Loadable Modules

The loadable modules feature allows you to add a device driver or other kernel module to a running system without rebooting the system or rebuilding the kernel.

Loadable modules:

- reduce time spent on driver development by streamlining the driver installaion process
- make it easier for users to install drivers from other vendors
- improve system availability by allowing drivers to be configured into the kernel while the system is running
- conserve system resources by unloading infrequently used drivers when not in use
- provide users with the ability to load and unload drivers on demand
- provide the kernel with the ability to load drivers automatically

NOTE Drivers that are going to be configured into the system as loadable modules must be converted to loadable form.

This section contains four parts. The first part is an introduction to loadable modules from the driver writer's perspective. The second part explains the process of converting a non-loadable driver to be a loadable driver. The fourth part contains sections that explain the conversion process for each of the supported module-types. The third part explains the various procedures for installing, configuring, and maintaining a loadable driver.

Introduction to Loadable Modules

This section explains what a loadable module is, the various types of loadable modules, the differences between static and loadable modules, and the process of loading and unloading a module.

What is a Loadable Module?

A loadable module is a kernel module that can be loaded and unloaded during runtime. Previously, to install a new driver, you had to edit the system file, run **`config`** to create a new kernel, shut down the system, and then bring the system back up before you could use the new driver. Now, you can use a loadable module and avoid unnecessary work and delays.

Loadable modules also conserve system resources by enabling you to unload infrequently used drivers, thus keeping the kernel smaller.

Types of Loadable Modules

Some of the kernel modules that can be dynamically loaded and unloaded include:

- WSIO class and interface drivers
- device drivers
- STREAMS modules
- STREAMS drivers
- miscellaneous

The Difference Between Static Modules and Loadable Modules

Both static modules and dynamically loadable modules are maintained as individual object files. However, static modules are configured so that including a new module or excluding an existing module requires relinking the entire kernel and rebooting the system. Some modules continue to be linked in this traditional manner because they are required in the system (for example, the boot hard disk driver), or they are used so frequently or consume so few resources (for example, the user terminal pseudo–device driver) that it makes sense to keep the module continuously configured.

Dynamically loadable modules are configured so that a module can be loaded into the kernel when needed and can be unloaded when no longer needed. Some modules are linked in this manner because they are not required in the system, they are used infrequently, or they consume large amounts of resources.

Module Registration & Unregistration

Before a module may be loaded it must be registered with the system. Likewise, before a module may be unregistered with the system, it must first be unloaded.

Registration is a module-specific operation. Unique routines are provided to register each type of module. The modadm(2) system call provides a common interface for all module registration:

- modadm(type, MOD_C_MREG, arg)
- modadm(type, MOD_C_UREG, name)

where type may be one of the following:

Table 1 Module Types

Module Type	Description
MOD_TY_WSIO2	WSIO2 Drivers
MOD_TY_STR	STREAMS Modules
MOD_TY_MISC	Miscellaneous
MOD_TY_FS	File System
MOD_TY_EXEC	EXEC

where arg points to the following structure:

```
struct mod_mreg {            /* structure for module registration*/
   char md_modname[MODMAXNAMELEN];/* name of file for module*/
   void *md_typedata;             /* module type specific data*/
};
```

and where name points to a null terminated string indicating the module to unregister.

Registration establishes a correspondence between the unique module name and the device major number. This correspondence is used by the autoload mechanism to determine which module should be loaded for a given autoload event. For example, when an attempt is made to open a device with a specified major number and type, the system looks up the name that is registered to that major number for that type.

Unregistering the module breaks the association between the module's unique name and the device major number.

Rules for Module Registration and Unregistration. The following rules apply to module registration and unregistration:

1. Modules must be registered before they can be loaded.
2. A registered module can't be re-registered without first being un-registered.
3. A loaded module can't be unregistered.

Type Specific Interface. In the process of registering or unregistering a module, modadm() calls the type specific register/unregister routine provided by the type specific code for the module in question. These interfaces are:

- int32_t mod_type_reg(void *arg, void **tspecpp)

  Here arg is passed into the kernel from user space through the modadm() system call. Depending on type, arg may point to data in user space that needs to be copied into the kernel before it can be used. Typically, this routine will allocate a type specific datastructure and populate it with the information passed from user space. The address of this datastructure is passed back to the caller through tspecpp.

- int32_t mod_type_ureg(void *tspecp)

  The type specific unregister routine is passed a pointer to the type specific datastructure originally allocated and returned by the register routine, tspecp. This routine should free any memory associated with this pointer when it is done.

What Happens During Loading

When a module is dynamically loaded, its object file is read from disk and loaded into newly allocated kernel memory. Once in memory, the module's symbols are relocated and any external references are resolved. Special code in the module is then executed to perform any required module–specific setup. Then the code specific to the module's type, if any, is executed, making the newly loaded module accessible to the rest of the kernel.

Two types of events can cause a module to be loaded: a demand load and an autoload event.

