).
- **flags**  
  The only possible flag value is:
  - `DDI_PROP_DONTPASS`  
    Do not pass request to parent if property not found.
- **name**  
  Pointer to name of property to be interrogated.
- **valuep**  
  If `prop_op` is `PROP_LEN_AND_VAL_BUF`, this should be a pointer to the user’s 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 `lengthp` should point to an `int` that contains the length of caller’s buffer, before calling `prop_op()`.

### Description
`prop_op()` is an entry point which reports the values of certain properties of the driver or device to the system. Each driver must have a `prefix prop_op` entry point, but most drivers that do not need to create or manage their own properties can use `ddi_prop_op()` for this entry point. Then the driver can use `ddi_prop_update(9F)` to create properties for its device.

### Return Values
- **DDI_PROP_SUCCESS**  
  Property found and returned.
- **DDI_PROP_NOT_FOUND**  
  Property not found.
DDI_PROP_UNDEFINED  Prop explicitly undefined.
DDI_PROP_NO_MEMORY  Property found, but unable to allocate memory. lengthp has
                     the correct property length.
DDI_PROP_BUF_TOO_SMALL  Property found, but the supplied buffer is too small. lengthp
                        has the correct property length.

Examples  EXAMPLE 1  Using prop_op() to Report Property Information

In the following example, prop_op() intercepts requests for the temperature property. The
driver tracks changes to temperature using a variable in the state structure in order to avoid
frequent calls to ddi_prop_update(9F). The temperature property is only updated when a
request is made for this property. It then uses the system routine ddi_prop_op(9F) to process
the property request. If the property request is not specific to a device, the driver does not
intercept the request. This is indicated when the value of the dev parameter is equal to
DDI_DEV_T_ANY.

```c
int temperature; /* current device temperature */
...
static int
xxprop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
int flags, char *name, caddr_t valuep, int *lengthp)
{
    int instance;
    struct xxstate *xsp;
    if (dev == DDI_DEV_T_ANY)
        goto skip;
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (DDI_PROP_NOT_FOUND);
    if (strcmp(name, "temperature") == 0) {
        ddi_prop_update_int(dev, dip, "temperature", temperature);
    }
    /* other cases... */
skip:
    return (ddi_prop_op(dev, dip, prop_op, flags,
            name, valuep, lengthp));
}
```

See Also  Intro(9E), ddi_prop_op(9F), ddi_prop_update(9F)

Writing Device Drivers
put – receive messages from the preceding queue

**Synopsis**

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

```c
int prefixrput(queue_t *q, mblk_t *mp /* read side */
int prefixwput(queue_t *q, mblk_t *mp /* write side */
```

**Interface Level**

Architecture independent level 1 (DDI/DKI). This entry point is required for STREAMS.

**Arguments**

- `q` Pointer to the `queue(9S)` structure.
- `mp` Pointer to the message block.

**Description**

The primary task of the `put()` routine is to coordinate the passing of messages from one queue to the next in a stream. The `put()` routine is called by the preceding stream component (stream module, driver, or stream head). `put()` routines are designated “write” or “read” depending on the direction of message flow.

With few exceptions, a streams module or driver must have a `put()` routine. One exception is the read side of a driver, which does not need a `put()` routine because there is no component downstream to call it. The `put()` routine is always called before the component's corresponding `srv(9E)` (service) routine, and so `put()` should be used for the immediate processing of messages.

A `put()` routine must do at least one of the following when it receives a message:

- pass the message to the next component on the stream by calling the `putnext(9F)` function;
- process the message, if immediate processing is required (for example, to handle high priority messages); or
- enqueue the message (with the `putq(9F)` function) for deferred processing by the service `srv(9E)` routine.

Typically, a `put()` routine will switch on message type, which is contained in the `db_type` member of the `datab` structure pointed to by `mp`. The action taken by the `put()` routine depends on the message type. For example, a `put()` routine might process high priority messages, enqueue normal messages, and handle an unrecognized `M_IOCTL` message by changing its type to `M_IOCNAK` (negative acknowledgement) and sending it back to the stream head using the `qreply(9F)` function.
The `putq(9F)` function can be used as a module's `put()` routine when no special processing is required and all messages are to be enqueued for the `srv(9E)` routine.

**Return Values**  Ignored.

**Context**  `put()` routines do not have user context.

**See Also**  `srv(9E), putctl(9F), putctl1(9F), putnext(9F), putnextctl(9F), putnextctl1(9F), putq(9F), qreply(9F), queue(9S), streamtab(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*
include <sys/types.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/uio.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixread(dev_t dev, struct uio *uio_p, cred_t *cred_p);

**Interface Level**
Architecture independent level 1 (DDI/DKI). This entry point is optional.

**Parameters**
- **dev** Device number.
- **uio_p** Pointer to the uio(9S) structure that describes where the data is to be stored in user space.
- **cred_p** Pointer to the user credential structure for the I/O transaction.

**Description**
The driver read() routine is called indirectly through cb_ops(9S) by the read(2) system call. The read() routine should check the validity of the minor number component of dev and the user credential structure pointed to by cred_p (if pertinent). The read() routine should supervise the data transfer into the user space described by the uio(9S) structure.

