Preface

OVERVIEW

A man page is provided for both the naive user, and sophisticated user who is familiar with the SunOS operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

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

- Section 1 describes, in alphabetical order, commands available with the operating system.

- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.

- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.

- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.

- Section 5 contains miscellaneous documentation such as character set tables, etc.

- Section 6 contains available games and demos.

- Section 7 describes various special files that refer to specific hardware peripherals, and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.

- Section 9 provides reference information needed to write device drivers in the kernel operating systems environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver–Kernel Interface (DKI).

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

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

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

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

NAME

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

SYNOPSIS

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

The following special characters are used in this section:

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

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

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

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

**PROTOCOL**

This section occurs only in subsection 3R to indicate the protocol description file. The protocol specification pathname is always listed in **bold** font.

**AVAILABILITY**

This section briefly states any limitations on the availability of the command. These limitations could be hardware or software specific.

A specification of a class of hardware platform, such as x86 or SPARC, denotes that the command or interface is applicable for the hardware platform specified.

In Section 1 and Section 1M, **AVAILABILITY** indicates which package contains the command being described on the manual page. In order to use the command, the specified package must have been installed with the operating system. If the package was not installed, see **pkgadd(1)** for information on how to upgrade.

**MT-LEVEL**

This section lists the **MT-LEVEL** of the library functions described in the Section 3 manual pages. The **MT-LEVEL** defines the libraries’ ability to support threads. See **Intro(3)** for more information.
DESCRIPTION

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

IOCTL

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

OPTIONS

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

OPERANDS

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

OUTPUT

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

RETURN VALUES

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

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

USAGE

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

- Commands
- Modifiers
- Variables
- Expressions
- Input Grammar

EXAMPLES

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

\texttt{example%}

or if the user must be super-user,

\texttt{example#}

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

ENVIRONMENT

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

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

FILES

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

SEE ALSO

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

DIAGNOSTICS

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

WARNINGS

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

NOTES

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

BUGS

This section describes known bugs and wherever possible suggests workarounds.
NAME
Intro, intro – introduction to device driver entry points

DESCRIPTION
Section 9E describes the entry-point routines a developer may 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. 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>dump</td>
<td>Solaris DDI</td>
</tr>
</tbody>
</table>

modified 21 Sep 1994
The table below lists the error codes that should be returned by a driver routine when an error is encountered. It lists the error values in alphabetic order. All the error values 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 returning the value. 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, as the function that it calls, `cmn_err(9F)`, is declared as `void` (no error is returned).
### Error Values

<table>
<thead>
<tr>
<th>Error Value</th>
<th>Error Description</th>
<th>Use in these Driver Routines (9E)</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).</td>
<td>open, ioctl, read, write, strategy</td>
</tr>
<tr>
<td>EFAULT</td>
<td>An invalid address has been passed as an argument; memory addressing error.</td>
<td>open, close, ioctl, read, write, strategy</td>
</tr>
<tr>
<td>EINTR</td>
<td>Sleep interrupted by signal.</td>
<td>open, close, ioctl, read, write, strategy</td>
</tr>
<tr>
<td>EINVAL</td>
<td>An invalid argument was passed to the routine.</td>
<td>open, ioctl, read, write, 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).</td>
<td>open, close, ioctl, read, write, 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.</td>
<td>open, close, ioctl, read, write, strategy</td>
</tr>
<tr>
<td>EPERM</td>
<td>A process attempting an operation did not have required permission.</td>
<td>open, ioctl, read, write, close</td>
</tr>
<tr>
<td>EROFS</td>
<td>An attempt was made to open for writing a read-only device.</td>
<td>open</td>
</tr>
</tbody>
</table>

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>EINTR</td>
<td>EIO</td>
</tr>
<tr>
<td>ENXIO</td>
<td>EPERM</td>
<td>EROFS</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aread(9E)</td>
<td>asynchronous read from a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attach(9E)</td>
<td>attach a device to the system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awrite(9E)</td>
<td>asynchronous write to a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chpoll(9E)</td>
<td>poll entry point for a non-STREAMS character driver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(9E)</td>
<td>relinquish access to a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>detach(9E)</td>
<td>detach a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dump(9E)</td>
<td>dump memory to device during system failure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_fini(9E)</td>
<td>loadable module configuration entry points</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getinfo(9E)</td>
<td>get device driver information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>identify(9E)</td>
<td>claim to drive a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_info(9E)</td>
<td>See _fini(9E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_init(9E)</td>
<td>See _fini(9E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioctl(9E)</td>
<td>control a character device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ks_update(9E)</td>
<td>dynamically update kstats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapdev_access(9E)</td>
<td>device mapping access entry point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapdev_dup(9E)</td>
<td>device mapping duplication entry point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapdev_free(9E)</td>
<td>device mapping free entry point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmap(9E)</td>
<td>check virtual mapping for memory mapped device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>open(9E)</td>
<td>gain access to a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print(9E)</td>
<td>display a driver message on system console</td>
<td></td>
<td></td>
</tr>
<tr>
<td>probe(9E)</td>
<td>determine if a non-self-identifying device is present</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prop_op(9E)</td>
<td>report driver property information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put(9E)</td>
<td>receive messages from the preceding queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(9E)</td>
<td>read data from a device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>segmap(9E)</td>
<td>map device memory into user space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srv(9E)</td>
<td>service queued messages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strategy(9E)</td>
<td>perform block I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_abort(9E)</td>
<td>abort a SCSI command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_destroy_pkt(9E)</td>
<td>See tran_init_pkt(9E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_dmafree(9E)</td>
<td>SCSI HBA DMA deallocation entry point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_getcap(9E)</td>
<td>get/set SCSI transport capability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_init_pkt(9E)</td>
<td>SCSI HBA packet preparation and deallocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tran_reset(9E)</td>
<td>reset a SCSI bus or target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