Loading the Module. Loading the module into the kernel is accomplished in several steps:

- the module is loaded from the disk and link-edited into the kernel
- if the module has a defined _load entry point, it is called to perform any module-specific initialization and setup
- the module is logically connected to the rest of the kernel, often with the help of the module-specific installation routines accessed through the wrappers

The Difference Between a Demand Load and an Auto Load

A demand load is a user level request for a specific module to be loaded. The load is accomplished through the **`kmadmin`** command.

An autoload occurs when the kernel detects that a specific module is required to provide the functionality necessary to perform a task. The load is triggered by the initiation of the task. Once the required module is loaded, the task continues.

What Happens During Unloading

When the functionality provided by a module is no longer needed the module can be unloaded, thus freeing its resources for later use.

A module may be unloaded only by a user level request specifying the module to be unloaded. The unload is accomplished through the **`kmadmin`** command. This request may fail for a number of reasons, the most common being that the module is busy at the time. An example of this would be attempting to unload a device while there are outstanding opens on the device.

Unloading the Module. Unloading a module is the reverse of the load process and is also accomplished in several steps:

- the module is disconnected from the kernel
- if the module has a defined _unload entry point, it is called to perform any module-specific cleanup tasks:
  - freeing all dynamic memory
  - cancelling all outstanding calls to timeout(D3)
  - disabling device interupts
  - optionally disabling and / or powering down the device

Kernel Support for Loadable Modules

The kernel code supporting each module type contains a set of routines collectively referred to as the module operations for that type. The module operations are accessed through the following structure:

```
struct mod_operations {
int32_t (*modm_install)(); /* install module */
int32_t (*modm_remove)(); /* remove module */
int32_t (*modm_info)(); /* return module status */
};
```

One such structure is defined for each module type:

- modm_install is called after the module is loaded and initialized. Its purpose is to make the module accessible to the kernel through its normal interface.
- modm_remove is called before the module is unloaded. This routine disconnects the module from its normal interface, resetting the autoload mechanism if required.
- modm_info is called in the processing of the modstat(2) system call to obtain module specific information.

When designing support for a given module type, you may choose not to define any or all of these operations:

- The operation may not apply to the module type in question. For example, modules that are not accessed through standard kernel switch tables don't require modm_install or modm_remove routines, since they have no interfaces to install or remove.
- You may choose to implement the operations within the framework of the target subsystem instead of DLKM. Here, the install and remove routines are provided as published external interfaces of the subsystem in question. These subsystem provided install and remove routines must be called from the module's _load() and _unload() routines respectively. Refer to “Wrapper Functions” on page 9.

Creating a Loadable Module

The process of creating a loadable module consists of several phases:

- Creating the mandatory mod.o, master, and system files that make up the module fileset.
- Debugging the loadable module

- Packaging the loadable module as an SD fileset which includes the necessary control scripts.

NOTE

In some cases you may have an object file that can be converted into a separate loadable module. In these cases, the object file would no longer be built into the static library (but extracted from it). The master and system file for the loadable module would be created by extracting the required information from the old master and system files and putting it into the new files.

The following sections explain each of the phases and provides background information specific to device driver development.

Module File Set

The components of a loadable module's file set consists of the following files:

- mod.o—the object file of the module
- master—a configuration file. Refer to “Master File Definition” on page 7.
- system—a configuration file. Refer to “System File Definition” on page 8.
- space.h (optional)—a configuration file specifying the tunable parameters
- node (optional)—a configuration file specifying the special device file format for the module.
- Modstub.o (optional)—module using the DLKM stubs mechanism must supply this file which gets built into the kernel to permit dynamic loading of the module when needed.

Master File Definition

Each loadable module has its own master file. The format of the master file includes the following extensions:

- $LOADABLE—indicates if a module can be loadable.
- $TYPE—indicates the module type and the type specific information which will be used to generate the mod_reg file to register a module.
- $VERSION—indicates the version of the file.

NOTE The version number for the master file and system file should be the same.

- $INTERFACE—indicates the base interface versions on which the module is built and points to the interface file. Refer to “Interface File” on page 8.
- Other Sections—includes $DRIVER_INSTALL, $DRIVER_DEPENDENCY, $TUNABLE.

NOTE The $DEVICE section of the master file is provided for compatibility with previous HP-UX releases. New drivers should be added to the $DRIVER_INSTALL section.

System File Definition

Every loadable module requires a system file. The module’s system file will be copied to the /stand/system.d/ directory as <mod-name> by kminstall(1M). The system file includes the following fields:

- $VERSION—indicates the version of the file
- $LOADABLE—indicates whether a module needs to be configured as a loadable module. This field will not exist for static modules. kmsystem(1M) provides the interface to modify the flag.
- $CONFIGURE—indicates if the module needs to be configured into the system. If configure is "Y" then the module will be configured. If it is "N", it will not be configured. kmsystem(1M) provides the interface to modify the flag.
- $TUNABLE—contains tunable parameters for the loadable module. kmtune1M) is the interface to modify tunables both in the module's system file and in /stand/system.

config(1M) reads the tunable parameters specified in the /stand/system file to generate the tune.h file that will contain #defines for the tunable parameters. This header file will be included during the build and will make the tunable parameters globally available to multiple modules.

