Chapter 19

STREAMS Drivers

The IRIX implementation of STREAMS drivers is intended to be compatible with the multiprocessor implementation of STREAMS in UNIX version SVR4.2.

STREAMS programming in SVR4.2 is documented in *STREAMS Modules and Drivers, UNIX SVR4.2*. That book contains detailed discussion and many examples of STREAMS programming.

References in this chapter to *STREAMS Modules and Drivers* are to the edition copyright 1992 by UNIX System Laborartories, published by UNIX Press/Prentice-Hall, and bearing ISBN 0-13-066879. If you are using an earlier edition, you should upgrade it. If you have a later edition, you may have to interpret references carefully.

This chapter contains the following major sections:

- "Driver Exported Names" on page 542 summarizes the public names and functions that a STREAMS driver must export.
- "Building and Debugging" on page 546 describes the ways that building a STREAMS driver are like and unlike other kernel-level drivers.
- "Special Considerations for Multiprocessing" on page 547 describes the methods you must use to work with the multi-threaded STREAMS monitor.
- "Special Considerations for IRIX" on page 549 details the points at which IRIX differs from the SVR4 STREAMS environment.
- "Summary of Standard STREAMS Functions" on page 554 lists the available kernel functions used by STREAMS drivers.
- "STREAMS Modules for X Input Devices" on page 556 describes the use of configuration files for special input devices used by the X display manager.

Driver Exported Names

A STREAMS driver or module must define certain public names for use by *lboot*, as described in “Summary of Driver Structure” on page 136. Only one of these names, the info structure, is unique to a STREAMS driver or module; all the others are also defined by kernel-level device drivers.

The public names all begin with a prefix (see “Driver Name Prefix” on page 136); the same prefix is specified in the configuration file (see “Describing the Driver in /var/sysgen/master.d” on page 233).

Streamtab Structure

A STREAMS driver or module must provide a global *streamtab* structure containing pointers to the *qinit* structures for its read and write queues. These structures in turn point to required *module_info* structures. The name of the streamtab is *pfx***info**.

Driver Flag Constant

A STREAMS driver or module should provide a driver flag constant containing either 0 or the flag D_MP. (See “Driver Flag Constant” on page 140 and “Flag D_MP” on page 141). The name of the constant is *pfx***devflag**.

Note: A driver or module that does not export *pfx***devflag** is assumed to use SVR3 calling conventions at its *pfx***open()** and *pfx***close()** entry points. However, this support will be withdrawn in a release of IRIX in the very near term. If you are porting a STREAMS driver or module to IRIX you are urged to make sure it uses SVR4 conventions and exports a *pfx***devflag** containing at least 0.

Initialization Entry Points

A STREAMS driver or module can define an entry point *pfx***init()**, or an entry point *pfx***start()**, or both. These entry points will be called during boot if the driver or module is included in the kernel, or when the driver or module is loaded if it is loadable. The operation of these entry points is the same as for device drivers (see "Initialization Entry Points" on page 143).

Many STREAMS drivers perform all initialization at open time, and have no *pfx***init()** or *pfx***start()** entry points. Many STREAMS modules perform initialization when they receive the I_PUSH ioctl message.

Entry Point open()

A STREAMS driver (but not module) must export a *pfx***open()** entry point. The argument list for a STREAMS driver's open differs from that of a device driver. The prototype for a STREAMS *pfx***open()** entry point is:

```
int
pfxopen(queue_t *q, dev_t *devp, int oflag, int sflag, cred_t *crp);
```

The argument values are

**q*	Pointer to the *queue* structure being opened.
**devp*	Pointer to a *dev_t* value from which you can extract both the major and minor device numbers.
oflag	Flag bits specifying user mode options on the **open()** call.
sflag	Flag bits specifying the type of STREAM open: driver, module or clone.
**crp*	Pointer to a *cred_t* object—an opaque structure for use in authentication.