**Return Values**
The read() routine should return 0 for success, or the appropriate error number.

**Examples**
**EXAMPLE 1** read() routine using physio()

The following is an example of a read() routine using physio(9F) to perform reads from a non-seekable device:

```c
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int rval;
    offset_t off;
    int instance;
    xx_t xx;

    instance = getminor(dev);
    xx = ddi_get_soft_state(xxstate, instance);
    if (xx == NULL)
        return (ENXIO);
    off = uiop->uio_loffset;
    rval = physio(xxstrategy, NULL, dev, B_READ,
```
EXAMPLE 1  read() routine using physio()  (Continued)

        xxmin, uiop);
        uiop->uiop_loffset = off;
        return (rval);
    }

See Also  read(2), write(9E), physio(9F), cb_ops(9S), uio(9S)

Writing Device Drivers
segmap – map device memory into user space

Synopsis

#include <sys/types.h>
#include <sys/mman.h>
#include <sys/param.h>
#include <sys/vm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixsegmap(dev_t dev, off_t off, struct asp *asp, caddr_t *addrp,
                  off_t len, unsigned int prot, unsigned int maxprot, unsigned int flags,
                  cred_t *cred_p);

Interface Level

Architecture independent level 2 (DKI only).

Arguments

dev Device whose memory is to be mapped.
off Offset within device memory at which mapping begins.
asp Pointer to the address space into which the device memory should be mapped.
addrp Pointer to the address in the 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. Possible settings are:
PROT_READ Read access is desired.
PROT_WRITE Write access is desired.
PROT_EXEC Execute access is desired.
PROT_USER User-level access is desired (the mapping is being done as a result of a mmap(2) system call).
PROT_ALL All access is desired.
maxprot Maximum protection flag possible for attempted mapping; the PROT_WRITE bit may be masked out if the user opened the special file read-only.
flags Flags indicating type of mapping. Possible values are (other bits may be set):
MAP_SHARED Changes should be shared.
MAP_PRIVATE Changes are private.
cred_p Pointer to the user credentials structure.
The segmap() entrypoint is an optional routine for character drivers that support memory mapping. The mmap(2) system call, when applied to a character special file, allows device memory to be mapped into user space for direct access by the user application.

Typically, a character driver that needs to support the mmap(2) system call supplies either an devmap(9E) entry point, or both an devmap(9E) and a segmap() entry point routine (see the devmap(9E) reference page). If no segmap() entry point is provided for the driver, devmap_setup(9F) is used as a default.

A driver for a memory-mapped device would provide a segmap() entry point if:

- needs to maintain a separate context for each user mapping. See devmap_setup(9F) for details.
- needs to assign device access attributes to the user mapping.

The responsibilities of a segmap() entry point are:

- Verify that the range, defined by offset and len, to be mapped is valid for the device. Typically, this task is performed by calling the devmap(9E) entry point. Note that if you are using ddi_devmap_segmap(9F) or devmap_setup(9F) to set up the mapping, it will call your devmap(9E) entry point for you to validate the range to be mapped.
- Assign device access attributes to the mapping. See ddi_devmap_segmap(9F), and ddi_device_acc_attr(9S) for details.
- Set up device contexts for the user mapping if your device requires context switching. See devmap_setup(9F) for details.
- Perform the mapping with ddi_devmap_segmap(9F), or devmap_setup(9F) and return the status if it fails.

The segmap() routine should return 0 if the driver is successful in performing the memory map of its device address space into the specified address space.

The segmap() must return an error number on failure. For example, valid error numbers would be ENXIO if the offset/length pair specified exceeds the limits of the device memory, or EINVAL if the driver detects an invalid type of mapping attempted.

If one of the mapping routines ddi_devmap_segmap() or devmap_setup() fails, you must return the error number returned by the respective routine.

See Also mmap(2), devmap(9E), devmap_setup(9F), ddi_devmap_segmap(9F), ddi_device_acc_attr(9S)

Writing Device Drivers
# srv (9E)

**Name**

srv – service queued messages

**Synopsis**

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

```
intprefixrsrv(queue_t *q/* read side */
intprefixwsrv(queue_t *q/* write side */
```

**Interface Level**

Architecture independent level 1 (DDI/DKI). This entry point is required for STREAMS.

**Arguments**

- `q` Pointer to the `queue(9S)` structure.

**Description**

The optional service `srv()` routine may be included in a STREAMS module or driver for many possible reasons, including:

- to provide greater control over the flow of messages in a stream;
- to make it possible to defer the processing of some messages to avoid depleting system resources;
- to combine small messages into larger ones, or break large messages into smaller ones;
- to recover from resource allocation failure. A module's or driver's `put(9E)` routine can test for the availability of a resource, and if it is not available, enqueue the message for later processing by the `srv()` routine.

A message is first passed to a module's or driver's `put(9E)` routine, which may or may not do some processing. It must then either:

- Pass the message to the next stream component with `putnext(9F)`.
- If a `srv()` routine has been included, it may call `putq(9F)` to place the message on the queue.

Once a message has been enqueued, the STREAMS scheduler controls the service routine's invocation. The scheduler calls the service routines in FIFO order. The scheduler cannot guarantee a maximum delay `srv()` routine to be called except that it will happen before any user level process are run.

Every stream component (stream head, module or driver) has limit values it uses to implement flow control. Each component should check the tunable high and low water marks to stop and restart the flow of message processing. Flow control limits apply only between two adjacent components with `srv()` routines.

---

**Driver Entry Points**

101
STREAMS messages can be defined to have up to 256 different priorities to support requirements for multiple bands of data flow. At a minimum, a stream must distinguish between normal (priority zero) messages and high priority messages (such as M_IOCACK). High priority messages are always placed at the head of the srv() routine's queue, after any other enqueued high priority messages. Next are messages from all included priority bands, which are enqueued in decreasing order of priority. Each priority band has its own flow control limits. If a flow controlled band is stopped, all lower priority bands are also stopped.