Interface File

Interface files describe the official symbols that may be accessed by a module. The interface files are maintained under /usr/conf/interface.d and should be named as <name>.<version>. Note that the <name> will correspond to the Field 1 of the $INTERFACE section of a master file for a module depending on the interfaces provided in that file. The version will correspond to Field 2 in the $INTERFACE section.

The format of the interface files should be as follows. Lines starting with (*)s will be comments.

- Field 1—symbol_name to be used.
- Field 2—(optional) new_symbol_name replacing the one specified in field 1.

The interface file may have an optional special entry $DEPEND with a list of modules, subsystems or drivers. This entry indicates the modules/ subsystems/drivers which should be built in or loaded before the module using this interface is loaded.

```
Example
* The name of this interface file is frick.1
foo
foo2
foo3 foo3-new
$DEPEND mod1 mod2.
```

Wrapper Functions

A loadable module is required to supply the {loadable module mechanism} with special initialization code called a wrapper. The wrapper “wraps” a module’s initialization and termination routines with special code that enables the {loadable module mechanism} to logically connect and disconnect the module to and from the kernel while the system is running.

The first step in converting a non-loadable module to a loadable module is to write the wrapper. Refer to the following sections for information on coding the wrapper for each of the supported module types:

- “Loadable WSIO2 Device Drivers” on page 10
- “Loadable STREAMS Device Drivers” on page 18
- “Loadable Miscellaneous Device Drivers” on page 21

Packaging a Loadable Module for Installation

Once you have written a wrapper for your loadable driver, you will compile your driver in the normal fashion. After compiling, you are ready to package the driver for installation. This section describes the procedures for packaging a loadable driver.

Debugging a Loadable Module

[section explaining procedures for debugging a loadable module.]

DLM Error Messages. [section explaining the basic DLKM error messages and how to retify them.]

nddb, fastdump, savecore. [section explaing the use of these commands for debugging a loadable module.]

Kernel Module Used in Loadable Modules

[section explaining why this is here]

Loadable WSIO2 Device Drivers

This section describes how WSIO2 device drivers may make use of the loadable module feature.

WSIO2 Device Driver Types

There are two types of WSIO2 loadable drivers. One type indicates driver's that require device switch table entries and the other type indicates those that are not accessed through the device switch tables. The following table details both types.

Type	Description	Example
MOD_TY_WSIO2_INTFC	WSIO2 interface drivers: do not have entries in the device switch tables.	scsi_c720, pc_fdc, fcpdev
MOD_TY_WSIO2_CLASS	WSIO2 class drivers and drivers that are both interface and class drivers (monolithic): require entries in the device switch tables.	scsi_tape, scsi_changer, audio

WSIO2 Registration

Register the correspondence between character and block major device numbers and the device driver's module name using the gio_mod_drv_reg() routine. In addition to name registration, this routine also initializes the autoload mechanism for registered drivers. The routine is called through the modadm(2) system call for all WSIO2 type modules. The algorithm for the gio_mod_drv_reg() routine is as follows:

- Copy in the registration data (struct mod_mreg).
- Acquire the appropriate switch table lock.
- If the module is currently registered, release the switch table lock and fail with an EEXIST error.
- Else, the module is being registered for the first time.
- Allocate kernel memory for the module's drv_info structure.
- Allocate kernel memory to store the module's name.
- Initialize the drv_info structure with md_typedata from the registration data and the pointer to the allocated name space.
- Copy the modules name from the registration data into the allocated name space.
- Call gio_allocate_majors() to get the device major numbers for this module.
- Initialize the autoload routine and drv_info structure pointers in the device switch table.
- Release the switch table lock.

NOTE

Initialization of the autoload mechanism will be discussed in detail in the WSIO2 Device Driver Autoload Procedure section. WSIO2 interface drivers are not registered. These drivers do not have entries in the device switch tables and therefore cannot be autoloaded and must be demand loaded. It is possible to have a driver that is both an interface driver and target device driver requiring entries in the device switch table. This driver would be registered as a MOD_TY_WSIO2_CLASS type.

WSIO2 Unregistration

Drivers that have been unloaded may be unregistered. Unregistration removes any references to the driver from the device switch tables and frees any allocated resources acquired during registration. Modules being unregistered must first be unloaded or unregistration will fail.

Unregistration is accomplished through the gio_mod_drv_unregister() routine. The routine is called through the modadm(2) system call for all WSIO2 type modules. The algorithm for this routine is as follows:

- Copy in the driver specific information.
- Acquire the appropriate switch table lock.
- Check if the driver is loaded. If it is loaded, release the switch table lock and fail with an EBUSY error.
- In the device switch tables, set all driver entry points to nodev, all pointers to data structures to NULL, and the flags field to indicate this slot's major number may be dynamically allocated.
- Free resources acquired during registration. These include the driver's name and class space and the drv_info structure.

