STREAMS Modules and Drivers

CONCURRENT COMPUTER CORPORATION™

0890426-020

February 1998

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- August 1994	000	PowerMAX OS 1.1
Previous Release -- February 1997	010	PowerMAX OS 4.1
Current Release -- February 1998	020	PowerMAX OS 4.2

Preface

Scope of Manual

This manual provides a programming guide for the PowerMAX OS STREAMS facility. It contains reference information and procedures for developing operating system communication services.

Structure of Manual

This manual consists of eleven chapters, a glossary, and an index. A brief description of the chapters is presented as follows:

- Chapter 1 provides an introduction to the manual and an overview of the STREAMS facility. It describes STREAMS components and highlights the main benefits of STREAMS.
- Chapter 2 explains the STREAMS-related system call interface.
- Chapter 3 provides additional information on the STREAMS I/O structure and data flow and contrasts it with the conventional character I/O mechanism.
- Chapter 4 describes the **`put`** and **`service`** procedures and provides an asynchronous protocol Stream example.
- Chapter 5 describes the STREAMS message structure and message queues and priorities. It explains the procedures and interfaces for sending messages.
- Chapter 6 provides an overview of STREAMS modules and drivers, explains the **`ioctl`** mechanism, and describes the device driver/driver-kernel interfaces (DDI/DKI) and the STREAMS interface. It also explains how to configure the system for STREAMS modules and drivers.
- Chapter 7 explains how to develop STREAMS modules.
- Chapter 8 explains how to develop STREAMS drivers.
- Chapter 9 explains how STREAMS multiplexing configurations are created and discusses multiplexing drivers.
- Chapter 10 describes the STREAMS-based Transport Provider Interface (TPI).
- Chapter 11 describes the STREAMS-based Data Link Provider Interface (DLPI).

The glossary contains definitions of technical terms that are important to understanding the concepts presented in this book.

The index contains an alphabetical reference to key terms and concepts and numbers of pages where they occur in text.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments

Referenced Publications

The following publications are referenced in this document:

0890425	*Device Driver Programming*
On line	*Command Reference*
On line	*Operating System API Reference*
On line	*System Files and Devices Reference*
On line	*Device Driver Reference*

Contents

1
Introduction to STREAMS

1
Introduction to STREAMS

Introduction

STREAMS Modules and Drivers describes everything you need to know about the STREAMS tool set so that you can develop PowerMAX OS operating system communication services. It is part of the *Device Driver Programming* series of manuals which includes:

- *Device Driver Programming*
- *Device Driver Reference* (on-line only)

It contains chapters regarding the following components of the STREAMS interface:

- System Calls
- Input/Output operations
- Processing Routines
- STREAMS Messages and Message Types
- Modules and Drivers
- Multiplexing

In addition, it contains chapters about how STREAMS works with the following ISO-standard protocols:

- Transport Provider Interface
- Data Link Provider Interface

A comprehensive glossary covering all of the terms found in the *Device Driver Programming* manual set is also included.

References

This book occasionally refers to other books, notably the reference manuals. The reference manuals are provided only in on-line form.

- *Command Reference* (Section 1)
- *Operating System API Reference* (Sections 2 and 3)
- *Windowing System API Reference* (Section 3 windowing functions)

- *System Files and Devices Reference* (Section 4, 5, and 7)
- *Device Driver Reference* (Sections D1 - D5)

These books contain the manual pages for the various commands, system calls, library functions, file contents, and devices. Within each book, manual pages are grouped numerically by section numbers. Within a section, the pages are sorted alphabetically, without regard to the letter that follows the section number. For example, the manual pages for Sections 3C, 3E, 3I, 3M, 3N, 3S, 3W, and 3X are all sorted together within Section 3 in the *Operating System API Reference.*

Manual pages are referred to with the function name showing first in constant width font, followed by the section number appearing in parentheses in normal font. For example, the Executable and Linking Format Library (ELF) manual page appears as **`elf(3E)`**.

The *Command Reference, Operating System API Reference,* and *System Files and Devices Reference* are foundation documents which describe formally and comprehensively every feature of the PowerMAX OS system and are a recommended supplement to this book.

Notation Conventions

The following conventions are observed in this book:

- Computer input and output appear in `constant width` type. Substitutable values appear in *italic* type:

  `$` **`cc`** *file.c file.c file.c*

  The dollar sign is the default system prompt for the ordinary user. There is an implied `RETURN` at the end of each command. When a command extends beyond the width of the page, the break is marked with a backslash and an indented second line:

  `$` **`cc -L`** *../archives* **`-L`** *../mylibs file1.c file2.c file3.c* **`\`**
  file4`.c` **`-l`** *foo*

  Of course, a command that extends beyond the width of your terminal screen will wrap around. You should use the backslash only if you enter the command exactly as we show it.

- In cases where you are expected to enter a control character, the character is shown as, for example, `control_d` or `^d`. Either form means that you press the `d` key while holding down the `CTRL` key.
- A number in parentheses following a command or function name refers to the section of the reference manuals where the command or function is described. For example, **`cc(1)`**, means that the **`cc`** command is described in Section 1 of the reference manuals. The sections which are in each book are listed earlier, under “References.”

Overview of STREAMS

STREAMS is a general, flexible facility and a set of tools for development of PowerMAX OS system communication services. It supports the implementation of services ranging from complete networking protocol suites to individual device drivers. STREAMS defines standard interfaces for character input/output within the kernel, and between the kernel and the rest of the PowerMAX OS system. The associated mechanism is simple and open-ended. It consists of a set of system calls, kernel resources, and kernel routines.

The standard interface and mechanism enable modular, portable development and easy integration of high-performance network services and their components. STREAMS does not impose any specific network architecture. The STREAMS user interface is upwardly compatible with the character I/O user level functions such as **`open`**, **`close`**, **`read`**, **`write`**, and **`ioctl`**.

A *Stream* is a full-duplex processing and data transfer path between a STREAMS driver in kernel space and a process in user space. See Figure 1-1. In the kernel, a Stream is constructed by connecting a “Stream head,” a “driver,” and zero or more “modules” between the Stream head and driver. The *Stream head* is the end of the Stream nearest to the user process. All system calls made by a user level process on a Stream are processed by the Stream head.

Pipes are also STREAMS-based. A STREAMS-based pipe is a full-duplex (bidirectional) data transfer path in the kernel. It implements a connection between the kernel and one or more user processes and also shares properties of STREAMS-based devices.

A STREAMS driver may be a device driver that provides the services of an external I/O device, or a software driver, commonly referred to as a pseudo-device driver. The driver typically handles data transfer between the kernel and the device and does little or no processing of data other than conversion between data structures used by the STREAMS mechanism and data structures that the device understands.

A STREAMS module represents processing functions to be performed on data flowing on the Stream. The module is a defined set of kernel-level routines and data structures used to process data, status, and control information. Data processing may involve changing the way the data is represented, adding or deleting header and trailer information to data, and/or packetizing and depacketizing data. Status and control information includes signals and input/output control information. Each module is self-contained and functionally isolated from any other component in the Stream except its two neighboring components. The module is not a required component in STREAMS, whereas the driver is, except in a STREAMS-based pipe where only the Stream head is required.

The STREAMS module communicates with its neighbors by passing “messages.” One or more modules may be inserted into a Stream between the Stream head and driver to perform intermediate processing of messages as they pass between the Stream head and driver. STREAMS modules are dynamically interconnected in a Stream by a user process. No kernel programming, assembly, or link editing is required to create the interconnection.

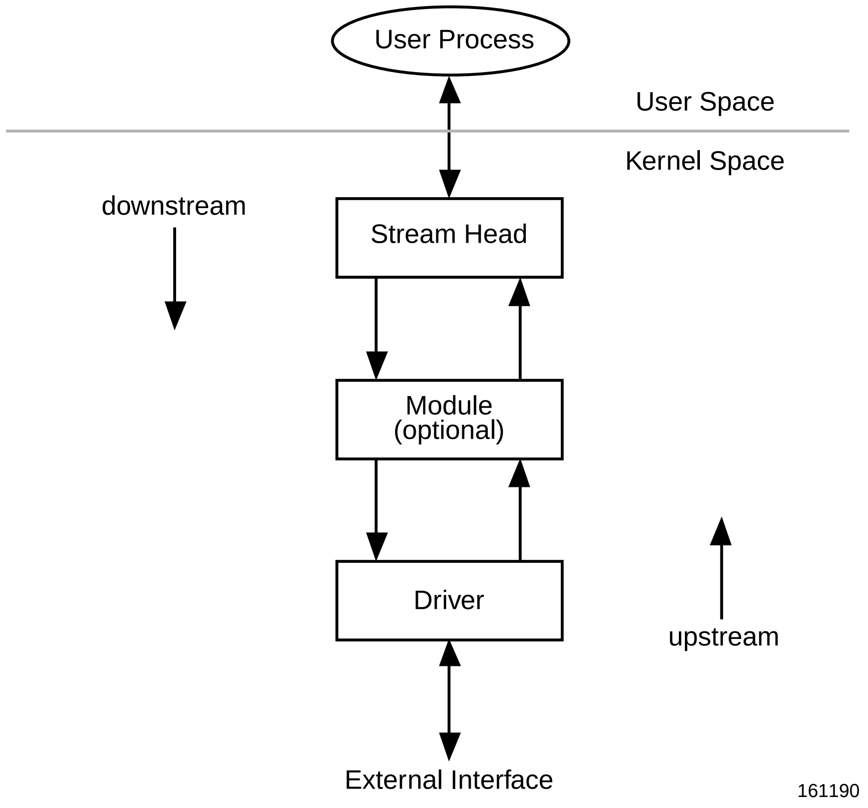


Figure 1-1. Simple STREAMs

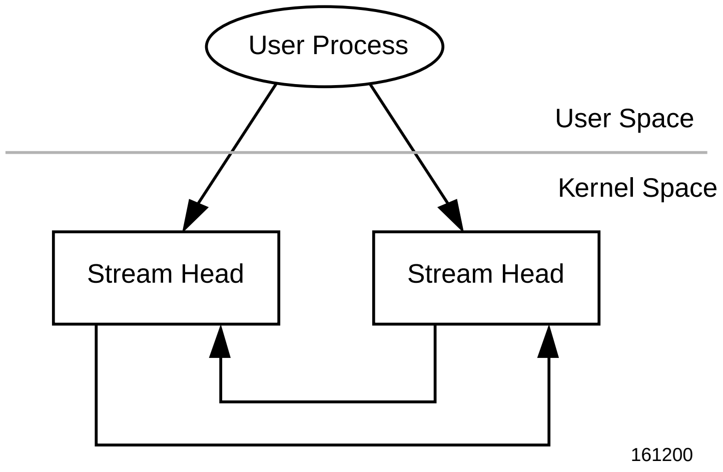


Figure 1-2. STREAMS-based Pipe

STREAMS uses queue structures to keep information about given instances of a pushed module or opened STREAMS device. A *queue* is a data structure that contains status information, a pointer to routines for processing messages, and pointers for administering the Stream. Queues are always allocated in pairs; one queue for the “read-side” and the other for the “write-side.” There is one queue pair for each driver and module, and the Stream head. The pair of queues is allocated whenever the Stream is opened or the module is pushed (added) onto the Stream.

Data is passed between a driver and the Stream head and between modules in the form of messages. A *message* is a set of data structures used to pass data, status, and control information between user processes, modules, and drivers. Messages that are passed from the Stream head toward the driver or from the process to the device, are said to travel *downstream* (also called *write-side*). Similarly, messages passed in the other direction, from the device to the process or from the driver to the Stream head, travel *upstream* (also called *read-side*).

A STREAMS message is made up of one or more “message blocks.” Each *block* consists of a header, a data block, and a data buffer. The Stream head transfers data between the data space of a user process and STREAMS kernel data space. Data to be sent to a driver from a user process is packaged into STREAMS messages and passed downstream. When a message containing data arrives at the Stream head from downstream, the message is processed by the Stream head, which copies the data into user buffers.

Within a Stream, messages are distinguished by a type indicator. Certain message types sent upstream may cause the Stream head to perform specific actions, such as sending a signal to a user process. Other message types are intended to carry information within a Stream and are not directly seen by a user process.

Basic Stream Operations

This section describes the basic set of operations for manipulating STREAMS entities.

A STREAMS driver is similar to a traditional character I/O driver in that it has one or more nodes associated with it in the file system, and it is accessed using the **open** system call. Typically, each file system node corresponds to a separate minor device for that driver. Opening different minor devices of a driver causes separate Streams to be connected between a user process and the driver. The file descriptor returned by the **open** call is used for further access to the Stream. If the same minor device is opened more than once, only one Stream is created; the first **open** call creates the Stream, and subsequent **open** calls return a file descriptor that references that Stream. Each process that opens the same minor device shares the same Stream to the device driver.

Once a device is opened, a user process can send data to the device using the **write** system call and receive data from the device using the **read** system call. Access to STREAMS drivers using **read** and **write** is compatible with the traditional character I/O mechanism.

The **close** system call closes a device and dismantles the associated Stream when the last open reference to the Stream is given up.

The following example shows how a simple Stream is used. In the example, the user program interacts with a communications device that provides point-to-point data transfer between two computers. Data written to the device transmitted over the communications line, and data arriving on the line can be retrieved by reading from the device.

```
#include <fcntl.h>

main()
{
    char buf[1024];
    int fd, count;

    if ((fd = open("/dev/comm/01", O_RDWR)) < 0) {
        perror("open failed");
        exit(1);
    }

    while ((count = read(fd, buf, 1024)) > 0) {
        if (write(fd, buf, count) != count) {
            perror("write failed");
            break;
        }
    }
    exit(0);
}
```

In the example, **`/dev/comm/01`** identifies a minor device of the communications device driver. When this file is opened, the system recognizes the device as a STREAMS device and connects a Stream to the driver. Figure 1-3 shows the state of the Stream following the call to **open**.

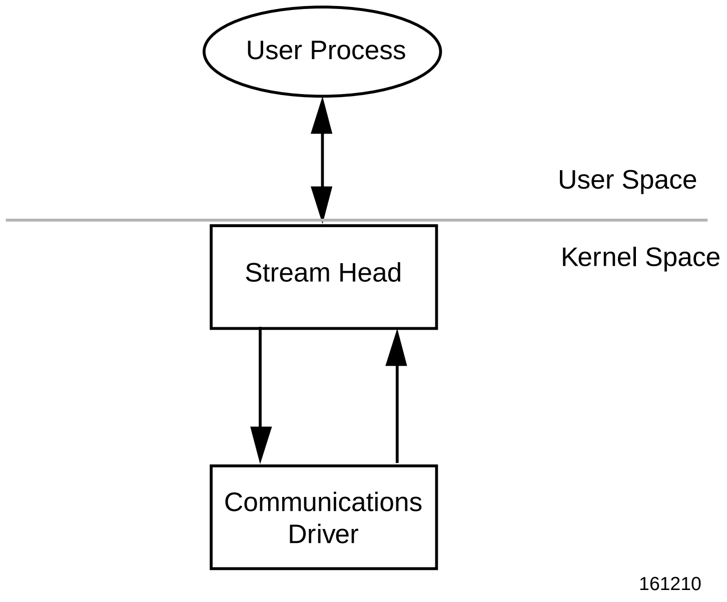


Figure 1-3. Stream to Communication Driver

This example illustrates a user reading data from the communications device and then writing the input back out to the same device. In short, this program echoes all input back over the communications line. The example assumes that a user sends data from the other side of the communications line. The program reads up to 1024 bytes at a time, and then writes the number of bytes just read.