Once the STREAMS scheduler calls a srv() routine, it must process all messages on its queue. The following steps are general guidelines for processing messages. Keep in mind that many of the details of how a srv() routine should be written depend on the implementation, the direction of flow (upstream or downstream), and whether it is for a module or a driver.

1. Use getq(9F) to get the next enqueued message.
2. If the message is high priority, process (if appropriate) and pass to the next stream component with putnext(9F).
3. If it is not a high priority message (and therefore subject to flow control), attempt to send it to the next stream component with a srv() routine. Use bcanputnext(9F) to determine if this can be done.
4. If the message cannot be passed, put it back on the queue with putbq(9F). If it can be passed, process (if appropriate) and pass with putnext().

Return Values Ignored.

See Also put(9E), bcanput(9F), bcanputnext(9F), canput(9F), canputnext(9F), getq(9F), nulldev(9F), putbq(9F), putnext(9F), putq(9F), qinit(9S), queue(9S)

Writing Device Drivers

STREAMS Programming Guide

Warnings Each stream module must specify a read and a write service srv() routine. If a service routine is not needed (because the put() routine processes all messages), a NULL pointer should be placed in module's qinit(9S) structure. Do not use nulldev(9F) instead of the NULL pointer. Use of nulldev(9F) for a srv() routine can result in flow control errors.
strategy(9E)

Name  strategy – perform block I/O

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

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

Interface Level  Architecture independent level 1 (DDI/DKI). This entry point is required for block devices.

Parameters  
- `bp`  Pointer to the `buf(9S)` structure.

Description  The `strategy()` routine is called indirectly (through `cb_ops(9S)`) by the kernel to read and write blocks of data on the block device. `strategy()` may also be called directly or indirectly to support the raw character interface of a block device (`read(9E)`, `write(9E)` and `ioctl(9E)`). The `strategy()` routine’s responsibility is to set up and initiate the transfer.

In general, `strategy()` should not block. It can, however, perform a `kmem_cache_create(9F)` with both the `KM_PUSHPAGE` and `KM_SLEEP` flags set, which might block, without causing deadlock in low memory situations.

Return Values  The `strategy()` function must return 0. On an error condition, it should call `bioerror(9F)` to set `b_flags` to the proper error code, and call `biodone(9F)`. Note that a partial transfer is not considered to be an error.

See Also  `ioctl(9E)`, `read(9E)`, `write(9E)`, `biodone(9F)`, `bioerror(9F)`, `buf(9S)`, `cb_ops(9S)`, `kmem_cache_create(9F)`

Writing Device Drivers
**Name**  tran_abort – abort a SCSI command

**Synopsis**  #include <sys/scsi/scsi.h>

```c
int prefixtran_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
```

**Interface Level**  Solaris architecture specific (Solaris DDI).

**Arguments**
- `ap`  Pointer to a `scsi_address(9S)` structure.
- `pkt`  Pointer to a `scsi_pkt(9S)` structure.

**Description**  The `tran_abort()` vector in the `scsi_hba_tran(9S)` structure must be initialized during the HBA driver’s `attach(9E)` to point to an HBA entry point to be called when a target driver calls `scsi_abort(9F)`.

`tran_abort()` should attempt to abort the command `pkt` that has been transported to the HBA. If `pkt` is NULL, the HBA driver should attempt to abort all outstanding packets for the target/logical unit addressed by `ap`.

Depending on the state of a particular command in the transport layer, the HBA driver may not be able to abort the command.

While the abort is taking place, packets issued to the transported layer may or may not be aborted.

For each packet successfully aborted, `tran_abort()` must set the `pkt_reason` to `CMD_ABORTED`, and `pkt_statistics` must be OR’ed with `STAT_ABORTED`.

**Return Values**  `tran_abort()` must return:

- 1  upon success or partial success.
- 0  upon failure.

**See Also**  `attach(9E), scsi_abort(9F), scsi_hba_attach(9F), scsi_address(9S), scsi_hba_tran(9S), scsi_pkt(9S)`

**Writing Device Drivers**

**Notes**  If `pkt_reason` already indicates that an earlier error had occurred, `tran_abort()` should not overwrite `pkt_reason` with `CMD_ABORTED`. 
Name tran_bus_reset – reset a SCSI bus

Synopsis #include <sys/scsi/scsi.h> int prefix

    tran_bus_reset(dev_info_t *hba_dip, int level);

Interface Level Solaris DDI

Parameters hba_dip The dev_info_t pointer associated with the SCSI HBA.
    level The level of reset required.

Description The tran_bus_reset() vector in the scsi_hba_tran(9S) structure should be initialized during the HBA driver's attach(9E). It is an HBA entry point to be called when a user initiates a bus reset through device control interfaces.

    tran_bus_reset() must reset the SCSI bus without resetting targets.

    level will be one of the following:

    RESET_BUS Reset the SCSI bus only, not the targets.

    Implementation is hardware specific. If it is not possible to reset the SCSI bus without changing the state and operating mode of the targets, the HBA driver should not initialize this vector or return failure.

Return Values tran_bus_reset() should return:

    1 on success.
    0 on failure.

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>Interface Stability</td>
<td>Committed</td>
</tr>
</tbody>
</table>

See Also attributes(5), tran_quiesce(9E), scsi_hba_tran(9S)
**Name**
tran_dmafree – SCSI HBA DMA deallocation entry point

**Synopsis**
#include <sys/scsi/scsi.h>

```c
void prefixtran_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt);
```

**Interface Level**
Solaris architecture specific (Solaris DDI).

**Arguments**
ap A pointer to a scsi_address structure. See scsi_address(9S).
pkt A pointer to a scsi_pkt structure. See scsi_pkt(9S).