The *pfx***open()** entry point is a public name. In addition a pointer to it must be defined in the *qinit* structure for the read queue.

Entry Point close()

A STREAMS driver (but not module) must export a *pfx***close()** entry point. The argument list for a STREAMS driver's close differs from that of a device driver. The prototype for a STREAMS *pfx***close()** entry point is:

```
int
pfxclose(queue_t *q, int oflag, cred_t *crp);
```

The argument values are the same as passed to *pfx***open()**. The *pfx***close()** entry point is a public name. In addition a pointer to it must be defined in the *qinit* structure for the read queue.

Put Functions wput() and rput()

Every STREAMS driver and module must define a **put()** function to handle messages as they are delivered to a queue.

The prototype of a **put()** function is as follows:

```
int
name(queue_t *q, mblk_t *mp);
```

Because the **put()** function for a given queue is addressed from the associated *qinit* structure, there is no requirement that the **put()** function be a public name, and no requirement that it begin with the prefix string. The **put()** function for the write queue, which handles messages moving "downstream" from the user process toward the driver, is conventionally called the **wput()** function. All write queues need a **wput()** function.

The **put()** function for the read queue, which handles messages moving "upstream" from the driver toward the user process, is conventionally called the **rput()** function. In some cases the **rput()** function is not required, for example in a driver where all upstream messages are generated by an interrupt handler.

Typically, a **put()** function decides what to do by switching on the message type value from *mp->b_datap->db_type*. A **put** routine must do at least one of the following:

- Process the message, if immediate processing is required, consuming the message or transforming it.
- Pass the original or processed message to the next component in the stream by calling the **putnext()** function (see the putnext(D3) reference page).
- Queue the message for deferred processing by the service routine with the **putq()** function (see the putq(D3) reference page).

When all processing is deferred to the service function, the address of the kernel function **putq()** can be given as a queue's **put()** function.

In a multiprocessor, a **put()** function can be called concurrently with user-level code, and concurrently with another **put()** function for the same or a different queue. A service function for the same or different queue can also be executing concurrently.

Service Functions rsrv() and wsrv()

When a STREAMS driver defers message processing by setting the kernel function **putq()** address as the driver's **put()** function, the queue must also define a service function **srv()**.

Because the **srv()** function for a given queue is addressed from the associated *qinit* structure, there is no requirement that the **srv()** function be a public name, and no requirement that it begin with the prefix string.

The prototype of a **svr()** function is as follows:

```
int
name(queue_t *q);
```

The **srv()** function for the write queue, which handles messages moving "downstream" from the user process toward the driver, is conventionally called the **wsrv()** function. The **srv()** function for the read queue, which handles messages moving "upstream" from the driver toward the user process, is conventionally called the **rsrv()** function.

An **srv()** function is called by the STREAMS monitor to deal with queued messages. It is called at a time chosen by the monitor, not necessarily related to any call to the **put()** function for the same queue. In a multiprocessor, only one instance of **srv()** is called per queue at any time. However, one or more instances of the **put()** function could execute concurrently with the **srv()** function—so any data that is used in common by **put()** and **srv()** must be protected with a lock (see “Waiting and Mutual Exclusion” on page 204). User-level code can also execute concurrently with a service function.

The service function is expected to dispose of all queued messages through one of the following actions:

- Consuming and freeing the message.
- Passing the message on to the following queue using **putnext()** (see the putnext(D3) reference page).
- Replacing the message on the same queue using **putbq()** for processing later (see the putbq(D3) reference page).

The service function implements flow control (which the **put()** function cannot do). Before applying **putnext()**, the service function calls a flow control function such as **canputnext()** to find out if the following queue can accept a message. If the following queue cannot accept a message, the service function replaces the message with **putbq()** and exits.

A STREAMS module or driver that is not multiprocessor-aware (lacks D_MP in its *pfx***devflags**) uses one set of functions for flow control (see the canput(D3) and bcanputnext(D3) reference pages), while one that is multiprocessor-aware uses a different set (see canputnext(D3) and bcanputnext(D3)).