modified 21 Sep 1994
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran_reset_notify(9E)</td>
<td>request to notify SCSI target of bus reset</td>
</tr>
<tr>
<td>tran_setcap(9E)</td>
<td>See tran_getcap(9E)</td>
</tr>
<tr>
<td>tran_start(9E)</td>
<td>request to transport a SCSI command</td>
</tr>
<tr>
<td>tran_sync_pkt(9E)</td>
<td>SCSI HBA memory synchronization entry point</td>
</tr>
<tr>
<td>tran_tgt_free(9E)</td>
<td>request to free HBA resources allocated on behalf of a target</td>
</tr>
<tr>
<td>tran_tgt_init(9E)</td>
<td>request to initialize HBA resources on behalf of a particular target</td>
</tr>
<tr>
<td>tran_tgt_probe(9E)</td>
<td>request to probe SCSI bus for a particular target</td>
</tr>
<tr>
<td>write(9E)</td>
<td>write data to a device</td>
</tr>
</tbody>
</table>
NAME
_fini, _info, _init – loadable module configuration entry points

SYNOPSIS
#include <sys/modctl.h>
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.

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

DESCRIPTION
_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 _finit() returns the value returned by mod_remove(9F). Upon successful return from _finit() 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, else 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).

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

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

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

modified 29 Jun 1995
static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    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, "XXX Global Mutex",
               MUTEX_DRIVER, (void *)NULL);

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

    return (i);
}

int _info(struct modinfo *modinfop)
{
    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),
modldrv(9S), modlinkage(9S), modlstrmod(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().
NAME aread – asynchronous read from a device

SYNOPSIS
#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);

ARGUMENTS
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.

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).

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. However, the aread() routine must not modify aio_reqp or the contents of the aio_req(9S) structure. aread() must call aphysio(9F) with the aio_req pointer, and a pointer to the driver’s strategy(9E) routine.

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 The following is an example of an aread() routine.

modified 9 Nov 1994
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));
}

SEE ALSO read(2), aioread(3), awrite(9E), read(9E), strategy(9E), write(9E), anocancel(9F), aphsiso(9F), ddi_get_soft_state(9F), drv_priv(9F), getminor(9F), minphys(9F), aio_req(9S), cb_ops(9S), uio(9S)

Writing Device Drivers

BUGS There is no way other than calling aphsiso(9F) to accomplish an asynchronous read.
NAME
attach – attach a device to the system

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

int prefix attach(dev_info_t *dip, ddi_attach_cmd_t cmd);

INTERFACE
Solaris DDI specific (Solaris DDI). This entry point is required and must be written.

LEVEL
ARGUMENTS
dip A pointer to the device’s dev_info structure.

cmd Attach type. Should be set to DDI_ATTACH. Other values are reserved. The driver should return DDI_FAILURE if reserved values are passed to it.

DESCRIPTION
attach() is the device-specific initialization entry point. When attach() 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.

attach() will be called once for each instance of the device on the system. Until attach() succeeds, the only driver entry points which may be called are open(9E) and getinfo(9E). See the “Autoconfiguration” chapter in Writing Device Drivers. The instance number may be obtained using ddi_get_instance(9F).

Successful returns from identify(9E) and probe(9E) are required before a call to the driver’s attach() entry point will be made.

RETURN VALUES
attach() should return:

DDI_SUCCESS on success.

DDI_FAILURE on failure.

SEE ALSO
identify(9E), probe(9E), ddi_add_intr(9F), ddi_create_minor_node(9F), ddi_get_instance(9F), ddi_map_regs(9F), kmem_alloc(9F),
Writing Device Drivers
NAME  awrite – asynchronous write to a device

SYNOPSIS  
#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);

ARGUMENTS  
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.

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).

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. However, the awrite() routine must not modify aio_reqp or the contents of the aio_req(9S) structure. awrite() must call aphysio(9F) with the aio_req pointer, and a pointer to the driver’s strategy(9E) routine.

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  The following is an example of an awrite() routine.
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)
        return (ENXIO);
    return (aphysio(xxstrategy, anocancel, dev, B_WRITE, xxminphys, aio));
}

SEE ALSO  write(2), aiowrite(3), aread(9E), read(9E), strategy(9E), write(9E), anocancel(9F),
aphysio(9F), ddi_get_soft_state(9F), drv_priv(9F), getminor(9F), minphys(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.
NAME
chpoll – poll entry point for a non-STREAMS character driver

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

int chpoll(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).