The **`read`** call returns the available data, which may contain fewer than 1024 bytes. If no data is currently available at the Stream head, the **`read`** call blocks until data arrive.

Similarly, the **`write`** call attempts to send *count* bytes to **`/dev/comm/01`**. However, STREAMS implements a flow control mechanism that prevents a user from exhausting system resources by flooding a device driver with data.

Flow control controls the rate of message transfer among the modules, drivers, Stream head, and processes. Flow control is local to each Stream and is advisory (voluntary). It limits the number of characters that can be queued for processing at any queue in a Stream. It also limits buffers and related processing at any queue and in any one Stream. However, it does not consider buffer pool levels or buffer usage in other Streams. Flow control is not applied to high-priority messages.

If the Stream exerts flow control on the user, the **`write`** call blocks until flow control is relieved. The call does not return until it has sent *count* bytes to the device. **`exit`**, which is called to terminate the user process, also closes all open files, and thereby dismantling the Stream in this example.

STREAMS components

This section gives an overview of the STREAMS components and discusses how these components interact with each other. A more detailed description of each STREAMS component is given later.

Queues

A *queue* is an <u>interface</u> between a STREAMS driver or module and the rest of the Stream. Queues are always allocated as an adjacent pair. The queue with the lower address in the pair is a read queue, and the queue with the higher address is used for the write queue.

A queue's *service* routine is invoked to <u>process</u> messages on the queue. It usually removes successive messages from the queue, processes them, and calls the "*put*" routine of the next module in the Stream to give the processed message to the next queue.

A queue's *put* routine is invoked by the preceding queue's *put* and/or *service* routine to <u>add</u> a message to the current queue. If a module does not need to enqueue messages, its *put* routine can call the neighboring queue's *put* routine.

Each queue also has a pointer to an "*open*" and "*close*" routine. The *open* routine of a driver is called when the driver is first opened and on every successive open of the Stream. The *close* routine of the driver is called when the last reference to the Stream is given up and the Stream is dismantled. The *open* routine of a module is called when the module is first pushed on the Stream and on every successive open of the Stream. The *close* routine of the module is called when the module is popped (removed) off the Stream.

Messages

All input and output under STREAMS is based on messages. The objects passed between STREAMS modules are pointers to messages. All STREAMS messages use two data structures (`msgb` and `datab`) to refer to the message data. These data structures describe the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a Stream by successive calls to the **`put`** procedure of each module or driver in the Stream.

Message Types

All STREAMS messages are assigned message types to indicate their intended use by modules and drivers and to determine their handling by the Stream head. A driver or module can assign most types to a message it generates, and a module can modify a message type during processing. The Stream head converts certain system calls to specified message types and sends them downstream. It responds to other calls by copying the contents of certain message types that were sent upstream.

Most message types are internal to STREAMS and can only be passed from one STREAMS component to another. A few message types, for example `M_DATA`, `M_PROTO`, and `M_PCPROTO`, can also be passed between a Stream and user processes. `M_DATA` messages carry data within a Stream and between a Stream and a user process. `M_PROTO` or `M_PCPROTO` messages carry both data and control information.

Figure 1-4 shows that a STREAMS message consists of one or more linked message blocks that are attached to the first message block of the same message.

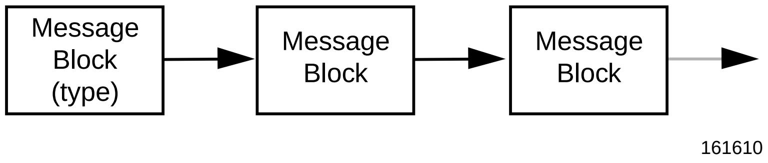


Figure 1-4. A Message

Messages can exist stand-alone, as in Figure 1-4, when the message is being processed by a procedure. Alternately, a message can await processing on a linked list of messages, called a message queue. In Figure 1-5, Message 2 is linked to Message 1.

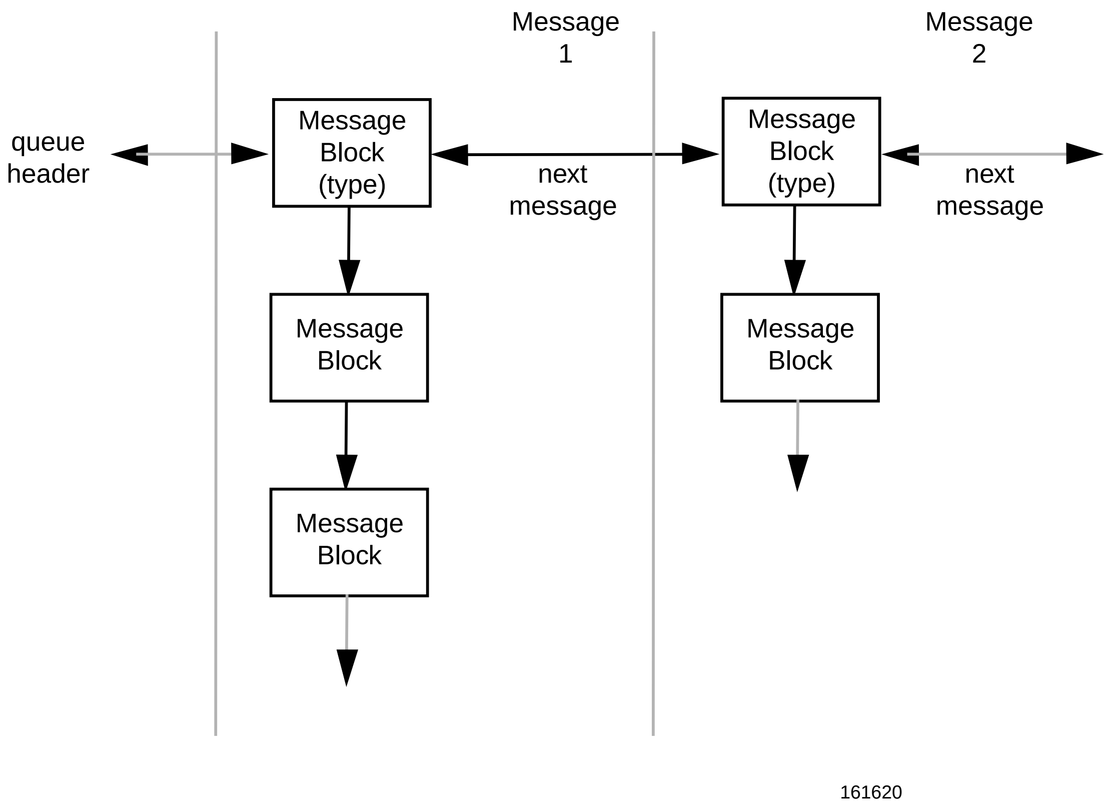


Figure 1-5. Messages on a Message Queue

When a message is on a queue, the first block of the message contains links to preceding and succeeding messages on the same message queue, in addition to the link to the second block of the message (if present). The message queue head and tail are contained in the queue.

STREAMS utility routines enable developers to manipulate messages and message queues.

Message Queuing Priority

In certain cases, messages containing urgent information (such as a break or alarm conditions) must pass through the Stream quickly. To accommodate these cases, STREAMS provides multiple classes of message queuing priority. All messages have an associated priority field. Normal (ordinary) messages have a priority of zero. Priority messages have a priority greater than zero. High-priority messages are high-priority by virtue of their message type. The priority field in high-priority messages is unused and should always be set to zero. STREAMS prevents high priority messages from being blocked by flow control and causes a `service` procedure to process them ahead of all ordinary messages on the queue. This results in the high priority message transiting each module with minimal delay.

Non-priority, ordinary messages are placed at the end of the queue following all other messages in the queue. Priority messages can be either high priority or priority band messages. High-priority messages are placed at the head of the queue but after any other high-priority messages already in the queue. Priority band messages that enable support of urgent, expedited data are placed in the queue after high-priority messages but before ordinary messages.

Message priority is defined by the message type; once a message is created, its priority cannot be changed. Certain message types come in equivalent high priority/ordinary pairs (for example, `M_PCPROTO` and `M_PROTO`), so that a module or device driver can choose between the two priorities when sending information.

Modules

A module performs intermediate transformations on messages passing between a Stream head and a driver. There may be zero or more modules in a Stream (zero when the driver performs all the required character and device processing).

Each module is constructed from a pair of queue structures (see `Au/Ad` and `Bu/Bd` in Figure 1-6). One queue performs functions on messages passing upstream through the module (`Au` and `Bu`). The other set (`Ad` and `Bd`) performs another set of functions on downstream messages.

Each queue in a module generally has distinct functions, that is, unrelated processing procedures and data. The queues operate independently and `Au` will not know if a message passes through `Ad` unless `Ad` is programmed to inform it. Messages and data can be shared only if the developer specifically programs the module functions to perform the sharing.

Each queue connects to the adjacent queue in the direction of message flow (for example, `Au` to `Bu` or `Bd` to `Ad`). In addition, within a module, a queue can readily locate its mate and access its messages and data.

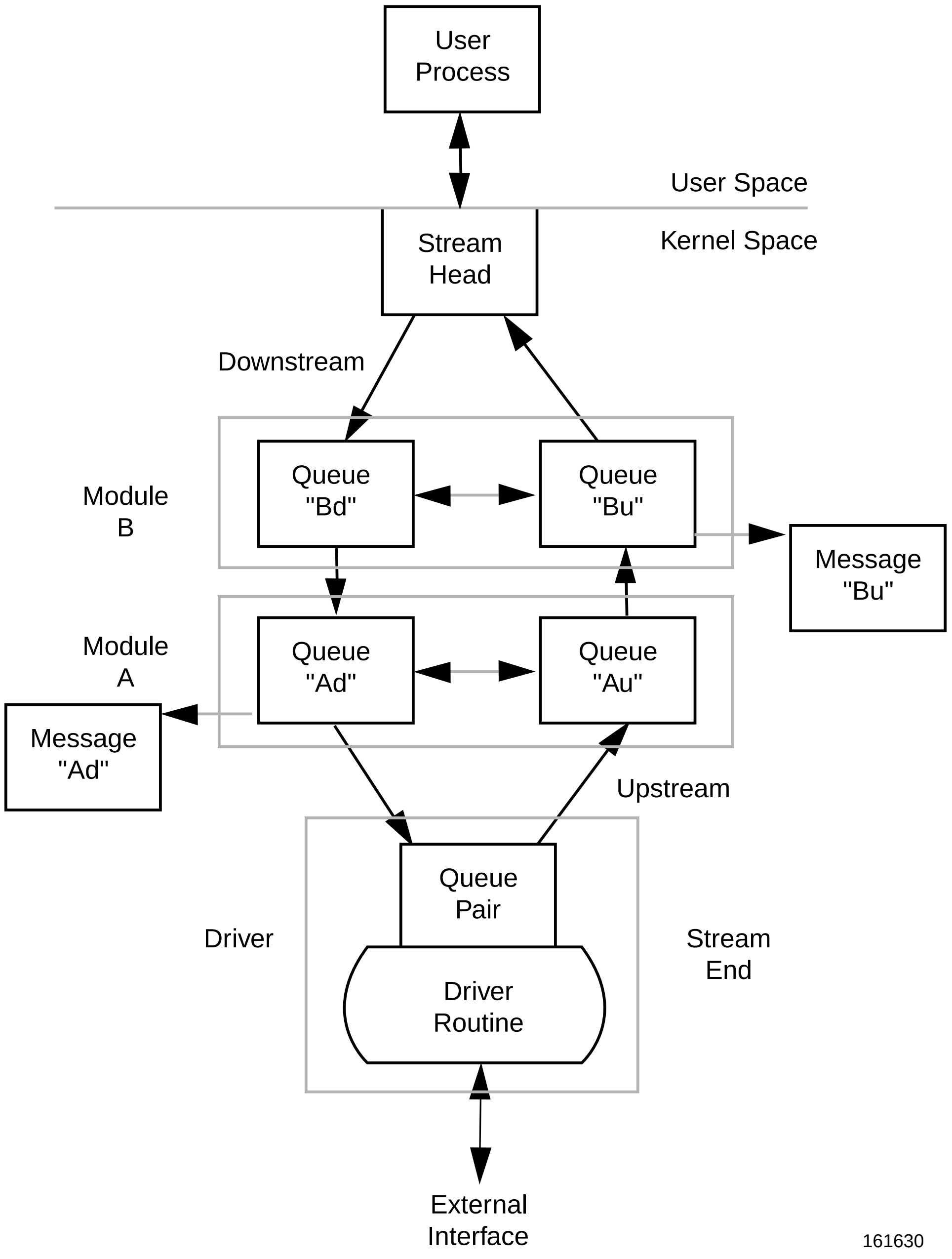


Figure 1-6. Detailed Stream

Each queue in a module points to messages, processing procedures, and data as follows:

- Messages — These are dynamically attached to the queue on a linked list (the message queue, see Ad and Bu in Figure 1-6) as they pass through the module.
- Processing procedures — a **`put`** procedure processes messages and must be incorporated in each queue. An optional **`service`** procedure can also be incorporated. According to their function, the procedures can send mes-

sages upstream and/or downstream, and can also modify the private data in their module.

- Data — developers may use a private field in the queue to reference private data structures (for example, state information and translation tables).

In general, each queue in a module has a distinct set of all these elements.

Drivers

STREAMS device drivers are an initial part of a Stream. They are structurally similar to STREAMS modules. The call interfaces to driver routines are identical to the interfaces used for modules.

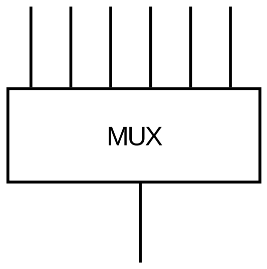
Three significant differences exist between modules and drivers. A driver must be able to handle interrupts from the device, a driver can have multiple Streams connected to it, and a driver is initialized/de-initialized using **`open`** and **`close`**, whereas a module is initialized/de-initialized using **`I_PUSH`** **`ioctl`** and **`I_POP`** **`ioctl`**.

Drivers and modules can pass signals, error codes, and return values to processes using message types provided for that purpose.

Multiplexing

Earlier, Streams were described as linear connections of modules, where each invocation of a module is connected to at most one upstream module and one downstream module. While this configuration is suitable for many applications, others require the ability to multiplex Streams in a variety of configurations. Typical examples are terminal window facilities, and internetworking protocols (which might route data over several subnetworks).

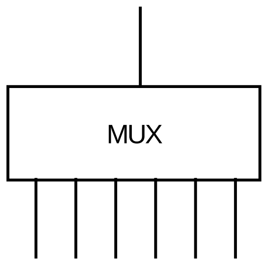
Figure 1-7 shows an example of a multiplexor that multiplexes data from several upper Streams over a single lower Stream. An upper Stream is one that is upstream from a multiplexor, and a lower Stream is one that is downstream from a multiplexor. A terminal windowing facility might be implemented in this fashion, where each upper Stream is associated with a separate window.



161470

Figure 1-7. Many-to-One Multiplexor

Figure 1-8 shows a second type of multiplexor that might route data from a single upper Stream to one of several lower Streams. An internetworking protocol could take this form, where each lower Stream links the protocol to a different physical network.



161480

Figure 1-8. One-to-Many Multiplexor

Figure 1-9 shows a third type of multiplexor that might route data from one of many upper Streams to one of many lower Streams.