Building and Debugging

A STREAMS driver or module is a kernel module and is compiled using the same compiler options as any driver (see “Compiling and Linking” on page 228).

You configure each STREAMS driver or module as part of the IRIX kernel by:

- Placing the executable module in */var/sysgen/boot*
- Writing a descriptive file and placing it in */var/sysgen/master.d* (see “Describing the Driver in /var/sysgen/master.d” on page 233)

- Placing a USE or INCLUDE line in */var/sysgen/system* (see “Configuring a Kernel” on page 236)

When a STREAMS driver or module is loadable, you specify the appropriate options in the descriptive file (see “Master File for Loadable Drivers” on page 238). You can configure a STREAMS driver or module to be autoregisterd and loaded automatically (see “Registration” on page 239). Alternatively, you can require a STREAMS driver or module to be loaded manually using the *ml* command (see “Loading” on page 239).

When you have configured a debugging kernel (see “Preparing the System for Debugging” on page 243), the symbols of a STREAMS driver or module are available for display. You can set breakpoints using *symmon* (see “Using symmon” on page 252). You can display symbols using *symmon* or *idbg* (see “Using idbg” on page 262). In particular, *idbg* has built-in support for displaying the contents of structures used by a STREAMS module or driver (see “Commands to Display STREAMS Structures” on page 268).

Special Considerations for Multiprocessing

In IRIX releases prior to 6.2, the STREAMS monitor was single-threaded, so that only one **put()** or **srv()** function in the entire system could execute at any time. That one **put()** or **srv()** function might execute concurrently with user-level code, but no two STREAMS functions could execute concurrently.

Beginning with IRIX 6.2, the STREAMS monitor is multi-threaded. Depending on the version of IRIX and on the number of CPUs in the system, the following functions can run concurrently in any combination: one **srv()** function for each queue; any number of **put()** functions for each queue; and one or more user processes. For general discussion of the consequences, see “Planning for Multiprocessor Use” on page 171.

In the multithreaded monitor, when a module or driver calls **putq()** or **qenable()**, the service function for the enabled queue can begin to execute at any time. It can begin execution before **putq()** or **qenable()** call has returned, and can run concurrently with the module or driver that enabled the queue.

The STREAMS monitor runs concurrently with interrupt handling. For this reason, the interrupt handler of a STREAMS driver must take an extra step before it performns any STREAMS-related processing such as **allocb()**, **putq()**, or **qenable()**. The IRIX-unique functions provided for this purpose are summarized in Table 19-1.

Table 19-1 Multiprocessing STREAMS Functions

Name	Can Sleep?	Summary
streams_interrupt(D3)	N	Synchronize interrupt-level function with STREAMS mechanism.
STREAMS_TIMEOUT(D3)	N	Synchronize timeout with STREAMS mechanism.

Suppose that the interrupt handler of a STREAMS driver needs to add a message to the read queue with **putq()**. It cannot simply call that function, since the STREAMS monitor might be using the queue at the same time in another CPU. The driver must define a function in which the **putq()** call is written. The name of this function and the pointer to the queue are passed to **streams_interrupt()**. As soon as possible, **streams_interrupt()** gets control of the queue and executes the passed function.

A callback function scheduled using **itimeout()** and similar functions (see "Waiting for Time to Pass" on page 214) must also be synchronized with the STREAMS monitor.

Suppose that a STREAMS driver or module needs to schedule a function to execute at a later time. (In a nonSTREAMS driver the function would be scheduled with **itimeout()**.) In the time-delayed function is a call to **qenable()**. That call cannot be executed freely whenever the interval expires, because the state of the STREAMS monitor is not known at that time.

The STREAMS_TIMEOUT macros provide a solution. Like **itimeout()**, it schedules a function to be executed at a later time. However, it defers calling the function until the function is synchronized with the STREAMS monitor, so that it can execute calls such as **qenable()**.