WSIO2 Device Driver Wrapper

Once a module has been loaded into kernel address space and its symbol definitions and references have been relocated and resolved, it must be logically linked into the system. To facilitate this linking, each module needs to supply the DLM mechanism with certain module specific information. This information is supplied through additional module code known as the module's wrapper.

The device driver wrapper consists of both initialized data structures and function definitions. Since this information is specific for a given driver, each must supply its own wrapper.

Device Driver-Specific Wrapper Structures. The upper level wrapper structures are defined in the Dynamically Loadable Kernel Modules section. For WSIO2 device drivers there is no private data currently needed to accomplish the load. As such, the mtd_pdata field in the mod_type_data structure is set to NULL. WSIO2 drivers only provide a modm_info() routine for the mod_operations structure. Both the modm_install() and modm_remove() fields are set to NULL.

Wrapper Routines. As described in the Dynamically Loadable Kernel Modules section, device drivers may define _load(), _unload(), and _halt() routines to perform any driver specific initialization and cleanup.

- _load() Routine: After the module has been loaded and relocated, the _load() routine is called. The routine registers the driver with WSIO2 and performs any driver specific initialization required.

NOTE

See the following Sample Wrapper section for a typical _load and _unload example.

- _unload() Routine: Just before the module is removed from the system and its memory freed, the _unload() routine is called. The routine informs WSIO2 that it is unloading and releases any system resources acquired by the driver.
- _halt() Routine: The halt() routine is used to perform any special operations that may be required of a particular device. These may include suspending or cancelling any queues on a device, or halting any on-board processing that may be occurring on the device.

Sample Wrapper. The following code represents a sample wrapper for a typical WSIO2 driver with the prefix xxx. The same wrapper would be used for both interface and target drivers. Procedures to handle the differences between target and interface driver initialization and removal are contained in the driver's xxx_init() and xxx_remove() routines. These routines are called as part of the normal WSIO2 initialization.

```
extern void *xxx_conf_data; /* Defined by configuration tools */
extern struct mod_operations gio_mod_drv_ops;
/* Defined in kernel */

static int xxx_load(), xxx_unload();
static void xxxhalt();

static struct mod_type_data xxx_drv_link = {
"xxx - Loadable xxx Driver", NULL
};

static struct modlink xxx_mod_link[] = {
{ &mod_drv_ops, &xxx_drv_link },
{ NULL, NULL }
};

struct modwrapper xxx_wrapper = {
MODREV, xxx_load, xxx_unload, xxx_halt,
&xxx_conf_data,
xxx_mod_link
};

static int
xxx_load()
{
IOSTATUS err;
DRVR_OBJECT_INFO drvr_obj_info;

drvr_obj_info.name = "xxx";
drvr_obj_info.class = "scsi";
err = wsio2_load_driver( WSIO_REVISION,
IO_DYNAMIC | XXX_FLAGS,
&drvr_obj_info,
&xxx_init,
&xxx_remove);
if(err != IO_OKAY) {
printf("xxx_load: Error loading driver(%d)\n", err);
return(err);
}
xxx_start();
};

static int
xxx_unload()
{
IOSTATUS err;
err = wsio2_unload_driver(drvr_object);
if(err != IO_OKAY) {
printf(xxx_unload: Error unloading driver(%d)\n", err);
return(err);
}
/* Free any driver resources here */
}
```

WSIO2 Device Driver Operations

The mod_operations structure for WSIO2 drivers is defined as follows:

```
struct mod_operations gio_mod_drv_ops = {
NULL, /* Install device driver */
NULL, /* Remove device driver */
mod_gio_drvinfo; /* Device driver status */
};
```

gio_mod_drvinfo() Routine. This routine is called by the modstat(2) system call to obtain information specific to the driver. It is passed a pointer to the driver's mod_type_data structure. The following information will be returned:

- WSIO2 loadable module type - MOD_TY_WSIO2_INTFC or MOD_TY_WSIO2_CLASS.
- Driver class (i.e. tape, disk, ext_bus, etc.).
- The modules block and/or character major numbers.

WSIO2 Wrapper Macro

A macro is available to aid in the generation of wrapper information for loadable modules. For WSIO2 device drivers this macro takes the form:

```
MOD_WSIO2_DRV_WRAPPER(prefix, load, unload, halt, name);
```

For the above macro, prefix is the module's prefix normally prepended to the module's entry points (i.e. stape, sdisk, etc.), load, unload, and halt are the names of the module's load, unload and halt routines, and name is the character string name describing the module.

Special considerations need to be made when defining the wrappers prefix_load() and prefix_unload() routines:

- Any resources allocated by the module must be released in _unload() after the call to wsio_unload_driver().
- Any outstanding timeout() requests must be canceled before the module is unloaded.

WSIO2 Device Driver Autoload Procedure

A device driver is automatically loaded on first open of any of its configured devices, and is considered unloadable on last close of all its configured devices (this may not be the case if other modules depend on the driver). When a driver is registered, the address of a special autoload open routine and an initialized drv_info structure is put into the corresponding switch table entries for the driver. For block and character devices, the address is placed directly into the d_open field of the switch table.