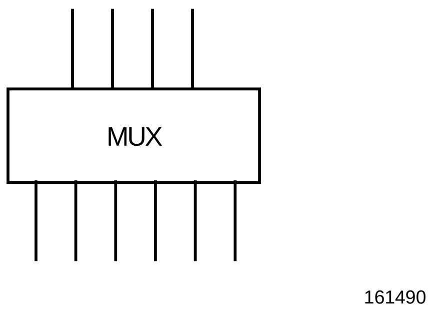


Figure 1-9. Many-to-Many Multiplexor

The STREAMS mechanism supports the multiplexing of Streams through special pseudo-device drivers. Using a linking facility, users can dynamically build, maintain, and dismantle multiplexed Stream configurations. Simple configurations like the ones shown in Figure 1-7 and Figure 1-9 can be further combined to form complex, multilevel multiplexed Stream configurations.

STREAMS multiplexing configurations are created in the kernel by interconnecting multiple Streams. Conceptually, there are two kinds of multiplexors: upper and lower multiplexors. Lower multiplexors have multiple lower Streams between device drivers and the multiplexor, and upper multiplexors have multiple upper Streams between user processes and the multiplexor.

Figure 1-10 is an example of the multiplexor configuration that typically occurs where internetworking functions are included in the system. This configuration contains three hardware device drivers. The IP (Internet Protocol) is a multiplexor.

The IP multiplexor switches messages among the lower Streams or sends them upstream to user processes in the system. In this example, the multiplexor expects to see the same interface downstream to Module 1, Module 2, and Driver 3.

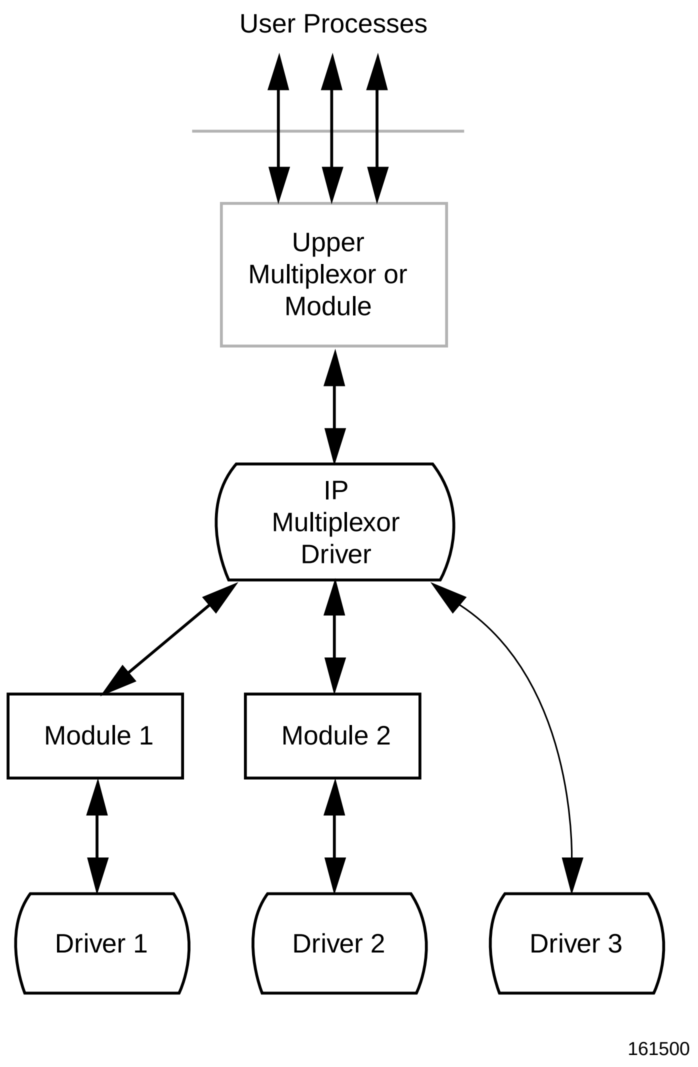


Figure 1-10. Internet Multiplexing Stream

Figure 1-10 depicts the IP multiplexor as part of a larger configuration. The multiplexor configuration, shown in the dashed rectangle, generally has an upper multiplexor and additional modules. Multiplexors can also be cascaded below the IP multiplexor driver if the device drivers are replaced by multiplexor drivers.

Figure 1-11 shows a multiplexor configuration where the multiplexor (or multiplexing driver) routes messages between the lower Stream and one upper Stream. This Stream performs X.25 multiplexing to multiple independent Switched Virtual Circuit (SVC) and Permanent Virtual Circuit (PVC) user processes. Upper multiplexors are a specific application of standard STREAMS facilities that support multiple minor devices in a device driver. This figure also shows that more complex configurations can be built by having one or more multiplexed drivers below and multiple modules above an upper multiplexor.

Developers can choose either upper or lower multiplexing, or both, when designing their applications. For example, a window multiplexor would have a similar configuration to the X.25 configuration of Figure 1-11, with a window driver replacing the Packet Layer, a tty driver replacing the driver XYZ, and the child processes of the terminal process replacing the user processes. Although the X.25 and window multiplexing Streams have similar configurations, their multiplexor drivers would differ significantly. The IP multiplexor in Figure 1-10 has a different configuration than the X.25 multiplexor, and the driver would implement its own set of processing and routing requirements in each configuration.

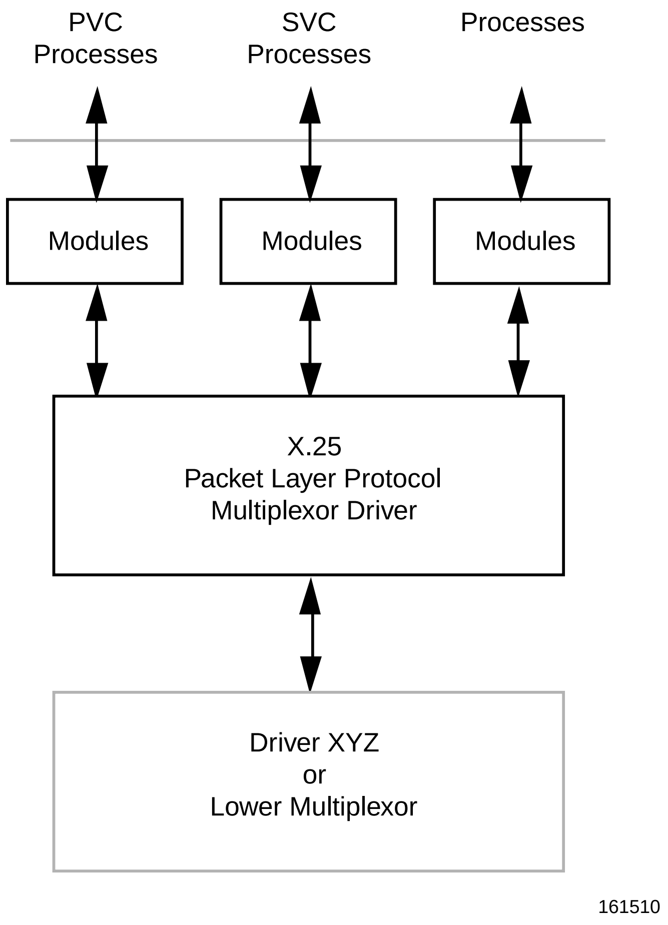

PVC Processes
SVC Processes
Processes
Modules
Modules
Modules
X.25 Packet Layer Protocol Multiplexor Driver
Driver XYZ or Lower Multiplexor
161510

Figure 1-11. X.25 Multiplexing Stream

In addition to upper and lower multiplexors, you can create more complex configurations by connecting Streams containing multiplexors to other multiplexor drivers. With such a diversity of needs for multiplexors, it is not possible to provide general purpose multiplexor drivers. Instead, STREAMS provides a general purpose multiplexing facility, which allows users to set up the intermodule/driver plumbing to create multiplexor configurations of generally unlimited interconnection.

Benefits of STREAMS

STREAMS provides the following benefits:

- A flexible, portable, and reusable set of tools for development of PowerMAX OS system communication services.
- Easy creation of modules that offer standard data communications services and the ability to manipulate those modules on a Stream.
- From user level, modules can be dynamically selected and interconnected; kernel programming, assembly, and link editing are not required to create the interconnection.

STREAMS also greatly simplifies the user interface for languages that have complex input and output requirements.

Standardized Service Interfaces

STREAMS simplifies the creation of modules that present a service interface to any neighboring application program, module, or device driver. A service interface is defined at the boundary between two neighbors. In STREAMS, a service interface is a specified set of messages and the rules that allow passage of these messages across the boundary. A module that implements a service interface receives a message from a neighbor and responds with an appropriate action (for example, sends back a request to retransmit) based on the specific message received and the preceding sequence of messages.

In general, any two modules can be connected anywhere in a Stream. However, rational sequences are generally constructed by connecting modules with compatible protocol service interfaces. For example, a module that implements an X.25 protocol layer, presents a protocol service interface at its input and output sides. See Figure 1-12. In this example, other modules should only be connected to the input and output side if they have the compatible X.25 service interface.

Manipulating Modules

STREAMS provides the capabilities to manipulate modules from the user level, to interchange modules with common service interfaces, and to change the service interface to a STREAMS user process. These capabilities yield further benefits when implementing networking services and protocols, including:

- User level programs can be independent of underlying protocols and physical communication media.
- Network architectures and higher level protocols can be independent of underlying protocols, drivers, and physical communication media.
- Higher level services can be created by selecting and connecting lower level services and protocols.

The following examples show the benefits of STREAMS capabilities for creating service interfaces and manipulating modules. These examples are only illustrations and do not necessarily reflect real situations.

Protocol Portability

Figure 1-12 shows how the same X.25 protocol module can be used with different drivers on different machines by implementing compatible service interfaces. The X.25 protocol module interfaces are Connection Oriented Network Service (CONS) and Link Access Protocol - Balanced (LAPB).

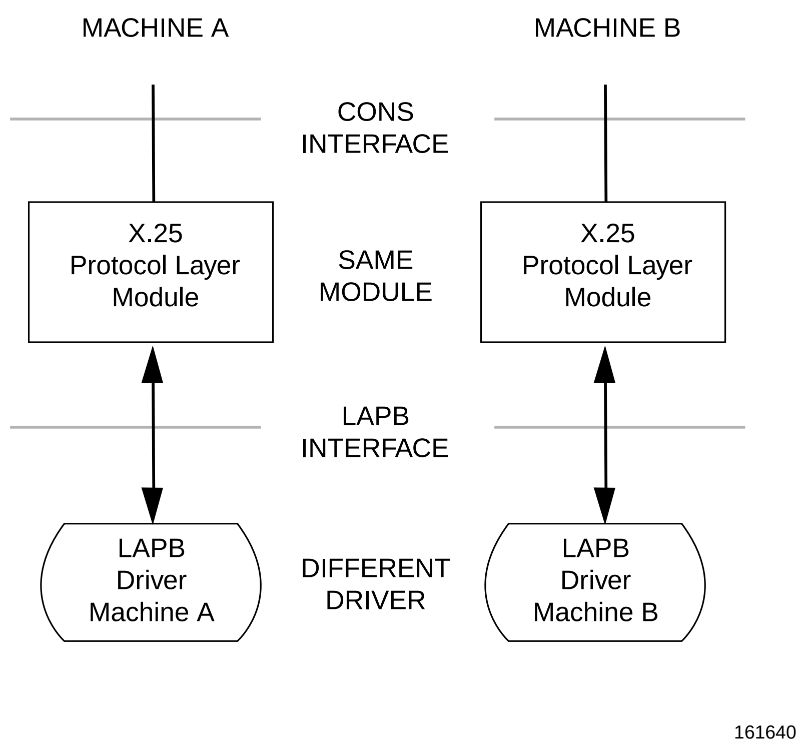


Figure 1-12. X.25 Multiplexing Stream

Protocol Substitution

Alternate protocol modules (and device drivers) can be interchanged on the same machine if they are implemented to an equivalent service interface.

Protocol Migration

Figure 1-13 illustrates how STREAMS can move functions between kernel software and front-end firmware. A common downstream service interface allows the transport protocol module to be independent of the number or type of modules below. The same transport module connects without change to either an X.25 module or X.25 driver that has the same service interface.

By shifting functions between software and firmware, developers can produce cost effective, functionally equivalent systems over a wide range of configurations. They can rapidly incorporate technological advances. The same transport protocol module can be used on a lower capacity machine, where economics may preclude the use of front-end hardware, and also on a larger scale system where a front-end is economically justified.

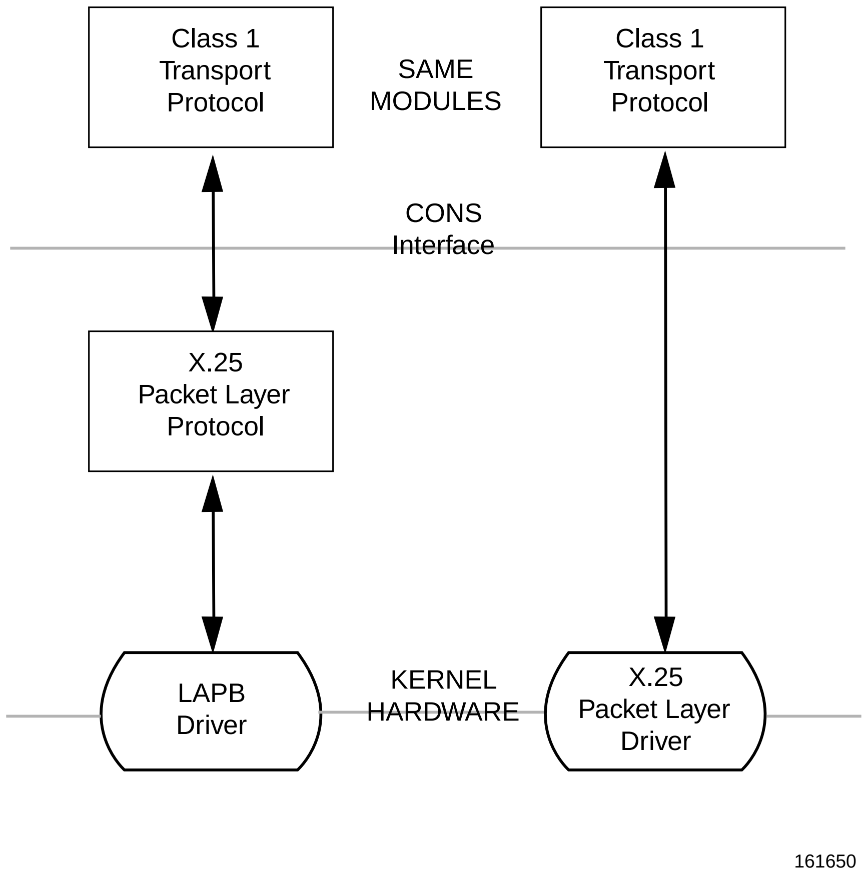


Figure 1-13. Protocol Migration

Module Reusability

Figure 1-14 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different Streams. This module is typically implemented as a filter, with no downstream service interface. In both cases, a tty interface is presented to the Stream's user process because the module is nearest to the Stream head.