Special Considerations for IRIX

While IRIX is largely compatible with UNIX SVR4.2, there are points of difference in the implementation of IRIX that have to be reflected in the design of a STREAMS driver or module. This topic lists points at which the contents of *STREAMS Modules and Drivers, UNIX SVR4.2* is not a correct description of IRIX and STREAMS use within IRIX.

Extension of Poll and Select

Under IRIX, the **poll()** system function is not limited to testing STREAMS, but can be applied to file descriptors of all types (see the poll(2) and select(2) reference pages). In addition the **select()** function can be applied to STREAMS file descriptors. You may want to note this under the heading "STREAMS System Calls" in Chapter 2 of *STREAMS Modules and Drivers, UNIX SVR4.2*.

Support for Pipes

IRIX supports two kinds of pipes with different semantics, as described in the pipe(2) reference page. The default type of pipe is compatible with UNIX SVR3, and does not conform to the description in Chapter 2 of *STREAMS Modules and Drivers, UNIX SVR4.2* under the heading "Creating a STREAMS-based Pipe."

The SVR4 pipe semantics are enabled on a system-wide basis by using the *systune* command to set the tuning parameter *svr3pipe* to 0. First test the configuration as shown in Example 19-1.

Example 19-1 Testing Pipe Configuration

```
# systune | grep svr3pipe
       svr3pipe = 1 (0x1)
```

Service Scheduling

At two points in *STREAMS Modules and Drivers, UNIX SVR4.2* (Under "Service Procedure" in Chapter 4 and under "Message Processing" in Chapter 5), the book explicitly says that in a uniprocessor, enabled service functions are always executed before returning to user-level processing. This promise is not supported by IRIX. In both uniprocessors and multiprocessors, user-level processes can potentially execute after a service function is enabled and before it executes.

Supplied STREAMS Modules

STREAMS Modules and Drivers, UNIX SVR4.2, Chapter 4, refers to some example STREAMS drivers named CHARPROC, CANONPROC, and ASCEBC. These examples are not supplied with IRIX.

The following STREAMS-based modules are supplied with IRIX. You can read their reference pages in volume 7:

alp(7)	Algorithm pool management module.
clone(7)	Clone-open driver; see "Support for CLONE Drivers" on page 552.
connld(7)	Line discipline for unique stream connections.
kbd(7)	Generalized string translation module.
log(7)	Interface to STREAMS error logging and event tracing.
sad(7)	STREAMS Administrative Driver.
streamio(7)	STREAMS ioctl commands.
timod(7)	Transport Interface cooperating STREAMS module.
tirdwr(7)	Transport Interface read/write interface STREAMS module.
tsd(7)	TELNET server protocol STREAMS device.

No #idefs

Chapter 4 of *STREAMS Modules and Drivers, UNIX SVR4.2* refers in a note to the use of the #idef and a transition period for SVR3-compatible drivers. None of this material is relevant to IRIX. IRIX is SVR4-compatible, with no special provision for SVR3 drivers.

Different I/O Hardware Model

Chapter 5 of *STREAMS Modules and Drivers, UNIX SVR4.2* discusses the use of memory-mapped hardware and of Dual-Access RAM (DARAM). None of these considerations are relevant in a MIPS processor. The MIPS I/O model is discussed in Chapter 1, “Physical and Virtual Memory.”

Different Network Model

Chapter 10 of *STREAMS Modules and Drivers, UNIX SVR4.2* describes the TPI interface model. This model is supported in IRIX. When an application uses the TLI library functions such as **t_open()**, the library uses IRIX-provided TPI STREAMS modules which implement the protocol described in chapter 10.

Chapter 11 of *STREAMS Modules and Drivers, UNIX SVR4.2* describes the Data Link Provider Interface (DLPI) as implemented using STREAMS facilities.