Both WSIO2 block and character device drivers have their own autoload routine. The only difference is which switch table the autoload information comes from. For WSIO2 drivers the autoload routines are:

- gio_mod_bdev_open() For block devices.
- gio_mod_cdev_open() For character devices.

The algorithm for both routines is as follows:

- Call modld() to load the module whose name is given by the name field in the drv_info structure pointed to by the switch table entry.
- If the load fails return ENOLOAD.
- On successful load, increment the module's reference count and release the module lock (the module is locked when modld() returns successfully).

  At this point we know the fields of the switch table have been repopulated with values for the newly loaded device.
- Call the driver's open routine through the newly populated d_open field of the switch table.
- If the open fails:
- Decrement the module's reference count.
- Return the error code from the open.
- Else, return success.

While the driver is loaded, subsequent open requests call the driver's open routine directly. When the driver is unloaded, the address of the autoload open routine is placed back into the switch table.

The reference count for the driver needs to be maintained in order to determine when the driver is considered unloadable. This requires the addition of code to the spec_open() and spec_close() routines. This level of interface was chosen because we are assured that each spec_open() will have a corresponding spec_close(), enabling us to maintain an accurate reference count for the driver.

On every successful spec_open() for a device whose driver is loadable, the driver's reference count is incremented. When the driver is closed spec_close() checks if the driver for the given device is loadable and, if so, decrements its reference count.

WSIO2 Driver Load/Unload Sequence

The following sections explain the algorithms for autoloading and unloading a WSIO2 device driver.

Autoloading a WSIO2 Driver. The following is the algorithm for autoloading a WSIO2 target driver:

- mod_xdev_open() - Called from pointer in d_open field of the device switch table.
- modld() - Called to load, relocate and register the module.
- mod_obj_load() - Actual module load and relocation done here.
- prefix_load() - The driver's load routine is called to begin driver initialization.
- wsio_load_driver() - Called to register the driver with WSIO2.
- Obtain module type specific data. This includes the module's major numbers and flags.
- prefix_init() - Initialize WSIO2 specific data.
- set_entry_points() - Call the various WSIO2 routines that set the driver's entry points.
- Save a pointer to the driver's object structure for unloading later.
- install_driver() - GIO routine called to install driver entry points into the device switch table.
- register_driver_object() - Registers the driver object structure with WSIO2.
- prefix_start() - Initialize driver specific data structures.
- prefix_open() - Call the driver's open routine through the device switch table.

Unloading a WSIO2 Driver. The following algorithm is used for unloading a WSIO2 target driver:

- modunld() - Called to begin unload sequence.
- prefix_unload() - The driver's unload routine.
- wsio2_unload_driver() - Begin unregistering the driver with WSIO2.

- gio_mod_drv_remove() - Reinitialize the device switch table entry with the DLM autoload pointers.
- unregister_driver_object() - Remove the driver object from the WSIO2 list.
- Free memory allocated for the driver object.
- Free any resources allocated by the driver.

Loadable STREAMS Device Drivers

This section describes how STREAMS device drivers may make use of the loadable module feature.

STREAMS Device Driver Registration

Registering the correspondence between a cdevsw entry and the STREAMS driver's name is accomplished through the mod_sdev_reg() routine.This routine is invoked by modadm() for module type MOD_TY_SDEV. As part of the registration, a slot in cdevsw is filled in with the autoloading open stub. If a demand load is done, these entries are overwritten with the actual entry points for a STREAMS drivers. If an open is attempted, the stub routine mod_sdev_open() is called. This routine is responsible for loading the actual code for the driver and initializing the cdevsw entries to make the driver usable. As a byproduct, the driver open routine is called, which makes the autoload transparent to the calling process.

STREAMS Device Driver Unregistration

Drivers that have been unloaded my be unregisterd. Unregistration removes any references to the driver from cdevsw and frees any allocated resources acquired during registration. Drivers being unregistered must first be unloaded or unregistration will fail. Unregistration is accomplished through the mod_sdev_unreg() routine. This routine is invoked by modadm() for module type MOD_TY_SDEV.

As part of the registration, a slot in fmodsw is filled in with the autoloading open stub. If a demand load is done, these entries are overwritten with the actual entry points for the module. If an open is attempted, the stub routine mod_smod_open() is called. This routine is responsible for loading the actual code for the module and initializing the fmodsw entries to make the module usable. As a byproduct, the module open routine is called, which makes the autoload transparent to the calling process. Note that modules are opened by pushing them on an existing stream via the I_PUSH ioctl.

Wrapper Routines

As described in the Dynamically Loadable Kernel Modules section, STREAMS modules and drivers may define _load(), _unload(), and _halt() routines to perform any driver/module specific initialization and cleanup.

Wrapper Routines. As described in the Dynamically Loadable Kernel Modules section, device drivers may define _load(), _unload(), and _halt() routines to perform any driver specific initialization and cleanup.