**Description**
The `tran_dmafree()` vector in the scsi_hba_tran structure must be initialized during the HBA driver's attach() to point to an HBA entry point to be called when a target driver calls scsi_dmafree(9F). See attach(9E) and scsi_hba_tran(9S).

`tran_dmafree()` must deallocate any DMA resources previously allocated to this pkt in a call to tran_init_pkt(9E). `tran_dmafree()` should not free the structure pointed to by pkt itself. Since tran_destroy_pkt(9E) must also free DMA resources, it is important that the HBA driver keeps accurate note of whether scsi_pkt(9S) structures have DMA resources allocated.

**See Also**
attach(9E), tran_destroy_pkt(9E), tran_init_pkt(9E), scsi_dmafree(9F),
scsi_dmaget(9F), scsi_hba_attach(9F), scsi_init_pkt(9F), scsi_address(9S),
scsi_hba_tran(9S), scsi_pkt(9S)

**Writing Device Drivers**

**Notes**
A target driver may call tran_dmafree() on packets for which no DMA resources were allocated.
Name tran_getcap, tran_setcap – get/set SCSI transport capability

Synopsis #include <sys/scsi/scsi.h>

```c
int prefixtran_getcap(struct scsi_address *ap, char *cap, int whom);
int prefixtran_setcap(struct scsi_address *ap, char *cap, int value,
    int whom);
```

Interface Level Solaris architecture specific (Solaris DDI).

Parameters
- **ap** Pointer to the `scsi_address(9S)` structure.
- **cap** Pointer to the string capability identifier.
- **value** Defines the new state of the capability.
- **whom** Specifies whether all targets or only the specified target is affected.

Description
The `tran_getcap()` and `tran_setcap()` vectors in the `scsi_hba_tran(9S)` structure must be initialized during the HBA driver's `attach(9E)` to point to HBA entry points to be called when a target driver calls `scsi_ifgetcap(9F)` and `scsi_ifsetcap(9F)`.

`tran_getcap()` is called to get the current value of a capability specific to features provided by the HBA hardware or driver. The name of the capability `cap` is the NULL terminated capability string.

If `whom` is non-zero, the request is for the current value of the capability defined for the target specified by the `scsi_address(9S)` structure pointed to by `ap`; if `whom` is 0, all targets are affected; else, the target specified by the `scsi_address` structure pointed to by `ap` is affected.

`tran_setcap()` is called to set the value of the capability `cap` to the value of `value`. If `whom` is non-zero, the capability should be set for the target specified by the `scsi_address(9S)` structure pointed to by `ap`; if `whom` is 0, all targets are affected; else, the target specified by the `scsi_address` structure pointed to by `ap` is affected. It is recommended that HBA drivers do not support setting capabilities for all targets, that is, `whom` is 0.

A device may support only a subset of the defined capabilities.

Refer to `scsi_ifgetcap(9F)` for the list of defined capabilities.

HBA drivers should use `scsi_hba_lookup_capstr(9F)` to match `cap` against the canonical capability strings.

Return Values `tran_setcap()` must return 1 if the capability was successfully set to the new value, 0 if the HBA driver does not support changing the capability, and −1 if the capability was not defined.
tran_getcap() must return the current value of a capability or −1 if the capability was not defined.

See Also  attach(9E), scsi_hba_attach(9F), scsi_hba_lookup_capstr(9F), scsi_ifgetcap(9F), scsi_address(9S), scsi_hba_tran(9S)

Writing Device Drivers
 tran_init_pkt, tran_destroy_pkt – SCSI HBA packet preparation and deallocation

Synopsis  
#include <sys/scsi/scsi.h>

struct scsi_pkt *prefixtran_init_pkt(struct scsi_address *ap, 
struct scsi_pkt *pkt, struct buf *bp, int cmdlen, 
int statuslen, int tgtlen, int flags, int (*callback, 
caddr_t), caddr_t arg);

void prefixtran_destroy_pkt(struct scsi_address *ap, 
struct scsi_pkt *pkt);

Parameters  
ap  
Pointer to a scsi_address(9S) structure.

pkt  
Pointer to a scsi_pkt(9S) structure allocated in an earlier call, or NULL.

bp  
Pointer to a buf(9S) structure if DMA resources are to be allocated for the pkt, or NULL.

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.

tgtlen  
The length of the packet private area within the scsi_pkt to be allocated on behalf of the SCSI target driver.

flags  
Flags for creating the packet.

callback  
Pointer to either NULL_FUNC or SLEEP_FUNC.

arg  
Always NULL.

Description  
The tran_init_pkt() and tran_destroy_pkt() vectors in the scsi_hba_tran structure must be initialized during the HBA driver's attach(9E) to point to HBA entry points to be called when a target driver calls scsi_init_pkt(9F) and scsi_destroy_pkt(9F).

tran_init_pkt()  
tran_init_pkt() is the entry point into the HBA which is used to allocate and initialize a scsi_pkt structure on behalf of a SCSI target driver. If pkt is NULL, the HBA driver must use scsi_hba_pkt_alloc(9F) to allocate a new scsi_pkt structure.

If bp is non-NULL, the HBA driver must allocate appropriate DMA resources for the pkt, for example, through ddi_dma_buf_setup(9F) or ddi_dma_buf_bind_handle(9F).

If the PKT_CONSISTENT bit is set in flags, the buffer was allocated by scsi_alloc_consistent_buf(9F). For packets marked with PKT_CONSISTENT, the HBA driver must synchronize any cached data transfers before calling the target driver's command completion callback.
If the PKT_DMA_PARTIAL bit is set in flags, the HBA driver should set up partial data transfers, such as setting the DDI_DMA_PARTIAL bit in the flags argument if interfaces such as ddi_dma_buf_setup(9F) or ddi_dma_buf_bind_handle(9F) are used.