The IRIX networking support is not STREAMS-based, but rather is based on BSD *ifnet* architecture. This is discussed in Chapter 16, “Network Device Drivers.” The IRIX network support includes DLPI support as an add-on feature to the *ifnet* driver interface. If you are porting a network device driver to IRIX, it is better to convert it to the *ifnet* interface. You can install a DLPI-based network device driver, but only other STREAMS modules could use it—there would be no connection to the rest of the IRIX networking system.

Support for CLONE Drivers

STREAMS Modules and Drivers, UNIX SVR4.2 discusses CLONE drivers; that is, STREAMS drivers that generate a new minor device number for each open. Refer to Chapter 3, "The CLONE Driver," and to Chapter 8, "Cloning." Clone opens and the clone driver are implemented under IRIX; this section clarifies the discussion in the SVR4 manual.

The essence of cloned access to a STREAMS driver is that the user process is indifferent to the minor device number, and simply wants to open a stream from this driver. A cloned stream is created using the following steps:

1. Recognize that the process calling **open()** is indifferent to the minor device number and simply wants cloned access.
2. Choose an unused minor device number from the set of minor numbers the driver supports.
3. Construct a new device number *dev_t* value based on the chosen minor number, and assign it to the argument passed to *pfx***open()**.

Using the CLONE Driver

The IRIX-supplied clone driver automates some of these steps for your driver. In order to use it, prepare a device special file with these characteristics:

- A device name that is related to the actual device name
- The major device number (10 decimal) that specifies the clone driver
- A minor device number equal to the major number of the actual driver

You can view the descriptive file for the clone driver in */var/sysgen/master.d/clone*. This file sets its major number (10) and states that it is not loadable. Although the clone driver is not specifically configured in the */var/sysgen/system/irix.sm* file, it is included in any kernel because it is listed as a dependency in the descriptive file of several other drivers (use *fgrep clone /var/sysgen/master.d/** to see which drivers depend on it; and see "Listing Dependencies" on page 234). You can specify it as a dependency in the same way, if your driver depends on it.

When a user process opens a device special file with the major number of the clone driver, the kernel naturally calls the clone driver's open entry point. The clone driver verifies that the minor number passed is the major number of an existing, STREAMS driver. (If it is not, the clone driver returns ENXIO).

The clone driver sets up the *qinit* structure appropriately for the target driver's queue and calls that driver's *pfx***open()** entry point, passing the CLONEOPEN flag in the *sflag* argument (see "Entry Point open()" on page 543).

Recognizing a Clone Request Independently

It is not essential to use the clone driver. You can instead designate a particular minor device number to stand for "clone open." You prepare a device special file with these characteristics:

- A device name related to the actual device name
- The major number of your driver
- Some minor number you define to mean "clone open"

When a user process opens this device special file, the kernel calls the *pfx***open()** entry point of your driver. It does not pass the CLONEOPEN flag in *sflag*, but your driver can recognize a request for a clone open based on the minor device number.

Responding to a Clone Request

In response to a clone request coming from either of the two methods described, your *pfx***open()** entry point must select an unused minor device number. (If no minor number is available, return EBUSY.)

Text in Chapter 3 of *STREAMS Modules and Drivers, UNIX SVR4.2* seems to suggest that your driver should scan through the kernel's *cdevsw* table to find an unused minor number (see "Kernel Switch Tables" on page 137). Under IRIX, the *cdevsw* table is not accessible to drivers. The reason is that the table layout differs between 32-bit and 64-bit kernels, and can change between releases. Instead, your driver must know the minor numbers that it supports, and must know which ones are currently in use.

Tip: You can design your driver so that the number of supported devices is specified in the descriptive file in */var/sysgen/master.d*, and passed in to the driver through that descriptive file (see "Variables Section" on page 235). Your driver can allocate and initialize an array of device information structures in its *pfx***init()** entry point.