- _load() Routine After the module/driver has been loaded and relocated, the _load() routine is called. This routine will frequently have nothing special to do since the linkages with the kernel data structures are provided by the type-specific code. Only module/driver specific initialization must be done. This routine may be NULL in the wrapper.
- _unload() Routine Just before the module/driver is removed from the system and its memory freed, the _unload() routine is called. This routine will frequently have nothing special to do since the linkages with the kernel data structure will be broken by the type-specific code. Only module/driver specific cleanup must be done. This routine may be NULL in the wrapper.
- halt() Routine The halt() routine is used to perform any special operations that may be required.

Sample Wrapper. The following code represents a sample wrapper for a typical STREAMS driver with the prefix xxx.

STREAMS Device Driver Operations

For STREAMS drivers, the following routines accomplish the data structure linkages:

* Install routine: mod_sdevinstall()

* Remove routine: mod_sdevremove()

* Info routine: mod_sdevinfo()

STREAMS Wrapper Macro

A macro is available to aid in the generation of wrapper information for loadable modules. For STRAMS device drivers this macro takes the form:

```
MOD_SDEV_WRAPPER(prefix, load, unload, halt, name, info);
```

For the above macro, prefix is the driver's prefix normally prepended to the modules entry points, load, unload, and halt are the names of the driver's load, unload, and halt routines, name is the character string name describing the driver, and info is the address of the streams_info_t structure associated with the driver.

STREAMS Device Driver Autoload Procedure

A device driver is automatically loaded on first open of any of its configured devices, and is considered unloadable on last close of all its configured devices (this may not be the case if other modules depend on the driver). When a driver is registered, the address of a special autoload open routine and an initialized drv_info structure is put into the corresponding switch table entries for the driver. For block and character devices, the address is placed directly into the d_open field of the switch table.

STREAMS Driver Load/Unload Sequence

The following sections explain the algorithms for autoloading and unloading a STREAMS device driver.

Autoloading a STREAMS Driver. The following is the algorithm for autoloading a STREAMS device driver:

- mod_xdev_open() - Called from pointer in d_open field of the device switch table.
- modld() - Called to load, relocate and register the module.
- mod_obj_load() - Actual module load and relocation done here.
- prefix_load() - The driver's load routine is called to begin driver initialization.
- streams_load_driver() - Called to register the driver with WSIO2.
- Obtain module type specific data. This includes the module's major numbers and flags.
- prefix_init() - Initialize STREAMS specific data.

- set_entry_points() - Call the various STREAMS routines that set the driver's entry points.
- Save a pointer to the driver's object structure for unloading later.
- install_driver() - GIO routine called to install driver entry points into the device switch table.
- register_driver_object() - Registers the driver object structure with STREAMS.
- prefix_start() - Initialize driver specific data structures.
- prefix_open() - Call the driver's open routine through the device switch table.

Unloading a STREAMS Driver. The following algorithm is used for unloading a STREAMS target driver:

- modunld() - Called to begin unload sequence.
- prefix_unload() - The driver's unload routine.
- streams_unload_driver() - Begin unregistering the driver with STREAMS.
- gio_mod_drv_remove() - Reinitialize the device switch table entry with the DLM autoload pointers.
- unregister_driver_object() - Remove the driver object from the STREAMS list.
- Free memory allocated for the driver object.
- Free any resources allocated by the driver.

Loadable Miscellaneous Device Drivers

This section describes how miscellaneous device drivers may make use of the loadable module feature.

Miscellaneous Device Driver Registration

Registering the correspondence between a cdevsw entry and the miscellaneous driver's name is accomplished through the mod_sdev_reg() routine.This routine is invoked by modadm() for module type MOD_TY_SDEV. As part of the registration, a slot in cdevsw is filled in with the autoloading open stub. If a demand load is done, these entries are overwritten with the actual entry points for a STREAMS drivers. If an open is attempted, the stub routine mod_sdev_open() is called. This routine is responsible for loading the actual code for the driver and initializing the cdevsw entries to make the driver usable. As a byproduct, the driver open routine is called, which makes the autoload transparent to the calling process.

Miscellaneous Device Driver Unregistration

Drivers that have been unloaded my be unregisterd. Unregistration removes any references to the driver from cdevsw and frees any allocated resources acquired during registration. Drivers being unregistered must first be unloaded or unregistration will fail. Unregistration is accomplished through the mod_sdev_unreg() routine. This routine is invoked by modadm() for module type MOD_TY_SDEV.

As part of the registration, a slot in fmodsw is filled in with the autoloading open stub. If a demand load is done, these entries are overwritten with the actual entry points for the module. If an open is attempted, the stub routine mod_smod_open() is called. This routine is responsible for loading the actual code for the module and initializing the fmodsw entries to make the module usable. As a byproduct, the module open routine is called, which makes the autoload transparent to the calling process. Note that modules are opened by pushing them on an existing stream via the I_PUSH ioctl.

Wrapper Routines

As described in the Dynamically Loadable Kernel Modules section, STREAMS modules and drivers may define _load(), _unload(), and _halt() routines to perform any driver/module specific initialization and cleanup.

Wrapper Routines. As described in the Dynamically Loadable Kernel Modules section, device drivers may define _load(), _unload(), and _halt() routines to perform any driver specific initialization and cleanup.