ARGUMENTS

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.
POLLWRNORM The same as POLLOUT.
POLLWRBAND Priority data (priority band > 0) may be written.

anyyet A flag that is non-zero if any other file descriptors in the pollfd array have
            events pending. The poll(2) system call takes a pointer to an array of pollfd
            structures as one of its arguments. See 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 fol-
lowing rules must be followed when implementing the polling discipline:
1. Implement the following algorithm when the chpoll() entry point is called:
   
   ```c
   if (events_are_satisfied_now) {
     *reventsp = mask_of_satisfied_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. None of its fields should be referenced. However, the size of this structure is guaranteed to remain the same across releases.

3. Call the pollwakeup() function whenever an event of type events listed above occur. This function should only be called with one event at a time.

**RETURN VALUES**

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

**SEE ALSO**

poll(2), pollwakeup(9F)

Writing Device Drivers

**NOTES**

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

close – relinquish access to a device

SYNOPSIS

Block and Character

#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 prefix close(dev_t dev, int flag, int otyp, cred_t *cred_p);

STREAMS

#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 prefix close(queue_t *q, int flag, cred_t *cred_p);

INTERFACE

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

LEVEL

ARGUMENTS

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, but the kernel will only call the `close()` routine when the last process using the device issues a `close(2)` or `umount(2)` system call or exits. (An exception is a close occurring with the `otyp` argument set to `OTYP_LYR`, for which a close (also having `otyp = OTYP_LYR`) occurs for each open.)

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.

SEE ALSO

`close(2)`, `detach(9E)`, `open(9E)`

*Writing Device Drivers*

*STREAMS Programming Guide*
NAME  
detach – detach 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). This entry point is required. It can be nodev.

ARGUMENTS  
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 is passed to it.

DESCRIPTION  
detach() is the complement of the attach(9E) routine. It is used to remove all the states associated with a given instance of a device node prior to the removal of that instance from the system. The dev_info nodes that belong to a driver are removed as part of the process of unloading a device driver from the system.

Depending on what was created in the drivers’ attach(9E) routine, this might mean calling ddi_unmap_regs() (see ddi_map_regs(9F)) to remove mappings, calling ddi_remove_intr() (see ddi_add_intr(9F)) to unregister interrupt handlers, calling kmem_free(9F) to free any heap allocations, and so forth. This should also include putting the underlying device into a quiescent state so that it will not generate interrupts.

If detach() determines that the particular instance of the device cannot be removed when requested, for example, because of some exceptional condition, detach() returns DDI_FAILURE, which prevents the particular device instance from being removed. This will also prevent the driver from being unloaded.

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

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.

RETURN VALUES  
DDI_SUCCESS  
The state associated with the given device was successfully removed.

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.

SEE ALSO  
attach(9E), open(9E), ddi_add_INTR(9F), ddi_map_REGS(9F), kmem_free(9F), timeout(9F)

Writing Device Drivers

modified 11 Mar 1992
### NAME
dump – dump memory to device during system failure

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

int prefixdump(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` should be used.

### ARGUMENTS
- `dev` Device number.
- `addr` address for the beginning of the area to be dumped.
- `blkno` Block offset to dump memory to.
- `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. 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.

dump() is called at interrupt priority.

### RETURN VALUES
dump() should return 0 on success, or the appropriate error number.

### SEE ALSO
Writing Device Drivers
NAME
getinfo – get device driver information

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

int prefix_getinfo(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_DEVT2DEVINFO and DDI_INFO_DEVT2INSTANCE.
arg  Command specific argument.
resultp  Pointer to where the requested information is stored.

DESCRIPTION
When cmd is set to DDI_INFO_DEVT2DEVINFO, 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_DEVT2INSTANCE, 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
/*ARGSUSED*/
static int
rd_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{

  /* Note that in this simple example
   * the minor number is the instance
   * number.
   */

  devstate_t *sp;
  int error = DDI_FAILURE;

modified 1 May 1992
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 *) getminor((dev_t) arg);
        error = DDI_SUCCESS;
        break;
}

    return (error);
}

SEE ALSO nodev(9F), cb_ops(9S), dev_ops(9S)

Writing Device Drivers
### NAME
identify – claim to drive a device

### SYNOPSIS
```c
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
int prefix identify(dev_info_t *dip);
```

### INTERFACE LEVEL
Solaris DDI specific (Solaris DDI). This entry point is **required**. You **must** write it.

### ARGUMENTS
- `dip` A pointer to a `dev_info` structure.

### DESCRIPTION
`identify()` determines whether this driver drives the device pointed to by `dip`.

### RETURN VALUES
- `DDI_IDENTIFIED` if it **claims** to drive this device.
- `DDI_NOT_IDENTIFIED` if it does **not** claim to drive this device.

### EXAMPLES
```c
#define XX_NAME "xx"
static int xxidentify(dev_info_t *dip) {
    if (strcmp(ddi_get_name(dip), XX_NAME) == 0) {
        /* name matches device name */
        return(DDI_IDENTIFIED);
    } else
        return(DDI_NOT_IDENTIFIED);
}
```

### SEE ALSO
- `attach(9E)`, `ddi_get_name(9F)`, `strcmp(9F)`
- *Writing Device Drivers*

### WARNINGS
This routine may be called multiple times. It may also be called at any time. The driver should not infer anything from the the sequence or the number of times this entry point has been called.