If only partial DMA resources are available, tran_init_pkt() must return in the pkt_resid field of pkt the number of bytes of DMA resources not allocated.

If both pkt and bp are non-NULL, if the PKT_DMA_PARTIAL bit is set in flags, and if DMA resources have already been allocated for the pkt with a previous call to tran_init_pkt() that returned a non-zero pkt_resid field, this request is to move the DMA resources for the subsequent piece of the transfer.

The contents of scsi_address(9S) pointed to by ap are copied into the pkt_address field of the scsi_pkt(9S) by scsi_hba_pkt_alloc(9F).

tgtlen is the length of the packet private area in the scsi_pkt structure to be allocated on behalf of the SCSI target driver.

statuslen is the required length for the SCSI status completion block. If the requested status length is greater than or equal to sizeof(struct scsi_arq_status) and the auto_rqsense capability has been set, automatic request sense (ARS) is enabled for this packet. If the status length is less than sizeof(struct scsi_arq_status), automatic request sense must be disabled for this pkt.

If the HBA driver is not capable of disabling ARQ on a per-packet basis and tran_init_pkt() is called with a statuslen that is less than sizeof(struct scsi_arq_status), the driver’s tran_init_pkt routine should allocate at least sizeof(struct scsi_arq_status). If an ARS is needed, upon successful ARS done by the HBA driver, the driver must copy the sense data over and set STAT_ARQ_DONE in pkt_state.

cmdlen is the required length for the SCSI command descriptor block.

Note: tgtlen, statuslen, and cmdlen are used only when the HBA driver allocates the scsi_pkt(9S), in other words, when pkt is NULL.

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.

tran_destroy_pkt() is the entry point into the HBA that must free all of the resources that were allocated to the scsi_pkt(9S) structure during tran_init_pkt().

Return Values tran_init_pkt() must return a pointer to a scsi_pkt(9S) structure on success, or NULL on failure.
If `pkt` is `NULL` on entry, and `tran_init_pkt()` allocated a packet through `scsi_hba_pkt_alloc(9F)` but was unable to allocate DMA resources, `tran_init_pkt()` must free the packet through `scsi_hba_pkt_free(9F)` before returning `NULL`.

**See Also**
- `attach(9E)`, `tran_sync_pkt(9E)`, `biodone(9F)`, `bioerror(9F)`, `ddi_dma_buf_bind_handle(9F)`, `ddi_dma_buf_setup(9F)`, `scsi_alloc_consistent_buf(9F)`, `scsi_destroy_pkt(9F)`, `scsi_hba_attach(9F)`, `scsi_hba_pkt_alloc(9F)`, `scsi_hba_pkt_free(9F)`, `scsi_init_pkt(9F)`, `buf(9S)`, `scsi_address(9S)`, `scsi_hba_tran(9S)`, `scsi_pkt(9S)`

**Writing Device Drivers**

**Notes**  
If a DMA allocation request fails with `DDI_DMA_NOMAPPING`, indicate the error by calling `bioerror(9F)` with `bp` and an error code of `EFAULT`.

If a DMA allocation request fails with `DDI_DMA_TOO_BIG`, indicate the error by calling `bioerror(9F)` with `bp` and an error code of `EINVAL`. 
tran_quiesce(9E)

Name tran_quiesce, tran_unquiesce – quiesce and unquiesce a SCSI bus

Synopsis #include <sys/scsi/scsi.h>

int prefixtran_quiesce(dev_info_t *hba_dip);
int prefixtran_unquiesce(dev_info_t *hba_dip);

Interface Level Solaris DDI

Parameters hba_dip The dev_info_t pointer associated with the SCSI HBA.

Description The tran_quiesce() and tran_unquiesce() vectors in the scsi_hba_tran(9S) structure should be initialized during the HBA driver’s attach(9E). They are HBA entry points to be called when a user initiates quiesce and unquiesce operations through device control interfaces.

tran_quiesce() should wait for all outstanding commands to complete and blocks (or queues) any I/O requests issued. tran_unquiesce() should allow I/O activities to resume on the SCSI bus.

Implementation is hardware specific.

Return Values tran_quiesce() and tran_unquiesce() should return:

0 Successful completion.
Non-zero An error occurred.

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>Interface Stability</td>
<td>Committed</td>
</tr>
</tbody>
</table>

See Also attributes(5), tran_bus_reset(9E), scsi_hba_tran(9S)
 tran_reset(9E)

**Name**
tran_reset – reset a SCSI bus or target

**Synopsis**
#include <sys/scsi/scsi.h>

```c
int prefixtran_reset(struct scsi_address *ap, int level);
```

**Interface Level**
Solaris architecture specific (Solaris DDI).

**Parameters**
- `ap` Pointer to the `scsi_address(9S)` structure.
- `level` The level of reset required.

**Description**
The `tran_reset()` vector in the `scsi_hba_tran(9S)` structure must be initialized during the HBA driver’s `attach(9E)` to point to an HBA entry point to be called when a target driver calls `scsi_reset(9F)`.

`tran_reset()` must reset either the SCSI bus, a SCSI target device, or a SCSI logical unit as specified by `level`.

`level` must be one of the following:
- `RESET_ALL` Reset the SCSI bus.
- `RESET_TARGET` Reset the target specified by `ap`.
- `RESET_LUN` Reset the logical unit specified by `ap`.