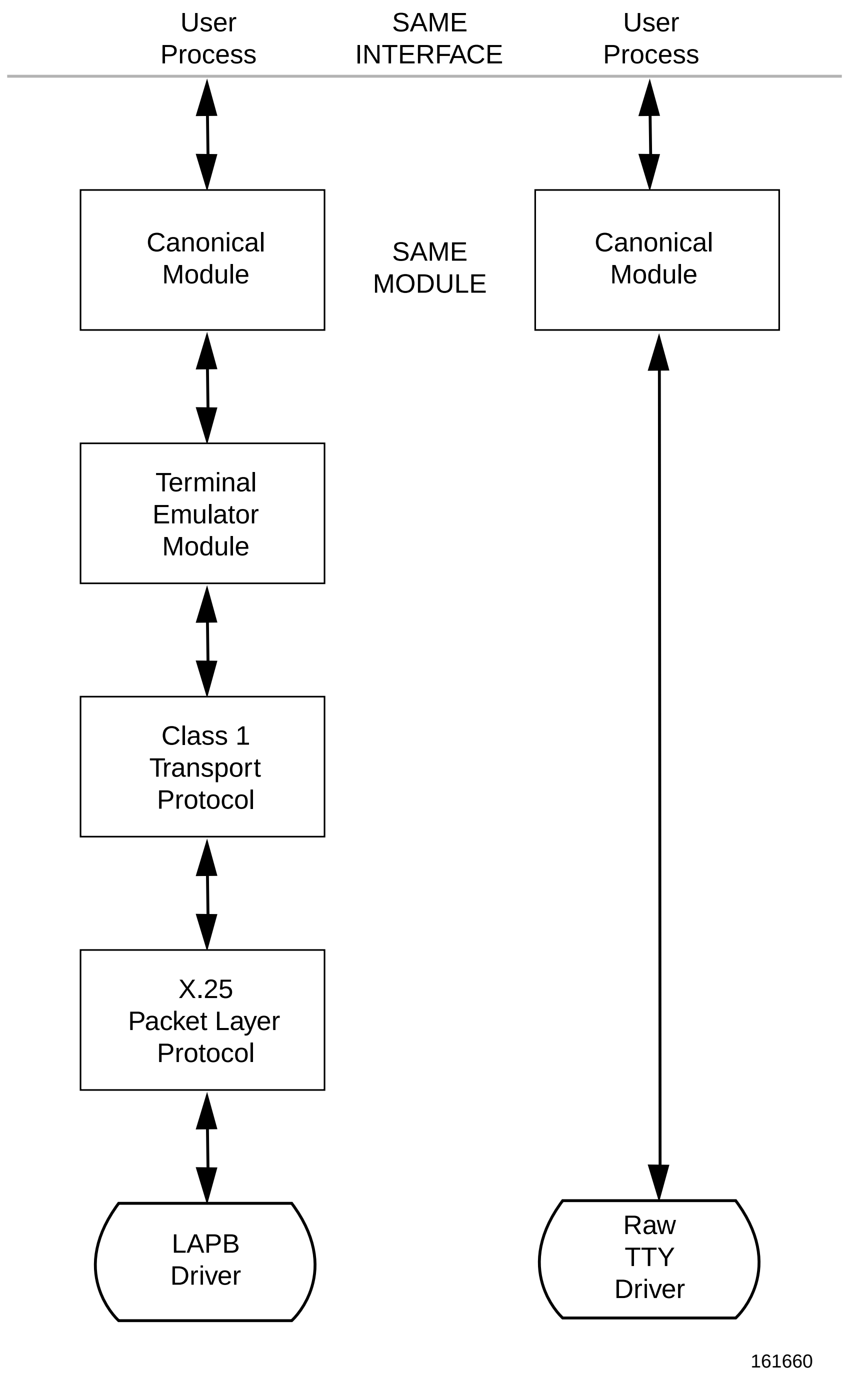


Figure 1-14. Module Reusability

2
STREAMS System Calls

2
STREAMS System Calls

Introduction

This chapter shows how to build, use, and dismantle a Stream using STREAMS-related systems calls. It also contains a section on STREAMS construction.

General and STREAMS-specific system calls provide the user level facilities required to implement application programs. This system call interface is upwardly compatible with the traditional character I/O facilities. The **`open(2)`** system call recognizes a STREAMS file and creates a Stream to the specified driver. A user process can receive and send data on STREAMS files using **`read(2)`** and **`write(2)`** in the same manner as with traditional character files. The **`ioctl(2)`** system call enables users to perform functions specific to a particular device. STREAMS **`ioctl`** commands (see **`streamio(7)`**) support a variety of functions for accessing and controlling streams. The last **`close(2)`** in a Stream dismantles a Stream.

In addition to the traditional **`ioctl`** commands and system calls, there are other system calls used by STREAMS. The **`poll(2)`** system call enables a user to poll multiple Streams for various events. The **`putmsg(2)`** and **`getmsg(2)`** system calls enable users to send and receive STREAMS messages, and are suitable for interacting with STREAMS modules and drivers through a service interface.

STREAMS provides kernel facilities and utilities to support development of modules and drivers. The Stream head handles most system calls so that the related processing does not have to be incorporated in a module or driver.

STREAMS System Calls

A STREAMS device responds to the standard character I/O system calls, such as **`read(2)`** and **`write(2)`**, by turning the request into a message. This feature ensures that STREAMS devices may be accessed from the user level in the same manner as non-STREAMS character devices. However, additional system calls provide other capabilities.

The STREAMS-related system calls are as follows:

`open(2)`	Open a Stream
`close(2)`	Close a Stream
`read(2)`	Read data from a Stream
`write(2)`	Write data to a Stream

ioctl(2)	Control a Stream
getmsg(2)	Receive a message at the Stream head
putmsg(2)	Send a message downstream
poll(2)	Notify the application program when selected events occur on a Stream
pipe(2)	Create a channel that provides a communication path between multiple processes

getmsg and putmsg

The **putmsg(2)** and **getmsg(2)** system calls enable a user process to send and receive STREAMS messages, in the same form the messages have in kernel modules and drivers. **read(2)** and **write(2)** are not designed to include the message boundaries necessary to encode messages.

The advantage of this capability is that a user process, as well as a STREAMS module or driver, can implement a service interface.

poll

The **poll(2)** system call allows a user process to monitor a number of streams to detect expected I/O events. Such events might be the availability of a device for writing, input data arriving from a device, a hangup occurring, an error being detected, or the arrival of a priority message. See **poll(2)** for more information.

STREAM Construction

STREAMS builds a Stream as a linked list of kernel resident data structures. The list is created as a set of linked queue pairs. The first queue pair is the head of the Stream and the second queue pair is the end of the Stream. The end of the Stream represents a device driver, pseudo device driver, or the other end of a STREAMS-based pipe. Kernel routines interface with the Stream head to perform operations on the Stream. Figure 2-1 depicts the upstream (read) and downstream (write) portions of the Stream. Queue H2 is the upstream half of the Stream head and Queue H1 is the downstream half of the Stream head. Queue E2 is the upstream half of the Stream end and Queue E1 is the downstream half of the Stream end.

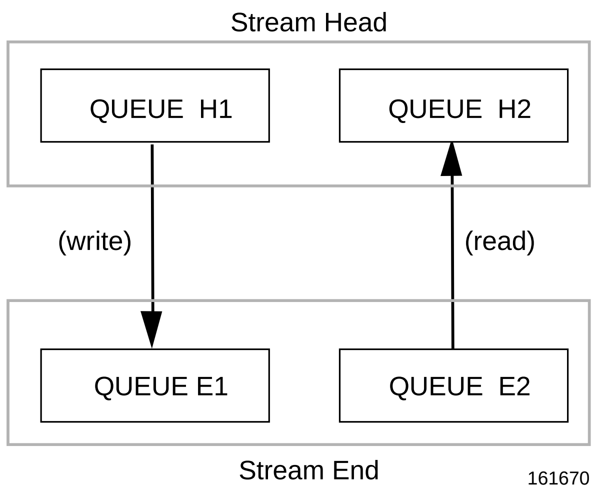


Figure 2-1. Upstream and Downstream Stream Construction

At the same relative location in each queue is the address of the entry point, a procedure to process any message received by that queue. The procedure for Queues H1 and H2 process messages sent to the Stream head. The procedure for Queues E1 and E2, process messages received by the other end of the Stream, the Stream end (tail). Messages move from one end to the other, from one queue to the next linked queue, as the procedure specified by that queue is executed.

Figure 2-2 shows the data structures forming each queue: `queue`, `qinit`, `qband`, `module_info`, and `module_stat`. The `qband` structures have information for each priority band in the queue. The `queue` data structure contains various modifiable values for that queue. The `qinit` structure contains a pointer to the processing procedures, the `module_info` structure contains initial limit values, and the `module_stat` structure is used for statistics gathering. Each queue in the queue pair contains a different set of these data structures. There is a `queue`, `qinit`, `module_info`, and `module_stat` data structure for the upstream portion of the queue pair and a set of data structures for the downstream portion of the pair. In some situations, a queue pair may share some or all the data structures. For example, there may be a separate `qinit` structure for each queue in the pair and one `module_stat` structure that represents both queues in the pair. These data structures are described in the *Device Driver Reference.*

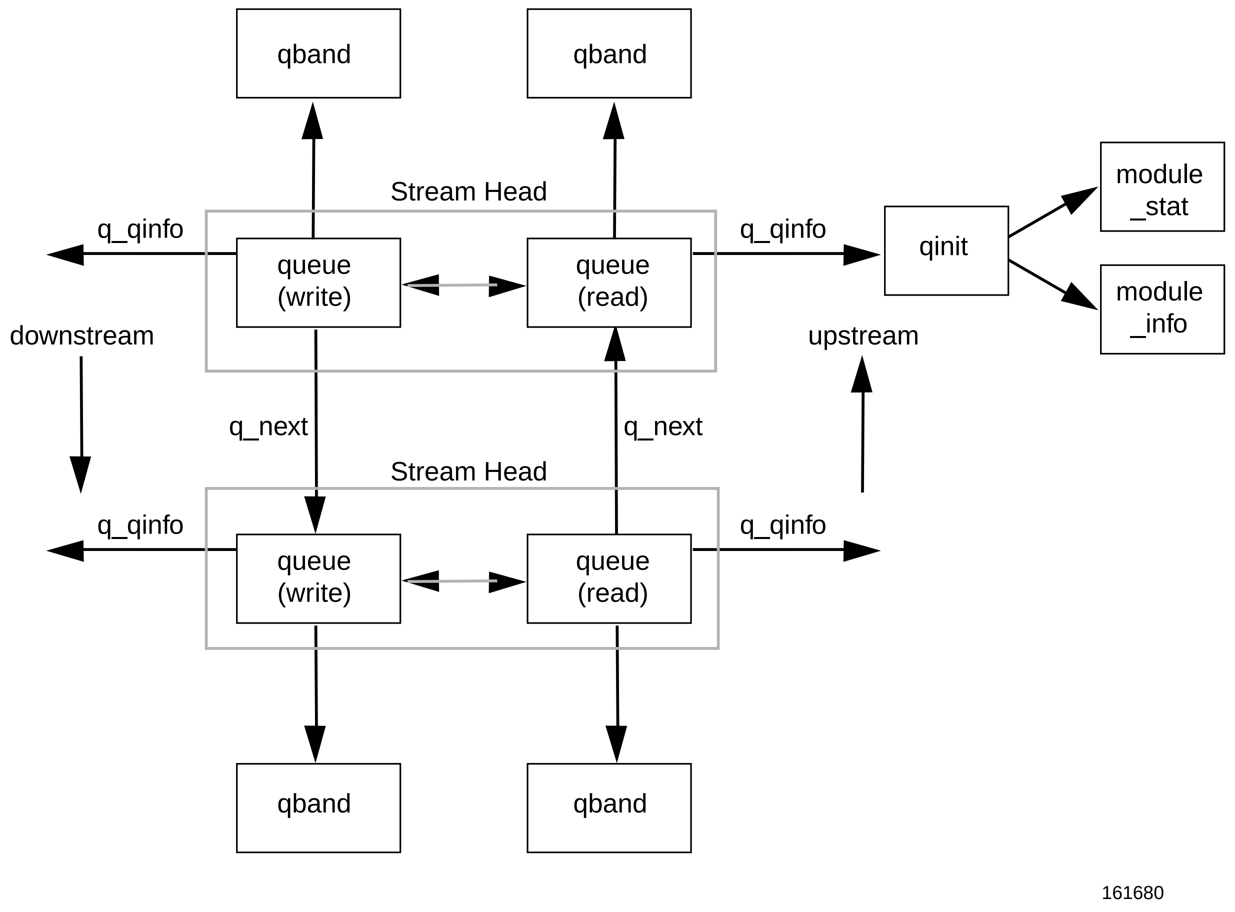


Figure 2-2. Stream Queue Relationship

Figure 2-2 shows two neighboring queue pairs with links (solid vertical arrows) in both directions. When a module is pushed onto a Stream, STREAMS creates a queue pair and links each queue in the pair to its neighboring queue in the upstream and downstream direction. The linkage allows each queue to locate its next neighbor. This relation is implemented between adjacent queue pairs by the `q_next` pointer. Within a queue pair, each `queue` locates its mate (see dashed arrows in Figure 2-2) by use of STREAMS utilities, because there is no pointer between the two `queue`s. The existence of the Stream head and Stream end is known to the queue procedures only as destinations towards which messages are sent.

Opening a STREAMS Device File

One way to build a Stream is to open (see **`open(2)`**) a STREAMS-based driver file as shown in Figure 2-3. All entry points into the driver are defined by the `streamtab` structure for that driver. The `streamtab` structure has a format as follows:

```
struct streamtab {
    struct qinit     *st_rdinit;
    struct qinit     *st_wrinit;
    struct qinit     *st_muxrinit;
    struct qinit     *st_muxwinit;
};
```

The `streamtab` structure defines a module or driver. `st_rdinit` points to the read `qinit` structure for the driver and `st_wrinit` points to the driver's write `qinit` structure. `st_muxrinit` and `st_muxwinit` point to the lower read and write `qinit` structures if the driver is a multiplexor driver.

If the **`open`** call is the initial file open, a Stream is created. (There is one Stream per major/minor device pair.) First, an entry is allocated in the user's file table and a `vnode` is created to represent the opened file. The file table entry is initialized to point to the allocated `vnode` (see `f_vnode` in Figure 2-3) and the `vnode` is initialized to specify a file of type character special.

Second, a Stream header is created from an `stdata` data structure and a Stream head is created from a pair of `queue` structures. The content of `stdata` and `queue` are initialized with predetermined values, including the Stream head processing procedures.

The `snode` contains the file system dependent information. It is associated with the `vnode` representing the device. The `s_commonvp` field of the `snode` points to the common device `vnode`. The `vnode` field, `v_data`, contains a pointer to the `snode`. Instead of maintaining a pointer to the `vnode`, the `snode` contains the `vnode` as an element. The `scavenged` field of `stdata` is initialized to point to the allocated `vnode`. The `v_stream` field of the `vnode` data structure is initialized to point to the Stream header; thus, there is a forward and backward pointer between the Stream header and the `vnode`. There is one Stream header per Stream. The `header` is used by STREAMS while performing operations on the Stream. In the downstream portion of the Stream, the Stream header points to the downstream half of the Stream head queue pair. Similarly, the upstream portion of the Stream terminates at the Stream header, because the upstream half of the Stream head queue pair points to the `header`. Figure 2-3 shows that from the Stream header onward, a Stream is built of linked queue pairs.