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 prefix ioctl(dev_t dev, int cmd, int arg, int mode, cred_t *cred_p, int *rval_p);

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

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 an integer 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  Contains values set when the device was opened. Use of this mode is optional. However, 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 for the ioctl routine's mode argument.

In some circumstances, mode is used to provide 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 `FKIOCTL` 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)`.

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`.

**SEE ALSO**  
`ttymon(1M)`, `stty(1)`, `dkio(7I)`, `fbio(7I)`, `termio(7I)`, `open(9E)`, `put(9E)`, `srv(9E)`, `copyin(9F)`, `copyout(9F)`, `ddi_copyin(9F)`, `ddi_copyout(9F)`, `cb_ops(9S)`

Writing DeviceDrivers

**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.

modified 27 Jun 1995
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_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    Solaris DDI specific (Solaris DDI)

LEVEL

ARGUMENTS   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.

modified 27 May 1994
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  
mapdev_access – device mapping access entry point

SYNOPSIS  
#include <sys/sunddi.h>

int prefix mapdev_access(ddi_mapdev_handle_t handle, void *devprivate, off_t offset);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
handle  An opaque pointer to a device mapping.
devprivate Driver private mapping data from ddi_mapdev(9F).
offset  The offset within device memory at which the access occurred.

DESCRIPTION  
mapdev_access() is called when an access is made to a mapping that has either been newly created with ddi_mapdev(9F) or that has been enabled with a call to ddi_mapdev_intercept(9F).

mapdev_access() is passed the handle of the mapped object on which an access has occurred. This handle uniquely identifies the mapping and is used as an argument to ddi_mapdev_intercept(9F) or ddi_mapdev_nointercept(9F) to control whether or not future accesses to the mapping will cause mapdev_access() to be called. In general, mapdev_access() should call ddi_mapdev_intercept() on the mapping that is currently in use and then call ddi_mapdev_nointercept() on the mapping that generated this call to mapdev_access(). This will ensure that a call to mapdev_access() will be generated for the current mapping next time it is accessed.

mapdev_access() must at least call ddi_mapdev_nointercept() with offset passed in in order for the access to succeed. A request to allow accesses affects the entire page containing the offset.

Accesses to portions of mappings that have been disabled by a call to ddi_mapdev_nointercept() will not generate a call to mapdev_access(). A subsequent call to ddi_mapdev_intercept() will enable mapdev_access() to be called again.

A non-zero return value from mapdev_access() will cause the corresponding operation to fail. The failure may result in a SIGSEGV or SIGBUS signal being delivered to the process.

RETURN VALUES  
mapdev_access() should return 0 on success, -1 if there was a hardware error, or the return value from ddi_mapdev_intercept() or ddi_mapdev_nointercept().

CONTEXT  
This function is called from user context only.

modified 15 Feb 1994
The following shows an example of managing a device context that is one page in length.

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

SEE ALSO mmap(2), mapdev_dup(9E), mapdev_free(9E), segmap(9E), ddi_mapdev(9F), ddi_mapdev_intercept(9F), ddi_mapdev_nointercept(9F), ddi_mapdev_ctl(9S), Writing Device Drivers
NAME  
mapdev_dup – device mapping duplication entry point

SYNOPSIS  
#include <sys/sunddi.h>

int prefix mapdev_dup(ddi_mapdev_handle_t handle, void *devprivate,  
                      ddi_mapdev_handle_t new_handle, void **new_devprivatep);

INTERFACE LEVEL  
Solaris DDI specific (Solaris DDI).

ARGUMENTS  
handle  The handle of the mapping that is being duplicated.
devprivate  Driver private mapping data from the mapping that is being duplicated.
new_handle  An opaque pointer to the duplicated device mapping.
new_devprivatep  A pointer to be filled in by the driver with the driver private mapping
data for the duplicated device mapping.

DESCRIPTION  
mapdev_dup() is called when a device mapping is duplicated such as through fork(2).  
mapdev_dup() is expected to generate new driver private data for the new mapping, and  
set new_devprivatep to point to it.  new_handle is the handle of the new mapped object.  
A non-zero return value from mapdev_dup() will cause the corresponding operation,  
such as fork() to fail.

RETURN VALUES  
mapdev_dup() returns 0 for success or the appropriate error number on failure.

CONTEXT  
This function is called from user context only.

EXAMPLES  
static int  
xxmapdev_dup(ddi_mapdev_handle_t handle, void *devprivate,  
              ddi_mapdev_handle_t new_handle, void **new_devprivate)  
{  
    struct xxpvtdata *pvtdata;  
    /* Allocate a new private data structure */  
    pvtdata = kmalloc(sizeof (struct xxpvtdata), KM_SLEEP);  
    /* Copy the old data to the new - device dependent*/  
    ...  
    /* Return the new data */  
    *new_pvtdata = pvtdata;  
    return (0);  
}

SEE ALSO  
fork(2), mmap(2), mapdev_access(9E), mapdev_free(9E), segmap(9E), ddi_mapdev(9F),  
ddi_mapdev_intercept(9F), ddi_mapdev_nointercept(9F), ddi_mapdev_ctl(9S),  
Writing Device Drivers

modified 28 Feb 1994
NAME mapdev_free – device mapping free entry point

SYNOPSIS #include <sys/sunddi.h>
void mapdev_free(ddi_mapdev_handle_t handle, void *devprivate);

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS
handle An opaque pointer to a device mapping.
devprivate Driver private mapping data from ddi_mapdev(9F).

DESCRIPTION mapdev_free() is called when a mapping created by ddi_mapdev(9F) is being destroyed. mapdev_free() receives the handle of the mapping being destroyed and a pointer to the driver private data for this mapping in devprivate.

The mapdev_free() routine is expected to free any resources that were allocated by the driver for this mapping.

CONTEXT This function is called from user context only.

EXAMPLES
static void
xxmapdev_free(ddi_mapdev_handle_t hdl, void *pvtdata)
{
    /* Destroy the driver private data - Device dependent */
    ...
    kmem_free(pvtdata, sizeof (struct xxpvtdata));
}