- _load() Routine After the module/driver has been loaded and relocated, the _load() routine is called. This routine will frequently have nothing special to do since the linkages with the kernel data structures are provided by the type-specific code. Only module/driver specific initialization must be done. This routine may be NULL in the wrapper.

- _unload() Routine Just before the module/driver is removed from the system and its memory freed, the _unload() routine is called. This routine will frequently have nothing special to do since the linkages with the kernel data structure will be broken by the type-specific code. Only module/driver specific cleanup must be done. This routine may be NULL in the wrapper.
- halt() Routine The halt() routine is used to perform any special operations that may be required.

Sample Wrapper. The following code represents a sample wrapper for a typical miscellaneous driver with the prefix xxx.

```
extern void *xxx_conf_data; /* defined by configuration tools */

extern struct mod_operations mod_misc_ops; /* defined by kernel */

static int32_t xxx_load(), xxx_unload();

static struct mod_type_data xxx_misc_link = { "Loadable xxx
Module", NULL };

static struct modlink xxx_mod_link[] = {
{ &mod_misc_ops, &xxx_misc_link },
{ NULL, NULL }
};

struct modwrapper xxx_wrapper = {
MODREV, xxx_load, xxx_unload, NULL,
&xxx_conf_data,
xxx_mod_link
};

static int32_t xxx_load(void)
{
/*
* Allocate any needed memory here and do other module specific
* initialization.
*/
return(0);
}

static int32_t xxx_unload(void)
{
/*
* Free any memory allocated in xxx_unload and elsewhere in the
* module and do any other necessary module specific cleanup.
*/
return(0);
}
```

Miscellaneous Device Driver Operations

The mod_operations structure for the miscellaneous type is initialized as follows:

```
struct mod_operations mod_misc_ops = {
mod_misc_donothing,
mod_misc_donothing,
mod_misc_info
};
```

Since there are no system tables or other data structures to be updated on installation or removal of these modules, the installation and removal routine does nothing. The mod_misc_info() routine fills in the space pointed to by the type argument with MOD_TY_MISC.

Miscellaneous Wrapper Macro

A macro is available to aid in the generation of wrapper information for loadable modules. For miscellaneous device drivers this macro takes the form:

```
MOD_MISC_WRAPPER(prefix, load, unload, halt, name, info);
```

For the above macro, prefix is the driver's prefix normally prepended to the modules entry points, load, unload, and halt are the names of the driver's load, unload, and halt routines, name is the character string name describing the driver, and info is the address of the misc_info_t structure associated with the driver.

Miscellaneous Device Driver Autoload Procedure

Miscellaneous modules are autoloaded via the stubs mechanism described below. Once they are loaded, they are not unloadable.

Loadable Module Installation Procedures

This section describes the process for installing a loadable module.

Installing a Loadable Module’s Components

Use this procedure to install a loadable module’s components using the **`kminstall`**command.

`kminstall` will expect a readable mod.o, master and system file in the module components’ directory. It will create required directories if they do not exist. If module_name already exists on the system, **`kminstall`** will print a message and fail.

1. Change directories to the directory containing the module components.

2. Execute **`kminstall`**, which copies the module components into the configuration–related subdirectories.

   ```
   /usr/sbin/kminstall -a module_name
   ```

Preparing a Loadable Module

Use this procedure to prepare a loadable module for configuration using the **`kmsystem`** command.

`kmsystem` will set the configuration and loadable flags so that the module can be configured and loaded.

- Execute **`kmsystem`**, which sets the module's configuration and loadable flags, as follows:

  ```
  /usr/sbin/kmsystem -c Y -l Y module_name
  ```

Preparing a Loadable Module as a Static Module

Use this procedure to prepare a loadable module as a static module using the **`kmsystem`** command.

`kmsystem` will set the configuration and loadable flags so that the module can be configured.

- Execute **`kmsystem`**, which sets the module's configuration and loadable flags, as follows:

  ```
  /usr/sbin/kmsystem -c Y -l N module_name
  ```

Configuring a Loadable Module for Loading

Use this procedure to configure a loadable module using the **`config`** command.

`config` will read the running kernel's system description file, system description files for kernel modules, and the master kernel configuration information table, checks the interface functions or symbols used by the modules, and creates the temporary build files.

- Execute **`config`**, which sets the module's configuration and loadable flags, as follows:

  ```
  /usr/sbin/config -M module_name
  ```

Configuring a Kernel That Statically Includes Loadable Modules

Use this procedure to configure a module that is statically linked as a loadable module using **`kmsystem`**.

NOTE

While any loadable module may be configured to be statically linked to the kernel, not all static modules may be dynamically loadable. Refer to the documentation that accompanied the module for information.

To use **`kmsystem`** to reconfigure the kernel:

1. Log in as superuser on the machine for which a new kernel is being generated. You can log in remotely from another location by using the `/usr/bin/rlogin` command.
2. Execute **`kmsystem`**, which sets the module's configuration and loadable flags, as follows:

   ```
   /usr/sbin/kmsystem -c Y -l N module_name
   ```

Registering a Loadable Module

Use this procedure to register a loadable module using the **`kmupdate`** command.