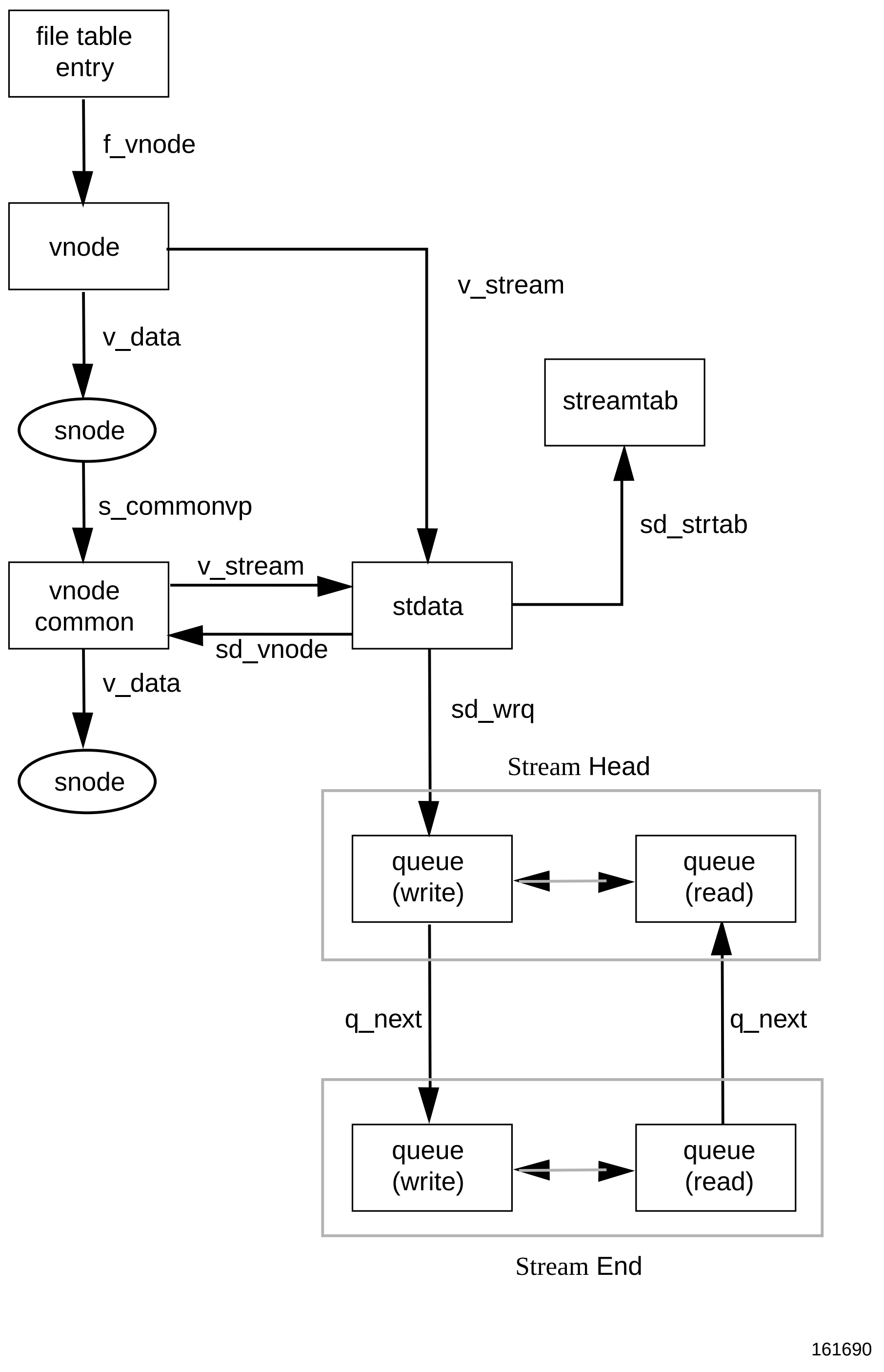


Figure 2-3. Opened STREAMS-Based Driver

Next, a `queue` structure pair is allocated for the driver. The `queue` limits are initialized to those values specified in the corresponding `module_info` structure. The `queue` processing routines are initialized to those specified by the corresponding `qinit` structure.

Then, the `q_next` values are set so that the Stream head write `queue` points to the driver write `queue` and the driver read `queue` points to the Stream head read `queue`. The `q_next` values at the ends of the Stream are set to null. Finally, the driver **open** procedure (located using its read `qinit` structure) is called.

If this is the initial open of this Stream, the driver **open** routine is called. If modules have been specified to be autopushed, they are pushed immediately after the driver **open**. When a Stream is already open, further **open**s of the same Stream result in calls to the **open** procedures of all pushable modules and the driver **open**. Note that this is done in the reverse order from the initial Stream **open**. In other words, the initial **open** processes from the Stream end to the Stream head, while later **open**s process from the Stream head to the Stream end.

Creating a STREAMS-based Pipe

In addition to opening a STREAMS-based driver, a Stream can be created by creating a pipe (see **`pipe(2)`**). Because pipes are not character devices, STREAMS creates and initializes a `streamtab` structure for each end of the pipe. As with modules and drivers, the `streamtab` structure defines the pipe. The `st_rdinit`, however, points to the read `qinit` structure for the Stream head and not for a driver. Similarly, the `st_wrinit` points to the Stream head's write `qinit` structure and not to a driver. The `st_muxrinit` and `st_muxwinit` are initialized to `NULL` because a pipe cannot be a multiplexor driver.

When the **`pipe`** system call is executed, two Streams are created. STREAMS follows the procedures similar to those of opening a driver; however, duplicate data structures are created. Two entries are allocated in the user's file table and two `vnodes` are created to represent each end of the pipe, as shown in Figure 2-4. The file table entries are initialized to point to the allocated `vnode`s and each `vnode` is initialized to specify a file of type `FIFO`.

Next, two Stream headers are created from `stdata` data structures and two Stream heads are created from two pairs of `queue` structures. The content of `stdata` and `queue` are initialized with the same values for all pipes.

Each Stream header represents one end of the pipe, and it points to the downstream half of each Stream head queue pair. Unlike STREAMS-based devices, however, the downstream portion of the Stream terminates at the upstream portion of the other Stream.

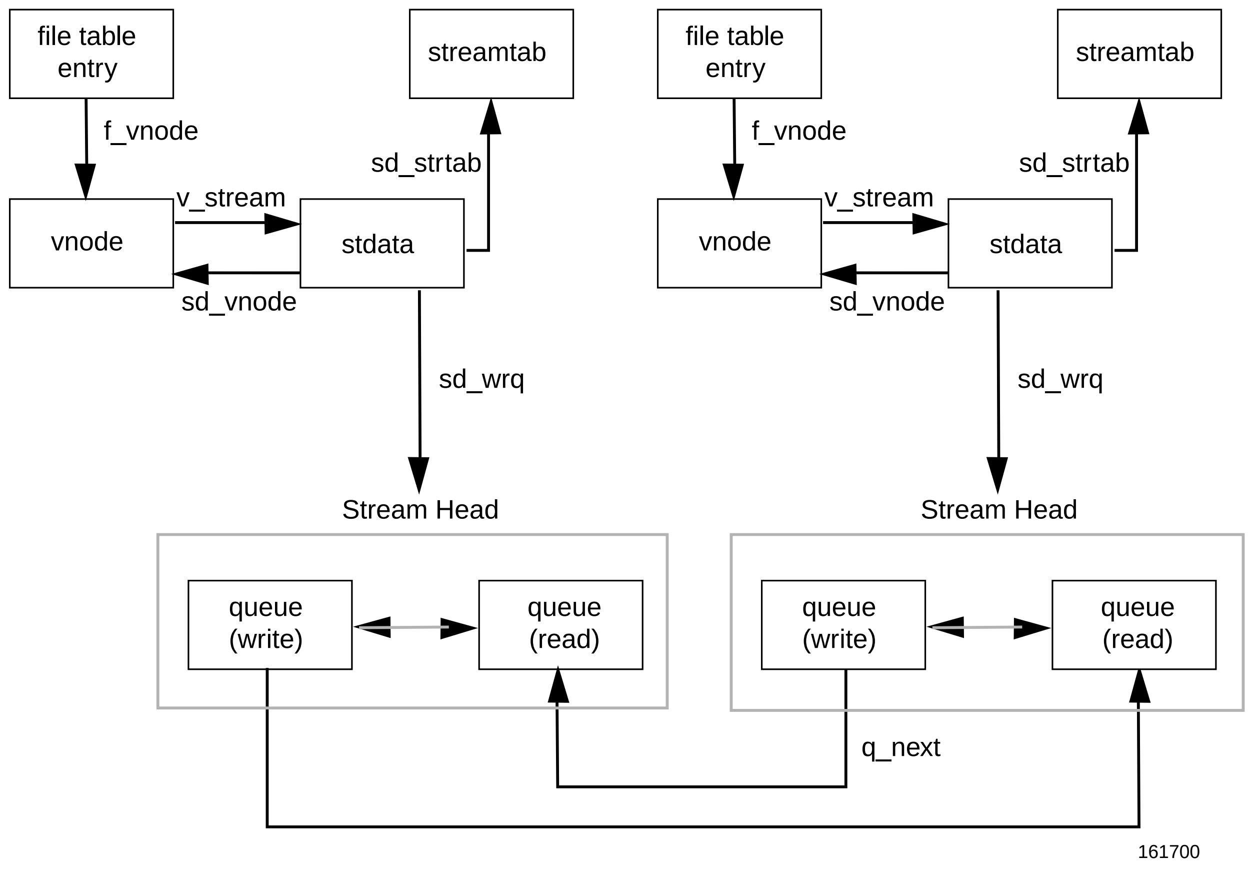


Figure 2-4. Creating STREAMS-Based Pipe

The `q_next` values are set so that the Stream head write `queue` points to the Stream head read `queue` on the other side. The `q_next` values for the Stream head's read `queue` points to null because it terminates the Stream.

Adding and Removing Modules

As part of building a Stream, a module can be added (*push*ed) with an **`ioctl I_PUSH`** (see **`streamio(7)`**) system call. The push inserts a module beneath the Stream head. Because of the similarity of STREAMS components, the push operation is similar to the driver **open**. First, the address of the `qinit` structure for the module is obtained.

Next, STREAMS allocates a pair of `queue` structures and initializes their contents as in the driver **open**.

Then, `q_next` values are set and modified so that the module is interposed between the Stream head and its neighbor immediately downstream. Finally, the module **open** procedure (located using `qinit`) is called.

Each push of a module is independent, even in the same Stream. If the same module is pushed more than once on a Stream, there will be multiple occurrences of that module in the Stream. The total number of pushable modules that may be contained on any one Stream is limited by the kernel parameter `NSTRPUSH`.

An **`ioctl`** **`I_POP`** (see **`streamio(7)`**) system call removes (*pop*s) the module immediately below the Stream head. The pop calls the module **`close`** procedure. On return from the module **`close`**, any messages left on the module's message queues are freed (deallocated). Then, STREAMS connects the Stream head to the component previously below the popped module and deallocates the module's `queue` pair. **`I_PUSH`** and **`I_POP`** enable a user process to alter dynamically the configuration of a Stream by pushing and popping modules as required. For example, a module may be removed and a new one inserted below the Stream head. Then the original module can be pushed back after the new module has been pushed.

Closing the Stream

The last **`close`** to a STREAMS file dismantles the Stream. Dismantling consists of popping any modules on the Stream and closing the driver. Before a module is popped, the **`close`** may delay to allow any messages on the write message queue of the module to be drained by module processing. Similarly, before the driver is closed, the **`close`** may delay to allow any messages on the write message queue of the driver to be drained by driver processing. If `O_NDELAY` (or `O_NONBLOCK`) (see **`open(2)`**) is clear, **`close`** waits up to 15 seconds for each module to drain and up to 15 seconds for the driver to drain. If `O_NDELAY` (or `O_NONBLOCK`) is set, the pop is performed immediately and the driver is closed without delay. Messages can remain queued, for example, if flow control is inhibiting execution of the write queue **`service`** procedure. When all modules are popped and any wait for the driver to drain is completed, the driver **`close`** routine is called. On return from the driver **`close`**, any messages left on the driver's queues are freed, and the `queue` and `stdata` structures are deallocated.

NOTE

STREAMS frees only the messages contained on a message queue. Any message or data structures used internally by the driver or module must be freed by the driver or module **`close`** procedure.

Finally, the user's file table entry and the `vnode` are deallocated and the file is closed.

Stream Construction Example

Screen 2-1 and Screen 2-2 extend the previous communications device echoing example shown in “Basic Stream Operations” in “Introduction,” by inserting a module in the Stream. The (hypothetical) module in this example can convert (change case, delete, and/or duplicate) selected alphabetic characters.

Inserting Modules

An advantage of STREAMS over the traditional character I/O mechanism stems from the ability to insert various modules into a Stream to process and manipulate data that pass between a user process and the driver. In the example, the character conversion module is passed a command and a corresponding string of characters by the user. All data passing through the module are inspected for instances of characters in this string; the operation identified by the command is performed on all matching characters. The necessary declarations for this program are shown in Screen 2-1:

```
#include <string.h>
#include <fcntl.h>
#include <stropts.h>

#define  BUFLEN   1024

/*
 * These defines would typically be
 * found in a header file for the module
 */
#define  XCASE        1   /* change alphabetic case of char */
#define  DELETE       2   /* delete char */
#define  DUPLICATE    3   /* duplicate char */

main()
{
    char buf[BUFLEN];
    int fd, count;
    struct strioctl strioctl;
```

Screen 2-1. Inserting a Module into a STREAM

The first step is to establish a Stream to the communications driver and insert the character conversion module. The following sequence of system calls accomplishes the following display:

```
    if ((fd = open(“/dev/comm/01”, O_RDWR)) < 0) {
        perror(“open failed”);
        exit(1);
    }

    if (ioctl(fd, I_PUSH, “chconv”) < 0) {
        perror(“ioctl I_PUSH failed”);
        exit(2);
    }
```

The **`I_PUSH`** **`ioctl`** call directs the Stream head to insert the character conversion module between the driver and the Stream head, creating the Stream shown in Figure 2-5. As with drivers, this module resides in the kernel and must have been configured into the system before it was booted, unless the system has an autoload capability.

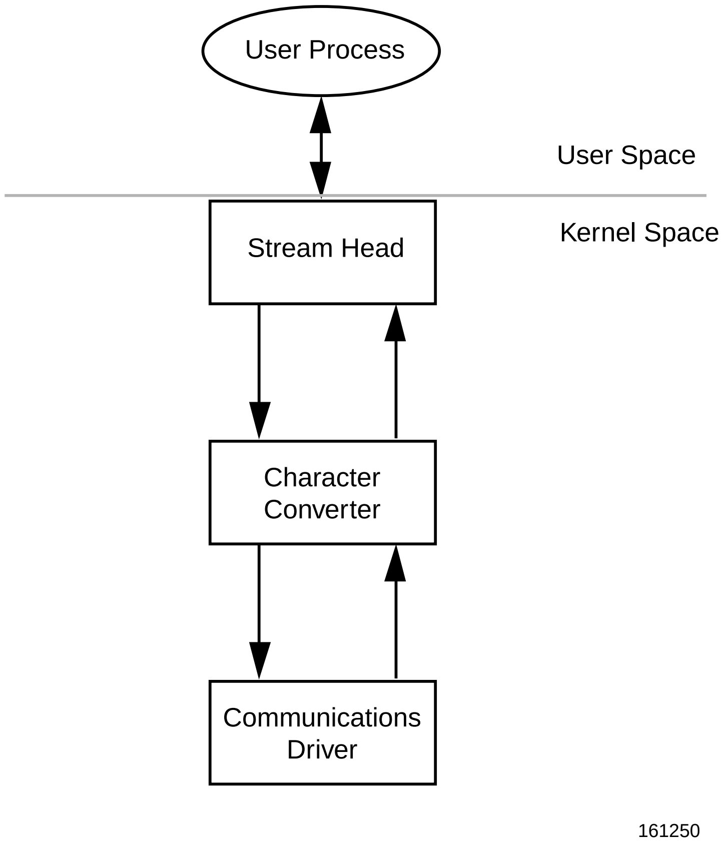


Figure 2-5. Case Converter Module

An important difference between STREAMS drivers and modules is illustrated here. Drivers are accessed through a node or nodes in the file system and may be opened just like any other device. Modules, on the other hand, do not occupy a file system node. Instead, they are identified through a separate naming convention, and are inserted into a Stream using **`I_PUSH`**. The name of a module is defined by the module developer.