`tran_reset` should set the `pkt_reason` field of all outstanding packets in the transport layer associated with each target or logical unit that was successfully reset to `CMD_RESET` and the `pkt_statistics` field must be OR’ed with either `STAT_BUS_RESET` (if the SCSI bus was reset) or `STAT_DEV_RESET` (if the target or logical unit was reset).

The HBA driver should use a SCSI Bus Device Reset Message to reset a target device. The HBA driver should use a SCSI Logical Unit Reset Message to reset a logical unit.

Packets that are in the transport layer but not yet active on the bus should be returned with `pkt_reason` set to `CMD_RESET` and `pkt_statistics` OR’ed with `STAT_ABORTED`.

Support for `RESET_LUN` is optional but strongly encouraged for new and updated HBA drivers. If an HBA driver provides `RESET_LUN` support, it must also create the `lun-reset` capability with a value of zero for each target device instance represented by a valid `ap`. The HBA is also required to provide the means to return the current value of the `lun-reset` capability in its `tran_getcap(9E)` routine, as well as the means to change the value of the `lun-reset` capability in its `tran_getcap(9E)` routine.
tran_reset(9E)

Return Values  tran_reset() should return:

1  on success.
0  on failure.

See Also  attach(9E), ddi_dma_buf_setup(9F), scsi_hba_attach(9F), scsi_reset(9F),
          scsi_address(9S), scsi_hba_tran(9S)

Writing Device Drivers

Notes  If pkt_reason already indicates that an earlier error had occurred for a particular pkt,
tran_reset() should not overwrite pkt_reason with CMD_RESET.
Name: tran_reset_notify – request to notify SCSI target of bus reset

Synopsis: #include <sys/scsi/scsi.h>

```
int prefixtran_reset_notify(struct scsi_address *ap, int flag,
   void (*callback, caddr_t),caddr_t arg);
```

Interface Level: Solaris architecture specific (Solaris DDI).

Parameters:
- `ap`: Pointer to the `scsi_address(9S)` structure.
- `flag`: A flag indicating registration or cancellation of a notification request.
- `callback`: A pointer to the target driver’s reset notification function.
- `arg`: The callback function argument.

Description: The `tran_reset_notify()` entry point is called when a target driver requests notification of a bus reset.

The `tran_reset_notify()` vector in the `scsi_hba_tran(9S)` structure may be initialized in the HBA driver’s `attach(9E)` routine to point to the HBA entry point to be called when a target driver calls `scsi_reset_notify(9F)`.

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.
- `SCSI_RESET_CANCEL`: Cancel the reset notification request for the target.

The HBA driver maintains a list of reset notification requests registered by the target drivers. When a bus reset occurs, the HBA driver notifies registered target drivers by calling the callback routine, `callback`, with the argument, `arg`, for each registered target.

Return Values:
- For `SCSI_RESET_NOTIFY` requests, `tran_reset_notify()` must return `DDI_SUCCESS` if the notification request has been accepted, and `DDI_FAILURE` otherwise.
- For `SCSI_RESET_CANCEL` requests, `tran_reset_notify()` must return `DDI_SUCCESS` if the notification request has been canceled, and `DDI_FAILURE` otherwise.

See Also: `attach(9E), scsi_ifgetcap(9F), scsi_reset_notify(9F), scsi_address(9S), scsi_hba_tran(9S)`

*Writing Device Drivers*
**Name**  tran_start – request to transport a SCSI command

**Synopsis**  
```
#include <sys/scsi/scsi.h>

int prefixtran_start(struct scsi_address *ap, struct scsi_pkt *pkt);
```

**Interface Level**  Solaris architecture specific (Solaris DDI).

**Parameters**  
- `pkt`  Pointer to the `scsi_pkt(9S)` structure that is about to be transferred.
- `ap`  Pointer to a `scsi_address(9S)` structure.

**Description**  
The `tran_start()` vector in the `scsi_hba_tran(9S)` structure must be initialized during the HBA driver's `attach(9E)` to point to an HBA entry point to be called when a target driver calls `scsi_transport(9F)`.

`tran_start()` must perform the necessary operations on the HBA hardware to transport the SCSI command in the `pkt` structure to the target/logical unit device specified in the `ap` structure.

If the flag `FLAG_NOINTR` is set in `pkt_flags` in `pkt`, `tran_start()` should not return until the command has been completed. The command completion callback `pkt_comp` in `pkt` must not be called for commands with `FLAG_NOINTR` set, since the return is made directly to the function invoking `scsi_transport(9F)`.

When the flag `FLAG_NOINTR` is not set, `tran_start()` must queue the command for execution on the hardware and return immediately. The member `pkt_comp` in `pkt` indicates a callback routine to be called upon command completion.

Refer to `scsi_pkt(9S)` for other bits in `pkt_flags` for which the HBA driver may need to adjust how the command is managed.

If the `auto_rqsense` capability has been set, and the status length allocated in `tran_init_pkt(9E)` is greater than or equal to `sizeof(struct scsi_arq_status)`, automatic request sense is enabled for this `pkt`. If the command terminates with a Check Condition, the HBA driver must arrange for a Request Sense command to be transported to that target/logical unit, and the members of the `scsi_arq_status` structure pointed to by `pkt_scbp` updated with the results of this Request Sense command before the HBA driver completes the command pointed by `pkt`.

The member `pkt_time` in `pkt` is the maximum number of seconds in which the command should complete. Timeout starts when the command is transmitted on the SCSI bus. A `pkt_time` of 0 means no timeout should be performed.