`kmupdate` will update the specified kernel modules. If the specified module is already loaded, **`kmupdate`** tries to unload it. If the module was unloaded successfully, or if it was not already loaded, **`kmupdate`** will then check if the module is registered and if so, unregister it. **`kmupdate`** will then overlay the existing loadable image of the module with the newly generated image, register the module with the latest registry information and perform module specific initialization, if required. If the module was originally loaded at the beginning of the process, **`kmupdate`** reloads the module before exiting.

- Execute **`kmupdate`** as follows:

  ```
  /usr/sbin/kmupdate -M module_name
  ```

Loading a Loadable Module

Use this procedure to demand load a loadable module using the **`kmadmin`** command.

`kmadmin` will demand load the specified kernel modules. The load operation performs all of the tasks associated with link editing the module to the kernel and making the module accessible to the system. If the module is dependent on other modules and these modules are not currently loaded, kmadmin will automatically load the dependent modules.

- Execute **kmadmin** as follows:

  ```
  /usr/sbin/kmadmin -L module_name
  ```

Requesting an Updated Kernel at the Next System Reboot

Use this procedure to update a loadable module's image at the next system reboot using the **kmupdate** command. Use this procedure when you have configured a loadable module as statically linked.

The module will be updated asynchronously at shutdown. When the system shuts down, the modules loadable image is updated. The module is registered when the system is restarted.

- Execute **kmupdate** as follows:

  ```
  /usr/sbin/kmupdate /stand/build/vmunix_test
  ```

Unloading a Loadable Module

Use this procedure to unload a loadable module using the **kmadmin** command. You have the option of unloading the module by its name or its ID number.

kmadmin will perform all tasks associated with disconnecting the module from the kernel and releasing any memory acquired by the module.

- To unload a loadable module by name, execute **kmadmin** as follows:

  ```
  /usr/sbin/kmadmin -U module_name
  ```

- To unload a loadable module by ID number, execute **kmadmin** as follows:

  ```
  /usr/sbin/kmadmin -u module_id
  ```

Tuning Loadable Modules

Use this procedure to query, set, or reset system parameters using the **kmtune** command.

kmtune reads the master files and the system description files of the kernel and kernel modules.

- To query the value of a specific system parameter, execute **`kmtune`** as follows:

  /usr/sbin/kmtune -q system_parameter_name

- To set the value of a specific system parameter, execute **`kmtune`** as follows:

  /usr/sbin/kmtune -s system_parameter_name=value

- To reset the value of a system parameter, execute **`kmtune`** as follows:

  /usr/sbin/kmtune -r system_parameter_name

Removing a Loadable Module's Components

Use this procedure to remove a module's components using the **`kminstall`** command.

If the module is currently loaded, **`kminstall`** will unload the module. **`kminstall`** will then unregister the module and delete the appropriate files and directories.

- Execute **`kminstall`** as follows:

  /usr/sbin/kminstall -d module_name

Updating a Loadable Module's Components

Use this procedure to update and configure a loadable module's components using the **`kminstall`** command.

`kminstall` will copy the appropriate files to the appropriate directories. The values of the tunable parameters and the $LOADABLE and $CONFIGURATION flags are taken from the running system and written to the to the new system file for the module.

`kminstall` will expect a readable mod.o, master and system file in the module components' directory. It will create required directories if they do not exist. If module_name already exists on the system, **`kminstall`** will print a message and fail.

- Execute **`kminstall`** as follows:

  /usr/sbin/kminstall -u module_name

Updating a Loadable Module's Image Without a Reboot

Use this procedure to update a loadable module's image without rebooting using the **`kmupdate`** command.

`kmupdate` will update the specified kernel modules. If the specified module is already loaded, **`kmupdate`** tries to unload it. If the module was unloaded successfully, or if it was not already loaded, **`kmupdate`** will then check if the module is registered and if so, unregister it. **`kmupdate`** will then overlay the existing loadable image of the module with the newly generated image, register the module with the latest registry information and perform module specific initialization, if required. If the module was originally loaded at the beginning of the process, **`kmupdate`** reloads the module before exiting.

- Execute **`kmupdate`** as follows:

  /usr/sbin/kmupdate -M module_name

Determining Which Modules Are Currently Loaded

Use this procedure to print the full status of all the modules that are currently loaded using the **`kmadmin`** command.

NOTE

To list statically linked loadable modules, use the **`kmadmin -k`** command.

- Execute **`kmadmin`** as follows:

  /usr/sbin/kmadmin -S

Obtaining Information About a Loaded Module

Use this procedure to print the status of all the modules that are currently loaded using the **`kmadmin`** command. You have the option of displaying the module by its name or ID number.

- To display a module's status by name, execute **`kmadmin`** as follows:

  /usr/sbin/kmadmin -q module_name

- To display a module's status by ID, execute **`kmadmin`** as follows:

  /usr/sbin/kmadmin -Q module_id