SEE ALSO exit(2), mmap(2), munmap(2), mapdev_access(9E), mapdev_dup(9E), segmap(9E), ddi_mapdev(9F), ddi_mapdev_intercept(9F), ddi_mapdev_nointercept(9F), ddi_mapdev_ctl(9S)

Writing Device Drivers
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>
#include <sys/sunddi.h>

int prefix_mmap(dev_t dev, off_t off, int prot);

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

ARGUMENTS
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
The mmap(9E) 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(9E) entry point is called as a result of an mmap(2) system call, and also as a result of a page fault. mmap(9E) is called to translate the offset off in device memory to the corresponding physical page frame number.

The mmap(9E) 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(9E) returns the value returned by hat_getkpfnum(9F) for the physical page in device memory containing the offset off.

hat_getkpfnum(9F) accepts a kernel virtual address as an argument. A kernel virtual address can be obtained by calling ddi_map_regs(9F) in the driver’s attach(9E) routine. The corresponding ddi_unmap_regs(9F) call can be made in the driver’s detach(9E) routine. Refer to the EXAMPLES section below for more information.

mmap(9E) should only be supported for memory-mapped devices. See the segmap(9E) and ddi_mapdev(9F) reference pages for further information on memory-mapped device drivers.

modified 2 Sep 1994
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.

The following is an example of the `mmap(9E)` 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_map_regs(9F)` can be used to obtain this address. For example, `ddi_map_regs(9F)` can be called in the driver’s `attach(9E)` routine. The resulting kernel virtual address is stored in the `xxstate` structure (see `ddi_soft_state(9F)`), which is accessible from the driver’s `mmap(9E)` entry point. The corresponding `ddi_unmap_regs(9F)` call can be made in the driver’s `detach(9E)` routine.

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

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

struct xxstate *xsp;
...

static int xxmmmap(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));
}
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_map_regs(9F)` and `ddi_unmap_regs(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 which are not locked down may cause unexpected and undefined behavior.
open – gain access to a device

SYNOPSIS
Block and Character
#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 prefix open(dev_t *devp, int flag, int otyp, cred_t *cred_p);

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

int prefix open(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).

ARGUMENTS
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, then allow both read and write access)
  - FWRITE: Open a device with write-only permission (if ORed with FREAD, then allow both read and write access)

- **otyp**: Parameter supplied so that the driver can determine how many times a device was opened and for what reasons.

For OTYP_BLK and OTYP_CHR, the open() routine may be called many times, but the close(9E) routine is called only when the last reference to a device is removed. If the device is accessed through file descriptors, this is by a call to close(2) or exit(2). If the device is accessed through memory mapping, this is by a call to munmap(2) or exit(2).

For OTYP_LYR, there is exactly one close(9E) for each open() called. This permits software drivers to exist above hardware drivers and removes any ambiguity from the hardware driver regarding how a device is used.

modified 13 Jan 1993
** 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) routine. The calling driver will make sure 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.

STREAMS

- *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 routine 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() routine is called by the kernel during an open(2) or a mount(2) on the special file for the device. The routine should verify that the minor number component of *devp* is valid, that the type of access requested by *otyp* and *flag* is appropriate for the device, and, if required, check permissions using the user credentials pointed to by *cred_p*.

The open() routine 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), 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.

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

modified 13 Jan 1993
RETURN VALUES

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

SEE ALSO

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)

Writing Device Drivers
STREAMS Programming Guide

WARNINGS

Do not attempt to change the major number.
NAME
print – display a driver message on system console

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

int prefix print(dev_t dev, char *str);

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

ARGUMENTS
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

modified 15 Sep 1992
NAME  probe – determine if a non-self-identifying device is present

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

static int prefetchprobe(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) and/or ddi.poke(9F) and determine if the device exists. Then the device registers should be unmapped using ddi.unmap_regs(9F).

probe() should only probe the device – it should not create or change 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

9E-44  modified 18 Nov 1992
NAME  prop_op – report driver property information

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

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

INTERFACE  Solaris DDI specific (Solaris DDI). This entry point is required, but it can be
LEVEL  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
          Don’t 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 users
        buffer. If prop_op is PROP_LEN_AND_VAL_ALLOC, this should be the address
        of a pointer.
lengthp  On exit, *lengthp will contain the property length. If prop_op is
         PROP_LEN_AND_VAL_BUF then before calling prop_op(), lengthp should
         point to an int that contains the length of callers buffer.

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 an xxprop_op entry point, but most
drivers which 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_create(9F) to
create properties for its device.

modified 15 Dec 1993  9E-45
RETURN VALUES

prop_op() should return:

- **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