For a command which has timed out, the HBA driver must perform some recovery operation to clear the command in the target, typically an Abort message, or a Device or Bus Reset.
pkt_reason member of the timed out pkt should be set to CMD_TIMEOUT, and pkt_statistics OR’ed with STAT_TIMEOUT. If the HBA driver can successfully recover from the timeout, pkt_statistics must also be OR’ed with one of STAT_ABORTED, STAT_BUS_RESET, or STAT_DEV_RESET, as appropriate. This informs the target driver that timeout recovery has already been successfully accomplished for the timed out command. The pkt_comp completion callback, if not NULL, must also be called at the conclusion of the timeout recovery.

If the timeout recovery was accomplished with an Abort Tag message, only the timed out packet is affected, and the packet must be returned with pkt_statistics OR’ed with STAT_ABORTED and STAT_TIMEOUT.

If the timeout recovery was accomplished with an Abort message, all commands active in that target are affected. All corresponding packets must be returned with pkt_reason, CMD_TIMEOUT, and pkt_statistics OR’ed with STAT_TIMEOUT and STAT_ABORTED.

If the timeout recovery was accomplished with a Device Reset, all packets corresponding to commands active in the target must be returned in the transport layer for this target. Packets corresponding to commands active in the target must be returned with pkt_reason set to CMD_TIMEOUT, and pkt_statistics OR’ed with STAT_DEV_RESET and STAT_TIMEOUT. Currently inactive packets queued for the device should be returned with pkt_reason set to CMD_RESET and pkt_statistics OR’ed with STAT_ABORTED.

If the timeout recovery was accomplished with a Bus Reset, all packets corresponding to commands active in the target must be returned in the transport layer. Packets corresponding to commands active in the target must be returned with pkt_reason set to CMD_TIMEOUT and pkt_statistics OR’ed with STAT_TIMEOUT and STAT_BUS_RESET. All queued packets for other targets on this bus must be returned with pkt_reason set to CMD_RESET and pkt_statistics OR’ed with STAT_ABORTED.

Note that after either a Device Reset or a Bus Reset, the HBA driver must enforce a reset delay time of ‘scsi-reset-delay’ milliseconds, during which time no commands should be sent to that device, or any device on the bus, respectively.

tran_start() should initialize the following members in pkt to 0. Upon command completion, the HBA driver should ensure that the values in these members are updated to accurately reflect the states through which the command transitioned while in the transport layer.

- **pkt_resid**: For commands with data transfer, this member must be updated to indicate the residual of the data transferred.

- **pkt_reason**: The reason for the command completion. This field should be set to CMD_CMPLT at the beginning of tran_start(), then updated if the command ever transitions to an abnormal
termination state. To avoid losing information, do not set pkt_reason to any other error state unless it still has its original CMD_CMPLT value.

pkt_statistics
Bit field of transport-related statistics.

pkt_state
Bit field with the major states through which a SCSI command can transition. Note: The members listed above, and pkt_hba_private member, are the only fields in the scsi_pkt(9S) structure which may be modified by the transport layer.

Return Values tran_start() must return:

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/logical unit, the HBA 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, or the packet could not be accepted for other reasons.

TRAN_FATAL_ERROR A fatal error has occurred in the HBA.

See Also attach(9E), tran_init_pkt(9E), scsi_hba_attach(9F), scsi_transport(9F), scsi_address(9S), scsi_arq_status(9S), scsi_hba_tran(9S), scsi_pkt(9S)

Writing Device Drivers
tran_sync_pkt - SCSI HBA memory synchronization entry point

#include <sys/scsi/scsi.h>

void prefixtran_sync_pkt(struct scsi_address *ap,
                        struct scsi_pkt *pkt);

**Interface Level**
Solaris architecture specific (Solaris DDI).

**Parameters**
- *ap* A pointer to a `scsi_address(9S)` structure.
- *pkt* A pointer to a `scsi_pkt(9S)` structure.

**Description**
The `tran_sync_pkt()` vector in the `scsi_hba_tran(9S)` structure must be initialized during the HBA driver’s `attach(9E)` to point to an HBA driver entry point to be called when a target driver calls `scsi_sync_pkt(9F)`.

`tran_sync_pkt()` must synchronize a HBA’s or device’s view of the data associated with the `pkt`, typically by calling `ddi_dma_sync(9F)`. The operation may also involve HBA hardware-specific details, such as flushing I/O caches, or stalling until hardware buffers have been drained.

**See Also**
- `attach(9E)`, `tran_init_pkt(9E)`, `ddi_dma_sync(9F)`, `scsi_hba_attach(9F)`,
- `scsi_init_pkt(9F)`, `scsi_sync_pkt(9F)`, `scsi_address(9S)`, `scsi_hba_tran(9S)`,
- `scsi_pkt(9S)`

*Writing Device Drivers*

**Notes**
A target driver may call `tran_sync_pkt()` on packets for which no DMA resources were allocated.
Name tran_tgt_free – request to free HBA resources allocated on behalf of a target

Synopsis #include <sys/scsi/scsi.h>

    void prefixtran_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
                             scsi_hba_tran_t *hba_tran, struct scsi_device *sd);

Interface Level Solaris architecture specific (Solaris DDI).

Parameters

    hba_dip Pointer to a dev_info_t structure, referring to the HBA device instance.
    tgt_dip Pointer to a dev_info_t structure, referring to the target device instance.
    hba_tran Pointer to a scsi_hba_tran(9S) structure, consisting of the HBA's transport vectors.
    sd Pointer to a scsi_device(9S) structure, describing the target.

Description The tran_tgt_free() vector in the scsi_hba_tran(9S) structure may be initialized during the HBA driver's attach(9E) to point to an HBA driver function to be called by the system when an instance of a target device is being detached. The tran_tgt_free() vector, if not NULL, is called after the target device instance has returned successfully from its detach(9E) entry point, but before the dev_info node structure is removed from the system. The HBA driver should release any resources allocated during its tran_tgt_init() or tran_tgt_probe() initialization performed for this target device instance.