Modules are pushed onto a Stream and removed from a Stream in Last-In-First-Out (LIFO) order. Therefore, if a second module was pushed onto this Stream, it would be inserted between the Stream head and the character conversion module.

Module and Driver Control

The next step in this example is to pass the commands and corresponding strings to the character conversion module. This can be done by issuing **`ioctl`** calls to the character conversion shown in Screen 2-2:

```
    /* change all uppercase vowels to lowercase */
    strioctl.ic_cmd = XCASE;
    strioctl.ic_timout = 0;/* default timeout (15 sec) */
    strioctl.ic_dp = “AEIOU”;
    strioctl.ic_len = strlen(strioctl.ic_dp);

    if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror(“ioctl I_STR failed”);
        exit(3);
    }

    /* delete all instances of the chars 'x' and 'X' */
    strioctl.ic_cmd = DELETE;
    strioctl.ic_dp = “xX”;
    strioctl.ic_len = strlen(strioctl.ic_dp);

    if (ioctl(fd, I_STR, &strioctl) < 0) {
        perror(“ioctl I_STR failed”);
        exit(4);
    }
```

Screen 2-2. Module and Driver Control

`ioctl` requests are issued to STREAMS drivers and modules indirectly, using the **`I_STR`** **`ioctl`** call (see **`streamio(7)`**). The argument to **`I_STR`** must be a pointer to a **`strioctl`** structure, which specifies the request to be made to a module or driver. This structure is defined in **`stropts.h`** and has the following format:

```
struct strioctl {
    int      ic_cmd;      /* ioctl request */
    int      ic_timout;   /* ACK/NAK timeout */
    int      ic_len;      /* length of data argument */
    char*    ic_dp;       /* ptr to data argument */
};
```

where `ic_cmd` identifies the command intended for a module or driver, `ic_timout` specifies the number of seconds an **`I_STR`** request should wait for an acknowledgment before timing out, `echelon` is the number of bytes of data to accompany the request, and `ic_dp` points to that data.

In the example, two separate commands are sent to the character conversion module. The first sets `ic_cmd` to the command **`XCASE`** and sends as data the string “`AEIOU`”; it converts all uppercase vowels in data passing through the module to lowercase. The second sets `ic_cmd` to the command **`DELETE`** and sends as data the string “xX”; it deletes all occurrences of the characters ‘x’ and ‘X’ from data passing through the module. For each command, the value of `ic_timout` is set to zero, which specifies the system default timeout value of 15 seconds. The `ic_dp` field points to the beginning of the data for each command; `ic_len` is set to the length of the data.

`I_STR` is intercepted by the Stream head, which packages it into a message, using information contained in the `strioctl` structure, and sends the message downstream. Any module that does not understand the command in `ic_cmd` passes the message further downstream. The request will be processed by the module or driver closest to the Stream head that understands the command specified by `ic_cmd`. The **`ioctl`** call will block up to `ic_timout` seconds, waiting for the target module or driver to respond with either a positive or negative acknowledgment message. If an acknowledgment is not received in `ic_timout` seconds, the **`ioctl`** call will fail.

NOTE

Only one **`I_STR`** request can be active on a Stream at one time. Further requests will block until the active **`I_STR`** request is acknowledged and the system call completes.

The `strioctl` structure is also used to retrieve the results, if any, of an **`I_STR`** request. If data is returned by the target module or driver, `ic_dp` must point to a buffer large enough to hold that data, and `ic_len` will be set on return to show the amount of data returned.

The remainder of this example is identical to the example earlier in this chapter:

```
    while ((count = read(fd, buf, BUFLEN)) > 0) {
        if (write(fd, buf, count) != count) {
            perror(“write failed”);
            break;
        }
    }
    exit(0);
}
```

Note that the character conversion processing was realized with no change to the communications driver.

The **`exit`** system call dismantles the Stream before terminating the process. The character conversion module is removed from the Stream automatically when it is closed. Alternatively, modules may be removed from a Stream using the **`I_POP`** **`ioctl`** call described in **`streamio(7)`**. This call removes the topmost module on the Stream, and enables a user process to alter the configuration of a Stream dynamically, by popping modules as needed.

A few of the important **`ioctl`** requests supported by STREAMS have been discussed. Several other requests are available to support operations such as determining if a given module exists on the Stream, or flushing the data on a Stream. These requests are described fully in **`streamio(7)`**.

3
STREAMS Input/Output

3
STREAMS Input/Output

Introduction

This chapter provides an overview of the STREAMS subsystem, and is intended to highlight the principal differences between STREAMS and standard UNIX® System V block and character device drivers. This includes:

- background information on what advantages STREAMS has over the standard character I/O mechanism
- a general overview of the components of a STREAMS implementation, and how they work together
- a summary of the most important differences between STREAMS and non-STREAMS drivers

The STREAMS Subsystem

The STREAMS subsystem was added to the UNIX operating system to respond to the shortcomings of the character I/O mechanism. It overcame these drawbacks by providing the building blocks for implementing robust, modular data connections for a wide variety of hardware configurations.

The STREAMS subsystem is made up of the following three components:

- *system calls*, some of which are unique to STREAMS and some of which are also used by other types of devices. See the "STREAMS System Calls" chapter for more information.
- *standard kernel functions* (sometimes called primitives) used along with other Block and Character Interface (BCI) functions to write drivers and modules
- *kernel resources* (for example, the STREAMS scheduler) responsible for managing and maintaining streams

In this chapter, STREAMS always refers to this system, which makes it possible to build and use an individual stream. As with other types of devices, a user process communicates with a STREAMS device through system calls. However, opening and communicating with a STREAMS device differs in several ways:

- A user process can select from available modules to build the stream. This feature enhances the portability and reusability of code.

- STREAMS-specific system calls (**`getmsg(2)`** and **`putmsg(2))`** provide a user process with the ability to receive and send STREAMS messages in the same form as they are passed between kernel modules and drivers.
- The **`poll(2)`** system call makes it possible for a user process to monitor several streams for input or output.

Modularity

A user process may build a customized stream using special **`ioctl(2)`** commands to push modules onto a basic stream (consisting only of the stream head and a driver). Modules added to a stream may perform, for example, canonical processing or they may implement a communication protocol. By breaking out functionality into modules, the driver itself can be kept simple and flexible, and modules can be mixed and matched, as needed.

Modules can also be reused by different streams, decreasing the size of code included in the kernel.

Messages

An essential concept in STREAMS programming is the *message*. All transferred data, control information, queue flushing, errors, and signals are transformed into messages in a stream. By imposing this uniformity on all information flowing in a stream, STREAMS can use a standard set of kernel functions and structures for moving and processing messages. To distinguish the different types of information typically passed between devices and processes, STREAMS classifies messages according to two main criteria: message contents type and message priority.

Message Contents Type

Messages are either *data* or *control* type messages. Some examples of control messages are `M_IOCTL` (generated in response to **`ioctl(2)`** system calls), `M_SIG` (sent upstream to post a signal to a process), and `M_DELAY` (to request a real-time delay).

Three message types are classified as data messages: `M_DATA` (which contain only data) and `M_PROTO`, and `M_PCPROTO` (which contain some control information in addition to data). The STREAMS function **`datamsg(D3)`** is used to test a message to see if it is a data message. Several other functions depend on this distinction: **`flushq(D3)`**, **`putnextctl(D3)`**, **`putnextctl1(D3)`**.

Message Priority

Messages are further classified as *ordinary* (also called normal) or *priority* also called (high priority). Normally, messages are passed from module to module by calling the **`put`** routine of a module with a pointer to the message as an argument. The module places ordinary messages on its own message queue where it remains until scheduled for processing. Ordinary messages are defined as those subject to the STREAMS flow control mechanisms, and are processed in the order in which they were placed on the message queues.

Some messages, for example, `error` and `nak` (negative acknowledgment) messages must move through the system quickly and so are designated priority messages. These messages are always placed at the head of the queue of messages waiting to be processed. When the queue's **`service`** routine is called, priority messages are processed before all ordinary messages.

Message Structure

To ensure uniformity in the passing of messages in a stream, all messages share a common structure. A message consists of at least one instance of each of the following three constructs:

- The message block structure (defined as type `mblk_t`) contains next and previous pointers (for message queue formation), pointers to the beginning and end of the data, and a pointer to a continuation block (for messages requiring more than one block), and a pointer to a data block (`dblk_t`).
- The data block structure (defined as type `dblk_t`) includes fields identifying the message type, pointers marking the data boundaries, and a count of the number of messages pointing to this data block.
- The data itself, delineated by fields in the `dblk_t` structure.

For most operations, a message is treated as a unit and is referenced by a pointer to its `mblk_t` structure. See “STREAMS Messages” for more information.

Structure Declarations

Three STREAMS structures must be declared for a driver to be correctly installed on a PowerMAX OS system. First, a `module_info` structure must be declared and populated with information about the queue to be created. Normally, there will be one instance of the structure for both the read and write sides of the driver, but, if they have identical requirements, they may share a `module_info` structure, as shown here.

```
static struct
module_info spminfo = {0, “sp”, 0, INFPSZ, 5120, 1024};
```

The six members of the `module_info` structure are: the identification number, the name, the maximum and minimum packet sizes, and the high and low water marks.

In the above example, no identification number is assigned `0`, the name is `sp`, and no effective minimum and maximum packet sizes are specified (the minimum is `0` and the maximum is set to an infinitely large constant (`INFPSZ`)). The high and low water marks (5120 and 1024, respectively) are for flow control. The specified numbers are compared against a weighted byte count of all messages held on a message queue.

The second required structure is the **`qinit(D4)`** structure. Again, one `qinit` structure is required for each side of the queue. When a stream is opened, the system allocates the required queue structures for the driver, and loads the driver entry point addresses from the `qinit` structures, which must contain non-`NULL` entries for all routines to be included in the driver, as shown:

```
static struct qinit rinit=
{sprput,NULL,spopen, spclose,NULL,   &spminfo,NULL};

static struct qinit winit=
{NULL,NULL,NULL,     NULL,   NULL,   &spminfo,NULL};
```

The seven members of the `qinit` structure are: the **`put`** routine, the **`service`** routine, the **`open`** routine, the **`close`** routine, the **`admin`** routine (reserved for future expansion), and the addresses of the `module_info` and the `module_stat` structures.

The third required structure, `streamtab`, is pointed to by the `cdevsw` table, and contains addresses of the read and write `qinit` structures.

```
struct streamtab spinfo= {&rinit, &winit, NULL, NULL};
```

For multiplexing drivers, a set of upper and lower `qinit` structures are required, and therefore the `streamtab` structure contains four entries, which are shown as follows:

```
struct streamtab spinfo= {&urinit, &uwinit, &lrinit,
                          &lwinit};
```

STREAMS Entry Points

This section outlines each of the entry points that may be included in a STREAMS module or driver. The inclusion of a particular routine depends on the functionality required. For example, the `CLONE` driver (described later in this section) has only an **`open`** routine. The emphasis of this section is on how drivers, not modules, use these routines. Much of the information applies to modules as well.

Open Routine

When a STREAMS device is opened (with the **`open(2)`** system call), the subsystem uses the **`cdevsw`** structure to identify the device type, and creates a stream consisting of the stream head and the driver. The driver's **`open`** routine is then executed. Though similar to a non-STREAMS driver `open`, the STREAMS routine has a different syntax. The *dev* and *flag* arguments are the same as in the standard driver **`open`**, although some of the *flag* values do not apply to STREAMS devices. The other two arguments are

- *q*, a pointer to the queue structure, which in turn, contains a pointer to the `qinit` structure which points to the driver's **`open`** routine.
- *sflag*, the stream **`open`** flag, which has a value of `0` for a normal driver open, `MODOPEN` for a normal module open, or `CLONEOPEN` if the `CLONE` driver is used. The `CLONE` driver is discussed in the next section.

The driver must return `0` if it is successful, or an appropriate `errno` value if it is not.

The only entry points that can communicate with the user level in a stream are the **`open`** and **`close`** routines. Only these two routines can sleep (at a priority less than `PZERO`), but must explicitly return to the driver routine if a signal is received. They may also access the `user` structure.

The **`open`** and **`close`** routines must be specified in the read queue of the driver.

The **`open`** routine may be used to initialize a private driver data structure, pointed to by the `q_ptr` member of the queue structure. See **`open(D2)`** for more information about opening STREAMS drivers.

The CLONE Driver

When the value of *sflag* has been set to `CLONEOPEN`, the `CLONE` driver is invoked. The `CLONE` driver has been provided to select a minor device number (that is, an unused stream). Without the `CLONE` driver, user processes would have to make an **`ioctl(2)`** call to search through a driver's minor devices for an unused one. To eliminate this requirement, STREAMS allows a driver to be implemented as a *cloneable* device. The `CLONE` driver removes the need to search for an unused stream.

Networking applications may sometimes require a separate stream for each communication channel. Because the user process needs a minor device number but is not concerned about the particular number, the `CLONE` driver is used to make the selection.

The `CLONE` driver consists of only an **`open`** routine. The minor portion of the device number passed to the `CLONE` driver is actually the major number of the cloneable device. The `CLONE` driver looks for the cloneable device in the `cdevsw`. See **`clone(7)`** and *Device Driver Programming* for more details.

The driver **`open`** routine must first test the *sflag* to see if it has been set to a value of `CLONEOPEN`. If it has, the driver searches for the first unused minor device number, up to `devcnt` (the maximum number of streams this device may support). An example is shown here.

```
case CLONEOPEN:
   for (dev=0; dev < devcnt; dev++);
```

The value of `devcnt` is derived from the `#DEV` field of the **`master.d`** file during the configuration process. The code shown then searches through the table until it finds the first open minor device, and returns that value.

Message Processing

The **put** and **service** routines represent the two basic ways a STREAMS module or driver processes messages. The **put** routine bypasses the flow control mechanism altogether, providing the fastest possible throughput. **srv** (service) routines are subject to flow control by the scheduler. A driver that must wait for output to complete before sending another message should use the scheduling mechanism.

Whether a driver has a **put** routine, a **srv** routine, or both depends on the processing the driver must perform. For a module, which has at least a driver downstream and the stream head upstream, **put** and **srv** procedures may be appropriate for both queue of the module. However, a driver does not normally need to have a write **service** routine, because it is not passing messages to another queue.