Your driver constructs a new *dev_t* value, specifying its major number and the selected minor number. The **makedevice()** function is used for this (see the makedevice(D3) reference page, which has some sample code for use in a clone open). The new *dev_t* value is stored into the **devp* argument passed to *pfx***open()**.

Summary of Standard STREAMS Functions

The supported kernel functions for STREAMS operations are summarized for reference in Table 19-2. To declare the necessary prototypes and data types, include *sys/types*.h and *sys/stream.h*.

Table 19-2 Kernel Entry Points

Name	Can Sleep?	Summary
adjmsg(D3)	N	Trim bytes from a message.
allocb(D3)	N	Allocate a message block.
bcanput(D3)	N	Test for flow control in a specified priority band.
bcanputnext(D3)	N	Test for flow control in a specified priority band.
bufcall(D3)	N	Call a function when a buffer becomes available.
canput(D3)	N	Test for room in a message queue.
canputnext(D3)	N	Test for room in a message queue.
copyb(D3)	N	Copy a message block.
copymsg(D3)	N	Copy a message.
datamsg(D3)	N	Test whether a message is a data message.
dupb(D3)	N	Duplicate a message block.
dupmsg(D3)	N	Duplicate a message.
enableok(D3)	N	Allow a queue to be serviced.
esballoc(D3)	N	Allocate a message block using an externally-supplied buffer.
esbbcall(D3)	N	Call a function when an externally-supplied buffer can be allocated.
flushband(D3)	N	Flush messages in a specified priority band.
flushq(D3)	N	Flush messages on a queue.
freeb(D3)	N	Free a message block.
freemsg(D3)	N	Free a message.

Table 19-2 (continued) Kernel Entry Points

Name	Can Sleep?	Summary
freezestr(D3)	N	Freeze the state of a stream.
getq(D3)	N	Get the next message from a queue.
insq(D3)	N	Insert a message into a queue.
linkb(D3)	N	Concatenate two message blocks.
msgdsize(D3)	N	Return number of bytes of data in a message.
msgpullup(D3)	N	Concatenate bytes in a message.
noenable(D3)	N	Prevent a queue from being scheduled.
OTHERQ(D3)	N	Get a pointer to queue's partner queue.
pcmsg(D3)	N	Test whether a message is a priority control message.
pullupmsg(D3)	N	Concatenate bytes in a message.
putbq(D3)	N	Place a message at the head of a queue.
putctl(D3)	N	Send a control message to a queue.
putctl1(D3)	N	Send a control message with a one-byte parameter to a queue.
putnext(D3)	N	Send a message to the next queue.
putnextctl(D3)	N	Send a control message to a queue.
putnextctl1(D3)	N	Send a control message with a one-byte parameter to a queue.
putq(D3)	N	Put a message on a queue.
qenable(D3)	N	Schedule a queue's service routine to be run.
qprocsoff(D3)	Y	Enable put and service routines.
qprocson(D3)	Y	Disable put and service routines
qreply(D3)	N	Send a message in the opposite direction in a stream.
qsize(D3)	N	Find the number of messages on a queue.
RD(D3)	N	Get a pointer to the read queue.

Table 19-2 (continued) Kernel Entry Points

Name	Can Sleep?	Summary
rmvb(D3)	N	Remove a message block from a message.
rmvq(D3)	N	Remove a message from a queue.
SAMESTR(D3)	N	Test if next queue is of the same type.
strqget(D3)	N	Get information about a queue or band of the queue.
strqset(D3)	N	Change information about a queue or band of the queue.
unbufcall(D3)	N	Cancel a pending bufcall request.
unfreezestr(D3)	N	Unfreeze the state of a stream.
unlinkb(D3)	N	Remove a message block from the head of a message.
WR(D3)	N	Get a pointer to the write queue.