```c
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_NOTFOUND);

    if (!strcmp(name, "nblocks")) {
        ddi_prop_modify(dev, dip, "nblocks", flags,
                        &xsp->nblocks, sizeof(int));
    }
    /* other cases... */

    skip:
    return (ddi_prop_op(dev, dip, prop_op, flags, name,
                        valuep, lengthp));
}
```

SEE ALSO

- `ddi_prop_create(9F)`, `ddi_prop_op(9F)`
- *Writing Device Drivers*
NAME
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>

int prefix rput(queue_t *q, mblk_t *mp); /* read side */
int prefix wput(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 com-
ponent (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 excep-
tion 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)
• 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.

modified 12 Nov 1992
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), streamtab(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*
NAME
read – read data from a device

SYNOPSIS
#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 prefix read(dev_t dev, struct uio *uio_p, cred_t *cred_p);

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

ARGUMENTS
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.

SEE ALSO
read(2), write(9E), cb_ops(9S), uio(9S)
Writing Device Drivers
NAME  
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, caddr_t *addrp, off_t len,
                 unsigned int prot, unsigned int maxprot, unsigned int flags, cred_t *cred_p);

INTERFACE 
LEVEL 
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). If (maxprot & prot) != prot then there is an access violation.
flags  Flags indicating type of mapping. Possible values are (other bits may be set):
  MAP_SHARED  Changes should be shared.
  MAP_PRIVATE  Changes are private.
cred_p  Pointer to the user credentials structure.

DESCRIPTION
The segmap() entry point 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 `mmap(9E)` entry point, or both an `mmap(9E)` and a `segmap()` entry point routine (see the `mmap(9E)` reference page). If no `segmap()` entry point is provided for the driver, `ddi_segmap(9F)` is used as a default.

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

- needs to maintain a separate context for each user mapping. See `ddi_mapdev(9F)` for details.
- needs to assign device access attributes to the user mapping. See `ddi_segmap_setup(9F)` and `ddi_mapdev_set_device_acc_attr(9F)` for details.

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 `mmap(9E)` entry point. Note that if you are using `ddi_segmap()` to set up the mapping, it will call your `mmap(9E)` entry point for you to validate the range to be mapped.
- Assign device access attributes to the mapping. See `ddi_mapdev_set_device_acc_attr(9F)`, `ddi_segmap_setup(9F)`, and `ddi_device_acc_attr(9S)` for details.
- Set up device contexts for the user mapping if your device requires context switching. See `ddi_mapdev(9F)` for details.
- Perform the mapping with `ddi_segmap()`, `ddi_segmap_setup()`, or `ddi_mapdev()` and return the status if it fails.

**RETURN VALUES**

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_segmap()`, `ddi_segmap_setup()`, or `ddi_mapdev()` fails, you must return the error number returned by the respective routine.

**SEE ALSO**

`mmap(2)`, `mmap(9E)`, `ddi_mapdev(9F)`, `ddi_mapdev_set_device_acc_attr(9F)`, `ddi_segmap(9F)`, `ddi_segmap_setup(9F)`, `ddi_device_acc_attr(9S)`

*Writing Device Drivers*
**NAME**

srv – service queued messages

**SYNOPSIS**

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

int prefix rsrv(queue_t *q); /* read side */
int prefix wsrv(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.

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 of 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), nuldev(9F) putbq(9F), putnext(9F), putq(9F), 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 nuldev(9F) instead of the NULL pointer. Use of nuldev(9F) for a srv() routine may result in flow control errors.

modified 12 Nov 1992
NAME       strategy – perform block I/O

SYNOPSIS   
#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.

ARGUMENTS 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.

RETURN VALUES The strategy() routine should always return 0. On an error condition, it should OR the b_flags member of the buf(9S) structure with B_ERROR, set the b_error member to the appropriate error value, 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), buf(9S), cb_ops(9S)

Writing Device Drivers
NAME  tran_abort – abort a SCSI command

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

int prefix tran_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  on success or partial success.
0  on 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.

modified 30 Aug 1995  9E-55
NAME  tran_dmafree – SCSI HBA DMA deallocation entry point

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

void prefix tran_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt);

INTERFACE LEVEL  Solaris architecture specific (Solaris DDI).

ARGUMENTS  
ap  A pointer to a scsi_address(9S) structure.
pkt  A pointer to a scsi_pkt(9S) structure.

DESCRIPTION  The tran_dmafree() 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_dmafree(9F).