The flexibility of STREAMS makes it almost impossible to establish rigid rules about which routines a driver should include. The next two sections show some of the typical processing done by the **put** and **srv** routines.

put Routine

The pseudo-code example shown below is a generic write queue **put** routine for a driver. It illustrates the basic structure used by many drivers.

```
xxwput(q, mp)
queue_t *q;
mblk_t *mp;
{
    switch (message type) {

    case M_FLUSH:
        flush specified queues
        send flush message upstream
         free message block

    case M_IOCTL:
        if recognizable command type
             handle
        else
             send M_IOCNAK message upstream

    case M_DATA:
        output data to device

    default:
        send error message upstream
  )
```

Screen 3-1. Pseudo-Code for a put Routine

The main task of this routine is to detect the incoming message's type, and then use a `switch` statement to process each type. The next five subsections illustrate how different message types are typically handled.

Each line of the pseudo-code will be expanded into C language statements illustrating how the functionality is implemented.

put Routine: Switch on Message Type

The message type of a STREAMS message is stored in the `db_type` field of the data block (`dblk_t`) structure of the message. If `mp` is a pointer to the message block, the type can be referenced with the following statement:

```
switch (mp->b_datap->db_type)
```

This line corresponds to line 5 in Screen 3-1.

put Routine: Flush Handling

Drivers must flush messages queues when either the `FLUSHR` (flush the read queue) or `FLUSHW` (flush the write queue) bits have been set.

```
1           if (*mp->b_rptr & FLUSHW) flushq(q, FLUSHDATA);
2           if (*mp->b_rptr & FLUSHR) {
3                 flushq(RD(q), FLUSHDATA);
4                 *mp->rptr &=  FLUSHW;
5                 qreply(q, mp);
6                 return;
7           }
8           freemsg(mp);
```

Screen 3-2. put Routine Example of Flush Handling

In line 1, the first byte of the message (`mp->b_rptr`) is tested to see if `FLUSHW` is set. If it is, the **`flushq(D3)`** function is called to remove messages from queue *q*. The `FLUSHDATA` flag removes only data messages; `FLUSHALL` would also remove control messages.

If the `FLUSHR` bit is also set (line 2), messages destined for the user process are flushed. The **`RD(D3)`** macro (line 3) is used to access the mate queue of *q*. The `FLUSHW` bit is then cleared (line 4) and the message is sent upstream (line 5). In line 8 the **`freemsg(D3)`** function deallocates the memory used by the message and data blocks.

put Routine: I/O Control Commands

The stream head interprets **`I_STR`** type **`ioctl(2)`** commands and constructs `M_IOCTL` messages from them. Processing depends on the driver and type of message. If the message type is not recognized, the driver should send a negative acknowledgment message back upstream, as shown here.

```
mp->b_datap->db_type = M_IOCNAK;
qreply(q, mp);
return;
```

Screen 3-3. put Routine Example of I/O Control Command Handling

Using the same message buffer as the incoming message, the driver changes the incoming message into a message of type `M_IOCNAK` and sends the negative acknowledgment back upstream via **`qreply()`**.

put Routine: Data Output

`M_DATA` messages may represent the normal type of data for output to the device. Processing may occur in line, or more likely, in a subordinate routine that is called to handle the actual output.

Data messages also may be enqueued for processing by the **`srv`** (service) procedure with the **`putq(D3)`** function, as shown in the example.

`putq``(`*q*`,` *mp*`);`

The arguments to the function are the pointer to the queue to be enabled (*q*) and a pointer to the message to be enqueued (*mp*).

put Routine: Error Detection

The default case is included to catch all unrecognized message types received by the driver.

```
mp->b_datap->db_type = M_ERROR;
mp->b_rptr = mp->b_datap->db_base;
*mp->b_rptr = EPROTO;
mp->b_wptr = mp->b_rptr+1;
qreply(q, mp);
return;
```

Screen 3-4. put Routine Example of Default Error Handling

In the same way as the negative acknowledgment was sent upstream, an error message (`M_ERROR`) is sent to the user process.

Service Routine

The service routine is called by the STREAMS scheduler to process messages linked to the queue. The scheduler calls service routines for all active queue in FIFO order.

Service routines are typically included in modules rather than in drivers, and a driver's downstream (write) queue generally does not need one. On the upstream side, some drivers may simply discard data if unable to pass it to the next queue. If this approach is inappropriate, the driver's read queue may include a **`service`** routine, as shown in this pseudo-code example.

```
xxrsrv(q)
queue_t *q;
{
     while (more messages on queue)
         retrieve next message
         if (next queue is full)
            put back on queue
         else
            process message
            send to next queue
}
```

Screen 3-5. Pseudo-Code for Service Routine

The rest of this section shows how a driver typically handles read-side messages. As was done in the **`put`** routine example, C language fragments corresponding to the pseudo-code will be presented.

Service Routine: Retrieve Message

The **`getq(D3)`** function attempts to retrieve the next message on the queue.

```
while (mp = getq(q))
```

Service Routine: Check for Blocking

The upstream queue must be tested with the **`canputnext(D3)`** function to see if the message may be passed to the next **`put`** procedure.

```
if (!canputnext(q->q_next))
```

Service Routine: Return Message to Queue

The **`putbq(D3)`** function places the message back on the queue, and awaits a successful **`canputnext`** call. All priority messages are placed ahead of ordinary messages.

`putbq`(*q*, *mp*)

The message pointed at by *mp* is placed at the beginning of the message queue pointed at by *q*.

Service Routine: Forward Message

The **`putnext(D3)`** macro passes the messages to the **`put`** procedure of the next queue upstream, but only after **`canputnext`** has succeeded.

`putnext``(`*q*`,` *mp*`)`

mp is a pointer to the message to be sent, and *q* is a pointer to the sending (not the next, or receiving) queue.

Close Routine

Like the **`open`** routine, the **`close`** routine is specified in the read queue of the driver. The argument to the **`close`** routine is a pointer to the queue.

Typically, the **`close`** routine of a STREAMS driver performs the following functions:

- clears the fields in private driver data structures by setting them to `NULL`
- flushes messages from both queue (read and write) associated with the driver, using the **`flushq(D3)`** function
- sends an `M_HANGUP` message to notify connected processes that the stream is being dismantled
- frees allocated message blocks with the **`freemsg(D3)`** function

4
STREAMS Processing Routines

4
STREAMS Processing Routines

Introduction

The **put** and **service** procedures in the queue are routines that process messages as they transit the queue. The processing is generally performed according to the message type and can result in a modified message, new message(s), or no message. A resultant message, if any, is generally sent in the same direction in which it was received by the queue, but may be sent in either direction. Typically, each **put** procedure places messages on its queue as they arrive, for later processing by the **service** procedure.

A queue will always contain a **put** procedure and may also contain an associated **service** procedure. Having both a **put** and **service** procedure in a queue enables STREAMS to provide the rapid response and the queuing required in multiuser systems.

The **service** and **put** routines pointed at by a queue, and the queues themselves, are not associated with any process. These routines may not sleep if they cannot continue processing, but must instead return. Any information about the current status of the queue must be saved by the routine before returning.

Put Procedure

A **put** procedure is the queue routine that receives messages from the preceding queue in the Stream. Messages are passed between queues by a procedure in one queue calling the **put** procedure contained in the following queue. A call to the **put** procedure in the appropriate direction is the only way to pass messages between STREAMS components. There is usually a separate **put** procedure for the read and write queues because of the full-duplex operation of most Streams. However, there can be a single **put** procedure shared between both the read and write queues.

The **put** procedure allows rapid response to certain data and events, such as echoing of input characters. It has higher priority than any scheduled **service** procedure and is associated with immediate, as opposed to deferred, processing of a message.

The **put** procedure executes before the **service** procedure for any given message.

In a multiprocessor system, both procedures could be running simultaneously.

Each STREAMS component accesses the adjacent **put** procedure indirectly using the DDI functions (for example, **putnext**).

NOTE

Under no circumstances may a driver or module directly call other driver or module routines, including **`put`** and **`service`** routines. All calls are indirect. See the *Device Driver Reference* for further information.

For example, consider that *modA*, *modB*, and *modC* are three consecutive components in a Stream, with *modC* connected to the Stream head. If *modA* receives a message to be sent upstream, *modA* processes that message and calls *modB*'s read **`put`** procedure, which processes it and calls *modC*'s read **`put`** procedure, which in turn processes it and calls the Stream head's read **`put`** procedure. Thus, the message is passed along the Stream in one continuous processing sequence. This sequence completes the entire processing in a short time with low overhead (subroutine calls). On the other hand, if this sequence is lengthy and the processing is implemented on a multiuser system, then this way of processing may be good for this Stream but may be harmful for others. Streams may have to wait too long to get their turn, because each **`put`** procedure is called from the preceding one, and the kernel stack (or interrupt stack) grows with each function call. The possibility of running off the stack exists, causing a system panic or producing indeterminate results.

NOTE

Because STREAMS modules in general do not know which modules they are connected to, **`put`** routines cannot depend on a message being handled solely by **`put`** routines at the stream head or in the driver. Any modules along the Stream may choose to queue the message and process it with a service routine.

Service Procedure

In addition to the **`put`** procedure, a **`service`** procedure may be contained in each queue to allow deferred processing of messages. If a queue has both a **`put`** and a **`service`** procedure, message processing is generally divided between both procedures. The **`put`** procedure is always called first, from a preceding queue. After completing its part of the message processing, it arranges for the **`service`** procedure to be called by passing the message to the **`putq`** routine. **`putq`** does two things: it places the message on the message queue of the queue and schedules the queue service procedure for deferred execution. When `putq` returns to the **`put`** procedure, the procedure can return or continue to process messages. Some time later, the **`service`** procedure is automatically called by the STREAMS scheduler.

The STREAMS scheduler is separate and distinct from the PowerMAX OS system process scheduler. The scheduler calls each **`service`** procedure of the scheduled queues one at a time in a FIFO manner.

The scheduling of queue **`service`** routines is machine-dependent.

STREAMS utilities deliver the messages to the processing **`service`** routine in the FIFO sequence within each priority class (high priority, priority band, ordinary), because the **`service`** procedure is unaware of the message priority and simply receives the next message. The **`service`** routine receives control in the order it was scheduled. When the **`service`** routine receives control, it may encounter multiple messages on its message queue. This buildup can occur if there is a long interval between the time a message is queued by a **`put`** procedure and the time that the STREAMS scheduler calls the associated **`service`** routine. In this interval, multiple calls to the **`put`** procedure can cause multiple messages to build up. The **`service`** procedure always processes all messages on its message queue unless prevented by flow control.

Terminal output and input erase and kill processing, for example, is typically performed in a **`service`** procedure because this type of processing does not have to be as timely as echoing. A **`service`** procedure also allows processing time to be more evenly spread among multiple Streams. As with the **`put`** procedure, there can be a separate **`service`** procedure for each queue in a STREAMS component or a single procedure used by both the read and write queues.

Asynchronous Protocol Stream Example

In the following example, the system supports different kinds of asynchronous terminals, each logging in on its own port. The port hardware is limited in function; for example, it detects and reports line and modem status, but does not check parity.

Communications software support for these terminals is provided using a STREAMS-based asynchronous protocol. The protocol includes a variety of options that are set when a user dials in to log on. The options are determined by a STREAMS user process, **`getstrm`**, which analyzes data sent to it through a series of dialogs (prompts and responses) between the process and the terminal user.

The process sets the terminal options for the duration of the connection by pushing modules onto the Stream or by sending control messages to cause changes in modules (or in the device driver) already on the Stream. The options supported include

- ASCII or EBCDIC character codes
- For ASCII code, the parity (odd, even or none)
- Echo or not echo input characters
- Canonical input and output processing or transparent (raw) character handling

These options are set with the following modules:

`CHARPROC`	Provides input character processing functions, including dynamically settable (using control messages passed to the module) character echo and parity checking. The module's default settings are to echo characters and not check character parity.
`CANONPROC`	Performs canonical processing on ASCII characters upstream and downstream (note that this performs some processing in a way different from the conventional UNIX system character I/O tty subsystem).

`ASCEBC`	Translates EBCDIC code to ASCII upstream and ASCII to EBCDIC downstream.

At system initialization, a user process, **`getstrm`**, is created for each tty port. **`getstrm`** opens a Stream to its port and pushes the `CHARPROC` module onto the Stream by an **`ioctl I_PUSH`** command. Then, the process issues a **`getmsg`** system call to the Stream and sleeps until a message reaches the Stream head. The Stream is now in its idle state.

The initial idle Stream, shown in Figure 4-1, contains only one pushable module, `CHARPROC`. The device driver is a limited function raw tty driver connected to a limited-function communication port. The driver and port transparently transmit and receive one unbuffered character at a time.

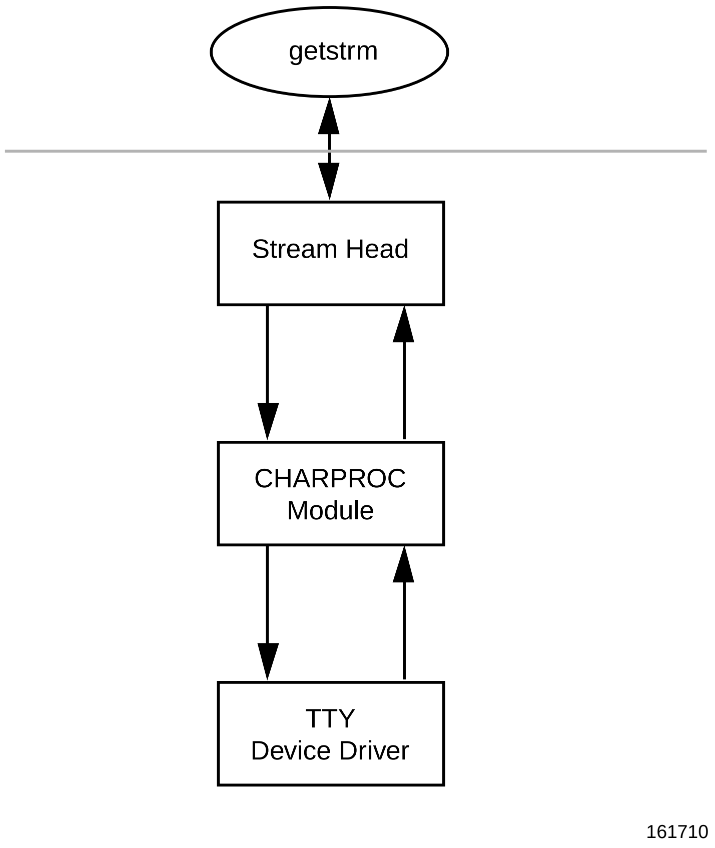


Figure 4-1. Idle Stream Configuration for Example

After receiving initial input from a tty port, **`getstrm`** establishes a connection with the terminal, analyzes the option requests, verifies them, and issues STREAMS system calls to set the options. After setting up the options, **`getstrm`** creates a user application process. Later, when the user terminates that application, **`getstrm`** restores the Stream to its idle state by similar system calls.