STREAMS Modules for X Input Devices

The Silicon Graphics, Inc. implementation of the X display manager, *Xsgi*, is a customized version of the MIT X11 Sample Server. Besides other enhancements such as integration with Silicon Graphics proprietary graphics subsystems, *Xsgi* implements a generalized input subsystem so that unusual input devices can easily be integrated into the X window system. The input system is based on STREAMS modules.

The X Input Subsystem

While X mandates that every X server support a keyboard and mouse, there is no standard system interface for accessing such devices on UNIX systems. This means each vendor has its own input subsystem for its X server. SGI's input subsystem not only meets the basic requirement to support a keyboard and mouse but also has the following features:

- A shared memory input queue is supported for high performance
- A wide variety of input devices is supported, including 3D devices such as the Spaceball

- Input devices are supported abstractly; knowledge of specific input devices is isolated to modular kernel-level device drivers
- Hardware cursor tracking is supported in the kernel

These features provide a more functional, responsive input subsystem than that available in the MIT Sample Server.

The programming interface to the input subsystem from the X client API is covered in the *X11 Input Extension Library Specification*, an online book that is distributed with the IRIX Developer's Option.

Note: Numerous code examples demonstrating the X input system are available in the X developer component (x_dev component) of the IRIX Developer Option. Source for STREAMS modules to integrate a Spaceball, a dial-and-button box, and other devices can be found in subdirectories of */usr/share/src/X*.

Shared Memory Input Queue

A shared memory input queue (called a *shmiq* in Silicon Graphics code comments, and pronounced "shmick") is a fast way of receiving input device events by eliminating the filesystem overhead to receive data from input devices. Instead of the X server reading the input devices through file descriptors, a kernel-level driver deposits input events directly into a region of the X server's address space, organized as a ring buffer.

The IRIX shmiq device driver is implemented as a STREAMS multiplexor. This allows an arbitrary number of input sources (in the form of STREAMS modules) to be linked to it so all input sources are funneled through the shmiq.

In addition to processing input events from input device modules, the schmiq driver also processes events from the graphics subsystem, and updates the screen cursor position. This allows smooth cursor movement since cursor positioning is done in kernel code, without *Xsgi* involvement.

IDEV Interface

X input devices are integrated into the shmiq driver by implementing STREAMS modules that translate raw device input into abstract events which are sent to the shmiq driver (and on to the server). For example, an input device that connects to a serial port can be integrated in the form of a STREAMS module that is pushed onto the stream from that serial device, and translates incoming bytes into event messages.

The shmiq driver expects messages from all input devices to be in the form of IDEV events, as documented in the */usr/include/sys/idev.h* header file; hence this is called the IDEV interface. IDEV device events appear as valuator, button, and pointer state changes.

The IDEV interface defines two-way communications between the input device and *Xsgi*. Besides the uniform set of IDEV input events, the interface defines a standard set of abstract commands that *Xsgi* can send down (using IOCTL messages) to initialize and control input devices. This allows the server to see input devices as abstract input sources and does not require special server code to be written every time a new input device is supported. Instead, device specific knowledge of each devices is encapsulated in an IDEV-based STREAMS module linked into the kernel.

Input Device Naming

Xsgi recognizes as input devices, any device special files in the */dev/input* directory. At a minimum this includes */dev/input/keyboard* and */dev/input/mouse*. Other input devices that are to be integrated into the IDEV interface must also appear in */dev/input*.

Typically an X input device is defined as a link from */dev/input* to some other device special file, for example a serial port in the */dev/tty** series. The filename in */dev/input* determines the name of the STREAMS module that is used to interface that device to the IDEV input system. For example, if the file is */dev/input/calcomp*, the *calcomp* STREAMS module is loaded and pushed onto the stream from the device.

When a single STREAMS module is used to support two or more devices, you can use a hyphen-digit suffix on the filename. For example, the *calcomp* STREAMS module would be used for both */dev/input/calcomp-1* and */dev/input/calcomp-2*.