See Also attach(9E), detach(9E), tran_tgt_init(9E), tran_tgt_probe(9E), scsi_device(9S), scsi_hba_tran(9S)

Writing Device Drivers
**Name**
tran_tgt_init – request to initialize HBA resources on behalf of a particular target

**Synopsis**
```
#include <sys/scsi/scsi.h>

void prefixtran_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
                        scsi_hba_tran_t *hba_tran, struct scsi_device *sd);
```

**Interface Level**
Solaris architecture specific (Solaris DDI).

**Parameters**
- **hba_dip**
  Pointer to a dev_info_t structure, referring to the HBA device instance.
- **tgt_dip**
  Pointer to a dev_info_t structure, referring to the target device instance.
- **hba_tran**
  Pointer to a scsi_hba_tran(9S) structure, consisting of the HBA’s transport vectors.
- **sd**
  Pointer to a scsi_device(9S) structure, describing the target.

**Description**
The tran_tgt_init() vector in the scsi_hba_tran(9S) structure may be initialized during the HBA driver’s attach(9E) to point to an HBA driver function to be called by the system when an instance of a target device is being created. The tran_tgt_init() vector, if not NULL, is called after the dev_info node structure is created for this target device instance, but before probe(9E) for this instance is called. Before receiving transport requests from the target driver instance, the HBA may perform any initialization required for this particular target during the call of the tran_tgt_init() vector.

Note that hba_tran will point to a cloned copy of the scsi_hba_tran_t structure allocated by the HBA driver if the SCSI HBA.TRAN_CLONE flag was specified in the call to scsi_hba_attach(9F). In this case, the HBA driver may choose to initialize the tran_tgt_private field in the structure pointed to by hba_tran, to point to the data specific to the particular target device instance.

**Return Values**
tran_tgt_init() must return:
- **DDI_SUCCESS**
  the HBA driver can support the addressed target, and was able to initialize per-target resources.
- **DDI_FAILURE**
  the HBA driver cannot support the addressed target, or was unable to initialize per-target resources. In this event, the initialization of this instance of the target device will not be continued, the target driver’s probe(9E) will not be called, and the tgt_dip structure destroyed.

**See Also**
attach(9E), probe(9E), tran_tgt_free(9E), tran_tgt_probe(9E),
scsi_hba_attach_setup(9F), scsi_device(9S), scsi_hba_tran(9S)

*Writing Device Drivers*
Name tran_tgt_probe – request to probe SCSI bus for a particular target

Synopsis #include <sys/scsi/scsi.h>

int prefixtran_tgt_probe(struct scsi_device *sd, int (*waitfunc, void));

Interface Level Solaris architecture specific (Solaris DDI).

Parameters

sd Pointer to a scsi_device(9S) structure.

waitfunc Pointer to either NULL_FUNC or SLEEP_FUNC.

Description The tran_tgt_probe() vector in the scsi_hba_tran(9S) structure may be initialized during the HBA driver’s attach(9E) to point to a function to be called by scsi_probe(9F) when called by a target driver during probe(9E) and attach(9E) to probe for a particular SCSI target on the bus. In the absence of an HBA-specific tran_tgt_probe() function, the default scsi_probe(9F) behavior is supplied by the function scsi_hba_probe(9F).

The possible choices the HBA driver may make are:

■ Initialize the tran_tgt_probe vector to point to scsi_hba_probe(9F), which results in the same behavior.
■ Initialize the tran_tgt_probe vector to point to a private function in the HBA, which may call scsi_hba_probe(9F) before or after any necessary processing, as long as all the defined scsi_probe(9F) semantics are preserved.

waitfunc indicates what tran_tgt_probe() should do when resources are not available:

NULL_FUNC Do not wait for resources. See scsi Probe(9F) for defined return values if no resources are available.

SLEEP_FUNC Wait indefinitely for resources.

See Also attach(9E), probe(9E), tran_tgt_free(9E), tran_tgt_init(9E), scsi_hba_probe(9F), scsi_probe(9F), scsi_device(9S), scsi_hba_tran(9S)

Writing Device Drivers
write – write data to a device

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int prefixwrite(dev_t dev, struct uio *uio_p, cred_t *cred_p);

Architecture independent level 1 (DDI/DKI). This entry point is optional.

Parameters

- dev: Device number.
- uio_p: Pointer to the uio(9S) structure that describes where the data is to be stored in user space.
- cred_p: Pointer to the user credential structure for the I/O transaction.

Description

Used for character or raw data I/O, the driver write() routine is called indirectly through cb_ops(9S) by the write(2) system call. The write() routine supervises the data transfer from user space to a device described by the uio(9S) structure.

The write() routine should check the validity of the minor number component of dev and the user credentials pointed to by cred_p, if pertinent.

Return Values

The write() routine should return 0 for success, or the appropriate error number.

Examples

The following is an example of a write() routine using physio(9F) to perform writes to a seekable device:

```c
static int
xxwrite(dev_t dev, struct uio *uio, cred_t *cred)
{   int instance;
    xx_t xx;

    instance = getminor(dev);
    xx = ddi_get_soft_state(xxstate, instance);
    if (xx == NULL)
        return (ENXIO);
    return (physio(xxstrategy, NULL, dev, B_WRITE,
                   xxmin, uiop));
}
```
See Also  read(2), write(2), read(9E), physio(9F), cb_ops(9S), uio(9S)

Writing Device Drivers