tran_dmafree() must deallocate any DMA resources previously allocated to this pkt in a call to tran_init_pkt(). tran_dmafree() should not free the structure pointed to by pkt itself. Since tran_destroy_pkt() 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_init_pkt(9F), scsi_hba_attach(9F), scsi_hba_tran(9S), scsi_address(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>

int prefix tran_getcap(struct scsi_address *ap, char *cap, int whom);
int prefix tran_setcap(struct scsi_address *ap, char *cap, int value, int whom);

INTERFACE LEVEL
Solaris architecture specific (Solaris DDI).

ARGUMENTS

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>Pointer to the scsi_address(9S) structure.</td>
</tr>
<tr>
<td>cap</td>
<td>Pointer to the string capability identifier.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the new state of the capability.</td>
</tr>
<tr>
<td>whom</td>
<td>Specifies whether all targets or only the specified target is affected.</td>
</tr>
</tbody>
</table>

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.

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

modified 30 Aug 1995
NAME  tran_init_pkt, tran_destroy_pkt – SCSI HBA packet preparation and deallocation

SYNOPSIS  

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

struct scsi_pkt *prefix tran_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 prefix tran_destroy_pkt(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 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)`.

The `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, via `ddi_dma_buf_setup(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)` are used.

 modified 1 Mar 1995
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 `pktp` and `bp` are non-NULL, the `PKT_DMA_PARTIAL` bit is set in `flags` and 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 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`.

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 `pkt` via `scsi_hba_pkt_alloc(9F)` but was unable to allocate DMA resources, `tran_init_pkt()` must free the `pkt` via `scsi_hba_pkt_free(9F)` before returning NULL.

**SEE ALSO**

`attach(9E), tran_sync_pkt(9E), 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`, the `B_ERROR` flag should be set in `bp`, and the `b_error` field should be set to `EFAULT`.

If a DMA allocation request fails with `DDI_DMA_TOOBIG`, the `B_ERROR` flag should be set in `bp`, and the `b_error` field should be set to `EINVAL`.

modified 1 Mar 1995
tran_init_pkt(9E)  Driver Entry Points  SunOS 5.5

9E-60  modified 1 Mar 1995
NAME	tran_reset – reset a SCSI bus or target

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

int prefix tran_reset(struct scsi_address *ap, int level);

INTERFACE
LEVEL
Solaris architecture specific (Solaris DDI).

ARGUMENTS
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 the SCSI bus or a SCSI target 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.

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

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

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.

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 prefix tran_reset_notify(struct scsi_address *ap, int flag, void (*callback)(caddr_t),
adv_t arg);

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

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 cal-
ing 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

9E-62 modified 30 Aug 1995
NAME tran_start – request to transport a SCSI command

SYNOPSIS

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

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

INTERFACE LEVEL Solaris architecture specific (Solaris DDI).

ARGUMENTS

- `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 `pkt` 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. A `pkt_time` of zero 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. The `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 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAN_ACCEPT</td>
<td>The packet was accepted by the transport layer.</td>
</tr>
<tr>
<td>TRAN_BUSY</td>
<td>The packet could not be accepted because there was already</td>
</tr>
<tr>
<td></td>
<td>a packet in progress for this target/logical unit, the HBA</td>
</tr>
<tr>
<td></td>
<td>queue was full, or the target device queue was full.</td>
</tr>
<tr>
<td>TRAN_BADPKT</td>
<td>The DMA count in the packet exceeded the DMA engine’s</td>
</tr>
<tr>
<td></td>
<td>maximum DMA size, or the packet could not be accepted for</td>
</tr>
<tr>
<td></td>
<td>other reasons.</td>
</tr>
<tr>
<td>TRAN_FATAL_ERROR</td>
<td>A fatal error has occurred in the HBA.</td>
</tr>
</tbody>
</table>

### SEE ALSO

- `attach(9E)`
- `tran_init_pkt(9E)`
- `scsi_hba_attach(9F)`
- `scsi_transport(9F)`
- `scsi_address(9S)`
- `scsi_hba_tran(9S)`
- `scsi_pkt(9S)`

*Writing Device Drivers*
**NAME**  
tran_sync_pkt – SCSI HBA memory synchronization entry point

**SYNOPSIS**  
```c  
#include <sys/scsi/scsi.h>  
void prefixtran_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt);  
```

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

**ARGUMENTS**  
- `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 CPU’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**  
- `tran_init_pkt(9E)`, `ddi_dma_sync(9F)`, `scsi_hba_attach(9F)`, `scsi_init_pkt(9F)`, `scsi_sync_pkt(9F)`, `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 prefix tran_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).

ARGUMENTS
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**
```c
#include <sys/scsi/scsi.h>