When a device is initialized (as described in the next section), the STREAMS module is asked to return the X name of the input device. This name can be the same as the name of the device and the module, or it can be different. Typically the device and module names will reflect the hardware type (for example *calcomp*), while the X name reflects the kind of device (for example *tablet*).

Opening Input Devices

An input device is opened at one of two times: when the X server starts up, and when an X client requests an open.

Starting Up the Server

When *Xsgi* starts up, it opens each device name in */dev/input* and for each one it:

- Loads a STREAMS module that has the same name as the name of the device special file, and pushes it onto the stream from the device, below the shmiq multiplexor.

  The STREAMS module may be loadable, and most IDEV modules are loadable.

- Looks for a file in */usr/lib/X11/input/config* having the same name as the module. The device controls in that file are sent down the stream as IOCTL messages.

  The format of device controls is discussed under “Device Controls” on page 560.

- Asks the device to describe itself. This is done by sending down an IOCTL message of the type IDEVDESC. The module must return the IOCTL message with descriptive data.

  The IDEV IOCTL structures are declared in */usr/include/sys/idev.h*. A key element of the device description is the X name of the input device.

- Looks for a file in */usr/lib/X11/input/config* having the X name of the device as returned in the device description. The X init controls in this file are processed by the X server.

  The format of X init controls is discussed under “Device Controls” on page 560.

- Unless autostart was specified for this device, the device is closed.

Opening from a Client

An X application can use the **XListInputDevices()** function to get a list of available input devices. Then it can call **XOpenDevice()** to open a selected device, so that input events from that device will be processed by the X server (see the XListInputDevices(3X) and XOpenDevice(3X) reference pages).

When **XOpenDevice()** is called for an input device that is not already open, it repeats the process done at startup time:

- Loads the STREAMS module and pushes it on the device stream, feeding the shmiq multiplexor.
- Sends device controls from a file in */usr/lib/X11/input/config* having the same name as the module.
- Asks the device (module) to describe itself, including the X name of the device.
- Processes X init controls from a file in */usr/lib/X11/input/config* having the X name of the device.

Device Controls

Device controls are string values that are passed via an IOCTL message to the STREAMS module for an input device at the time the device is opened. You can use device controls as a way of configuring the device module at runtime. Device controls are interpreted only by the module.

X init controls have the same syntax as device controls, but are processed by the X server after the device has been initialized.

Where Controls Are Stored

You can issue X server device controls on the fly by calling **XSGIDeviceControl** from within a program, or by storing them in configuration files in the */usr/lib/X11/input/config* directory. Specific documentation on controls can be found in */usr/lib/X11/input/config/README*.

There are (potentially) two configuration files per device. As noted under "Opening Input Devices" on page 559, the X server looks for device controls in a file with the same name as the STREAMS module that implements the device. After the module returns the X name of the device, the X server looks for X init controls in a file with the X name of the device.

Some devices use the same name for the STREAMS module and for the X device (*tablet*, *mouse*), but some use different names for the two. For example, the STREAMS module for the Spaceball device is *sball*, while the X name is *spaceball*.

The X server intercepts about a dozen **x_init** controls. For a list of the **x_init** controls and some of the more common **device_init** controls, see the file

Control Syntax

When the X server opens a file to look for device controls, it searches the file for a single set of controls with the following format:

```
device_init {
    name     "value"
    ...
}
```

Each *name* may have at most 15 characters. Each *value* may have at most 23 characters. Each pair of name and value are put in an IOCTL message of *idevOtherControl* type and sent down to the device module for interpretation.

When the X server opens a file to look for X init controls, it searches the file for a single set of controls with the following format:

```
x_init {
    name     "value"
    ...
}
```

The syntax is the same, except for the use of x_init instead of device_init.

The specific *name* and *value* strings that the X server supports are documented in the file */usr/lib/X11/input/config/README*. Any *name* strings that are not recognized by the X server are sent down to the device module, just as if they were device controls.