void prefix tran_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).

**ARGUMENTS**
- **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 device 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` 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_device`(9S), `scsi_hba_tran`(9S)
- *Writing Device Drivers*

9E-68 modified 1 Nov 1993
NAME  tran_tgt_probe – request to probe SCSI bus for a particular target

SYNOPSIS  

```c
#include <sys/scsi/scsi.h>
int prefix tran_tgt_probe(struct scsi_device *sd, int (*waitfunc)(void));
```

INTERFACE LEVEL  Solaris architecture specific (Solaris DDI).

ARGUMENTS  

- `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:

- to initialize the `tran_tgt_probe` vector to point to `scsi_hba_probe`(9F), which results in the same behavior.
- to 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), `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

modified 1 Nov 1993
NAME
write – write data to a device

SYNOPSIS
#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 prefix write(dev_t dev, struct uio *uio_p, cred_t *cred_p);

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

ARGUMENTS
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.

SEE ALSO read(2), read(9E), cb_ops(9S), uio(9S)
Writing Device Drivers
Index

A
aread — asynchronous read from a device, 9E-13
asynchronous read — aread, 9E-13
asynchronous write — awrite, 9E-16
awrite — asynchronous write to a device, 9E-16

C
character-oriented drivers
— ioctl, 9E-28

D
DDI device mapping
mapdev_access — device mapping access entry point, 9E-33
mapdev_dup — device mapping duplication entry point, 9E-35
mapdev_free — device mapping free entry point, 9E-36
dev_info structure
convert device number to — getinfo, 9E-25
device access
— close, 9E-21
— open, 9E-41
device mapping access entry point —
mapdev_access, 9E-33
device mapping duplication entry point —
mapdev_dup, 9E-35
device mapping free entry point — mapdev_free,
9E-36
device number
convert to dev_info structure — getinfo, 9E-25
devices
attach to system — attach, 9E-15
claim to drive a device — identify, 9E-27
detach from system — detach, 9E-23
read data — read, 9E-49
write data to a device — write, 9E-70
devices, memory mapped
check virtual mapping — mmap, 9E-37
devices, memory mapping
map device memory into user space — segmap, 9E-50
devices, non-self-identifying
determine if present — probe, 9E-44
Driver entry point routines
— _fini, 9E-10
— _info, 9E-10
— _init, 9E-10
— attach, 9E-15
— chpoll, 9E-18
— close, 9E-21
— detach, 9E-23
— dump, 9E-24
— getinfo, 9E-25
— identify, 9E-27
Driver entry point routines, continued
  — ioctl, 9E-28
  — mmap, 9E-37
  — open, 9E-41
  — print, 9E-43
  — probe, 9E-44
  — prop_op, 9E-45
  — put, 9E-47
  — read, 9E-49
  — segmap, 9E-50
  — srv, 9E-52
  — strategy, 9E-54
  — write, 9E-70

driver messages
  display on system console — print, 9E-43

driver property information
  report — prop_op, 9E-45
drivers, character-oriented
  — ioctl, 9E-28
dump — dump memory to disk during system failure, 9E-24
dynamically update kstats — ks_update, 9E-31

G
get/set SCSI transport capability — tran_getcap, 9E-57
  tran_setcap, 9E-57

H
HBA resources
  request to free HBA resources allocated on behalf of a target —
    tran_tgt_free, 9E-67
  request to initialize HBA resources on behalf of a particular target —
    tran_tgt_init, 9E-68

I
identify — claim to drive a device, 9E-27

K
kernel modules, dynamic loading
  initialize a loadable module — _init, 9E-10
  prepare loadable module for unloading —

_fini,
kernel modules, dynamic loading, continued
  9E-10
  return loadable module information — _info, 9E-10
  ks_update — dynamically update kstats, 9E-31

M
mapdev_access — device mapping access entry point, 9E-33
mapdev_dup — device mapping duplication entry point, 9E-35
mapdev_free — device mapping free entry point, 9E-36
memory mapping for devices
  check virtual mapping — mmap, 9E-37
  map device memory into user space — segmap, 9E-50

N
non-self-identifying devices
  determine if present — probe, 9E-44
  non-STREAMS character device driver poll entry point — chpoll, 9E-18

P
put — receive messages from the preceding queue, 9E-47

R
request to notify SCSI target of bus reset
  — tran_reset_notify, 9E-62
reset a SCSI bus or target — tran_reset, 9E-61

S
SCSI bus
  request to probe SCSI bus for a particular target
    — tran_tgt_probe, 9E-69
SCSI command
  abort — tran_abort, 9E-55
  request to transport — tran_start, 9E-63
SCSI HBA DMA deallocation entry point —
  tran_dmafree, 9E-56
SCSI HBA memory synchronization entry point — tran_sync_pkt, 9E-66
SCSI HBA packet preparation and deallocation —
  tran_init_pkt, 9E-58
  tran_destroy_pkt, 9E-58
strategy — perform block I/O, 9E-54
STREAMS message queues
  receive messages from the preceding queue —
    put, 9E-47
  service queued messages — srv, 9E-52

T
tran_abort — abort a SCSI command, 9E-55
tran_destroy_pkt — SCSI HBA packet preparation and deallocation, 9E-58
tran_dmafree — SCSI HBA DMA deallocation entry point, 9E-56
tran_getcap — get/set SCSI transport capability, 9E-57
tran_init_pkt — SCSI HBA packet preparation and deallocation, 9E-58
tran_reset — reset a SCSI bus or target, 9E-61
tran_reset_notify — request to notify SCSI target of bus reset, 9E-62
tran_setcap — get/set SCSI transport capability, 9E-57
tran_start — request to transport a SCSI command, 9E-63
tran_sync_pkt — SCSI HBA memory synchronization entry point, 9E-66
tran_tgt_free — request to free HBA resources allocated on behalf of a target, 9E-67
tran_tgt_init — request to initialize HBA resources on behalf of a particular target, 9E-68
tran_tgt_probe — request to probe SCSI bus for a particular target, 9E-69

V
virtual address space
  dump portion of to disk in case of system failure — dump, 9E-24

W
write — write data to a device, 9E-70