Figure 4-2 continues the example and associates kernel operations with user-level system calls. As a result of initializing operations and pushing a module, the Stream for port one has the following configuration:

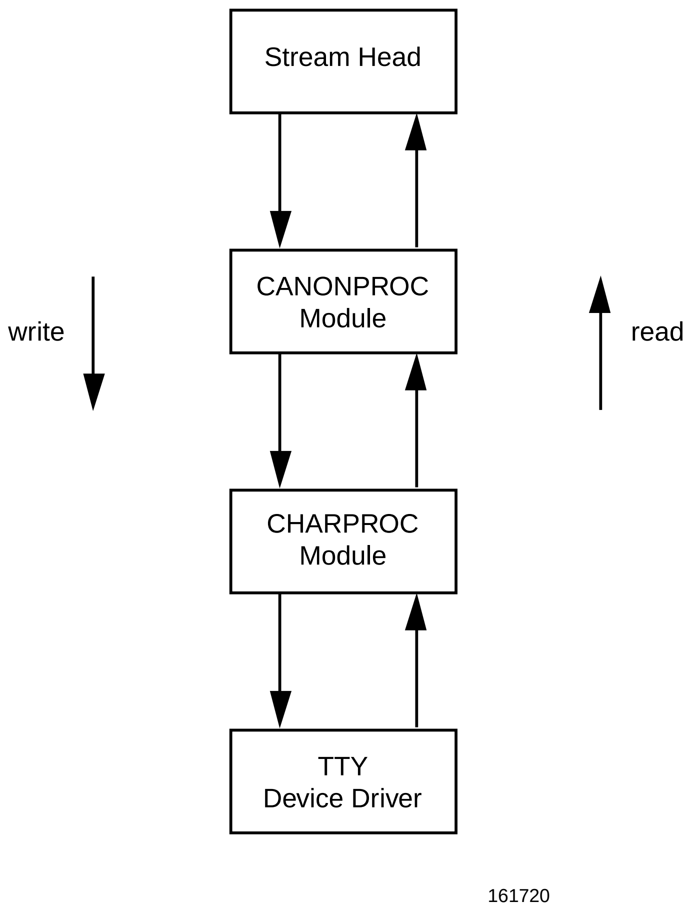


Figure 4-2. Operational Stream for Example

As mentioned before, the upstream queue is also referred to as the read queue reflecting the message flow direction. Correspondingly, downstream is referred to as the write queue.

Read-Side Processing

In our example, read-side processing consists of driver processing, `CHARPROC` processing, and `CANONPROC` processing.

Driver Processing

The user process has been blocked on the **`getmsg(2)`** system call while waiting for a message to reach the Stream head, and the device driver independently waits for input of a character from the port hardware or for a message from upstream. After receiving an input character interrupt from the port, the driver places the associated character in an `M_DATA` message, allocated previously. Then, the driver sends the message to the `CHARPROC` module by calling `CHARPROC`'s upstream **`put`** procedure. On return from `CHARPROC`, the driver calls the **`allocb`** utility routine to get another message for the next character.

CHARPROC

`CHARPROC` has both **`put`** and **`service`** procedures on its read-side. In the example, the other queues in the modules also have both procedures, as shown in Figure 4-3.

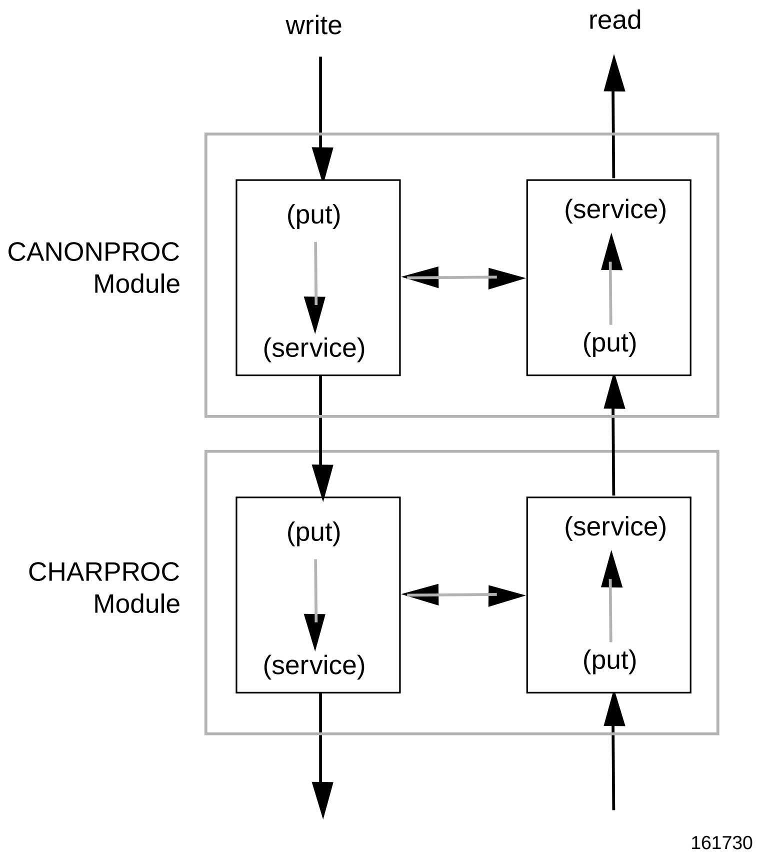


Figure 4-3. Module Put and Service Procedures

When the driver calls `CHARPROC`'s read queue **`put`** procedure, the procedure checks private data flags in the queue. In this example, the flags indicate that echoing is to be performed.

NOTE

Echoing is optional for this example and the port hardware can not automatically echo.

CHARPROC causes the echo to be transmitted back to the terminal by first copying the message with a STREAMS utility routine. Then, CHARPROC uses another utility routine to obtain the address of its own write queue. Finally, the CHARPROC read **put** procedure uses another utility routine to call its write **put** procedure and pass it the message copy. The write procedure sends the message to the driver to effect the echo and then returns to the read procedure.

This part of read-side processing is implemented with **put** procedures so that the entire processing sequence occurs as an extension of the driver input character interrupt.

After returning from echo processing, the CHARPROC read **put** procedure checks another of its private data flags and determines that parity checking should be performed on the input character. Parity should most reasonably be checked as part of echo processing. However, for this example, parity is checked only when the characters are sent upstream. This relaxes the timing in which the checking must occur, that is, it can be deferred along with the canonical processing. CHARPROC uses **putq** to schedule the (original) message for parity check processing by its read **service** procedure. When the CHARPROC read **service** procedure is complete, it forwards the message to the read **put** procedure of CANONPROC. Note that if parity checking was not required, the CHARPROC **put** procedure would call the CANONPROC **put** procedure through the **putnext** routine.

CANONPROC

CANONPROC performs canonical processing. As implemented, all read queue processing is performed in its **service** procedure so that CANONPROC's **put** procedure simply calls **putq** to schedule the message for its read **service** procedure and then exits. The **service** procedure extracts the character from the message buffer and places it in the line buffer contained in another M_DATA message it is constructing. Then, the message that contained the single character is returned to the buffer pool. If the character received was not an end-of-line, the **service** procedure returns. Otherwise, a complete line has been assembled and CANONPROC sends the message upstream to the Stream head that unblocks the user process from the **getmsg(2)** call and passes it the contents of the message.

Write-Side Processing

The write-side of this Stream carries two kinds of messages from the user process: **ioctl** messages for CHARPROC and M_DATA messages to be output to the terminal.

ioctl messages are sent downstream as a result of an **ioctl(2)** system call. When CHARPROC receives an **ioctl** message type, it processes the message contents to change internal flags and then uses a utility routine to send an acknowledgment message upstream to the Stream head. The Stream head acts on the acknowledgment message by unblocking the user from the **ioctl**.

For terminal output, it is presumed that M_DATA messages, sent by **write(2)** system calls, contain multiple characters. In general, STREAMS returns to the user process immediately after processing the **write** call so that the process may send additional messages. Flow control eventually blocks the sending process. The messages can queue on the write-side of the driver because of character transmission timing. When a message is received by the driver's write **put** procedure, the procedure uses **putq** to place the message on its write-side **service** message queue if the driver is currently transmitting a previous message buffer. However, there is generally no write queue **`service`** procedure in a device driver. Driver output interrupt processing takes the place of scheduling and performs the **`service`** procedure functions, removing messages from the queue.

Analysis

For reasons of efficiency, a module implementation would generally avoid placing one character per message and using separate routines to echo and parity check each character, as was done in this example. Nevertheless, even this design yields potential benefits. Consider a case where alternate, more intelligent, port hardware was substituted. If the hardware processed multiple input characters and performed the echo and parity checking functions of `CHARPROC`, then the new driver could be implemented to present the same interface as `CHARPROC`. Other modules such as `CANONPROC` could continue to be used without change.

5
STREAMS Messages

5
STREAMS Messages

Introduction

Messages are the means of communication within a Stream. All input and output under STREAMS is based on messages. The objects passed between Streams components are pointers to messages. All messages in STREAMS use two data structures to refer to the data in the message. These data structures describe the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a Stream by successive calls to the **`put`** routine of each queue in the Stream using the appropriate utility routines. Messages may be generated by a driver, a module, or by the Stream head.

Expedited Data

The Open Systems Interconnection (OSI) Reference Model developed by the International Standards Organization (ISO) and International Telegraph and Telephone Consultative Committee (CCITT) provides an international standard seven-layer architecture for the development of communication protocols. PowerMAX OS adheres to this standard and also supports the Transmission Control Protocol and Internet Protocol (TCP/IP).

OSI and TCP/IP support the transport of expedited data (see note below) for transmission of high-priority, emergency data. This data is useful for flow control, congestion control, routing, and various applications where immediate delivery of data is necessary.

Expedited data is mainly used for exceptional cases and transmission of control signals. Expedited data is processed immediately, ahead of normal data on the queue, but after STREAMS high-priority messages and after any expedited data already on the queue.

Expedited data flow control is unaffected by the flow control constraints of normal data transfer. Expedited data has its own flow control because it can easily run the system out of buffers if its flow is unrestricted.

Drivers and modules define separate high- and low-water marks for priority band data flow. (Water marks are defined for each queue and identify the upper and lower limit of bytes that can be contained on the queue.) The default water marks for priority band data and normal data are the same. The Stream head also ensures that incoming priority band data is not blocked by normal data already on the queue by associating a priority with the messages. This priority implies a certain ordering of the messages in the queue. See “Message Queues and Priorities” for more information.

NOTE

Within the STREAMS mechanism and in this guide expedited data is also referred to as priority band data.

Message Structure

All messages are composed of one or more message blocks. A message block is a linked triplet of two structures and a variable length data buffer. The structures are a message block (`msgb`) and a data block (`datab`). The data buffer is a location in memory where the data of a message are stored.

See **`datab(D4DK)`** and **`msgb(D4DK)`** for fields that can be referenced in data and message blocks.

The field `b_band` determines where the message is placed when it is enqueued using the STREAMS utility routines. This field has no meaning for high priority messages and is set to zero for these messages. When a message is allocated via **`allocb`**, the `b_band` field will be initially set to zero. Modules and drivers may set this field if so desired.

Message Linkage

The message block is used to link messages on a message queue, link message blocks to form a message, and manage the reading and writing of the associated data buffer. The `b_rptr` and `b_wptr` fields in the `msgb` structure locate the data currently contained in the buffer. As shown in Figure 5-1, the message block (`mblk_t`) points to the data block of the triplet. The data block contains the message type, buffer limits, and control variables. STREAMS allocates message buffer blocks of varying sizes. `db_base` and `db_lim` are the fixed beginning and end (+1) of the buffer.

A message consists of one or more linked message blocks. Multiple message blocks in a message can occur, for example, because of buffer size limitations, or as the result of processing that expands the message. When a message is composed of multiple message blocks, the type associated with the first message block determines the message type, regardless of the types of the attached message blocks.

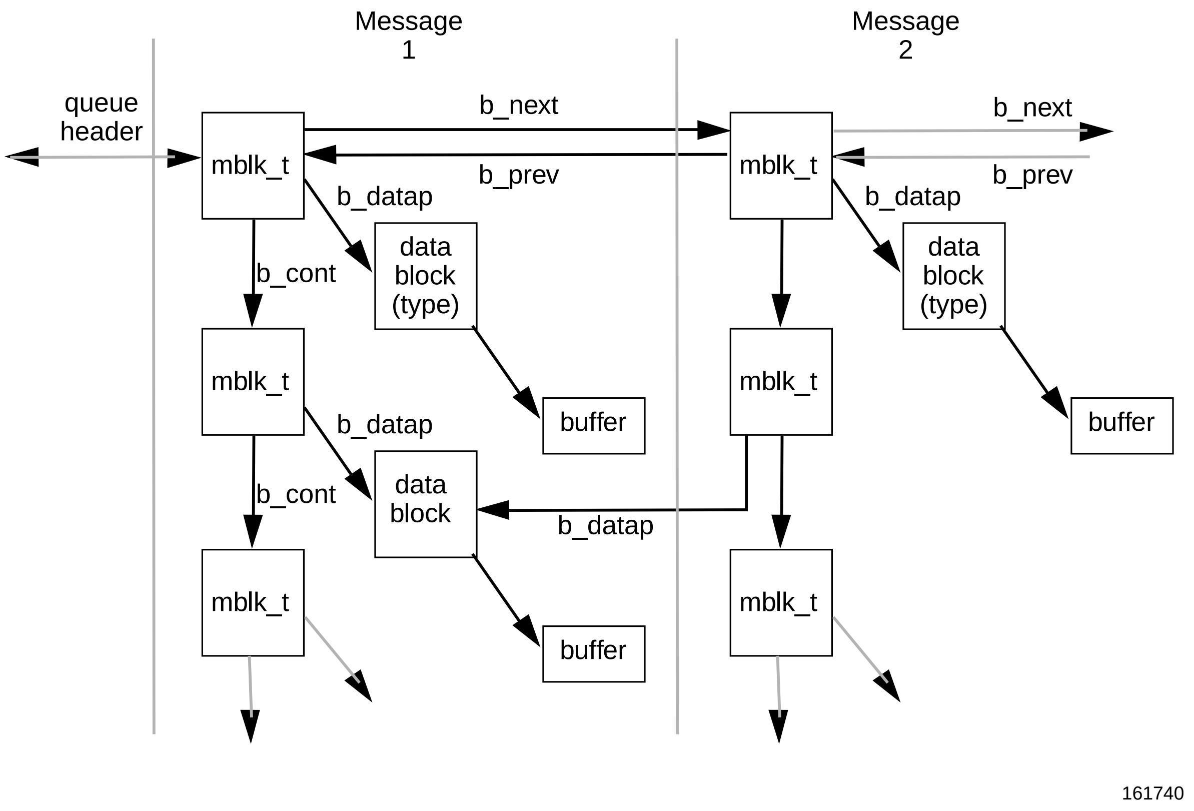


Figure 5-1. Message Form and Linkage

A message may occur singly, as when it is processed by a **`put`** procedure, or may be linked on the message queue in a queue, waiting to be processed by the **`service`** procedure. Message 2, as shown in Figure 5-1, links to Message 1.

Note that a data block in Message 1 is shared between Message 1 and another message. Multiple message blocks can point to the same data block to conserve storage and to avoid copying overhead. For example, the same data block, with associated buffer, may be referenced in two messages, from separate modules that use separate protocol levels. Figure 5-2 illustrates the concept, but data blocks typically are not shared by messages on the same queue. The buffer can be retransmitted, if required, because of errors or timeouts, from either protocol level without replicating the data. The **`dupmsg`** utility routine does data block sharing. See **`dupmsg(D3)`**. STREAMS maintains a count of the message blocks sharing a data block in the `db_ref` field.

STREAMS provides utility routines, specified in the *Device Driver Reference*, to assist in managing messages and message queues, and to assist in other areas of module and driver development. A utility routine should always be used when operating on a message queue or accessing the message storage pool. If messages are manipulated on the queue without using the STREAMS utilities, the message ordering may become confused and lead to inconsistent results.

CAUTION

If you do not use the STREAMS utilities as they are defined by the Driver-Kernel Interface, the system may panic or deadlock. Non-Driver-Kernel Interface drivers are not supported.

Sending/Receiving Messages

Most message types can be generated by modules and drivers. A few are reserved for the Stream head. The most commonly used messages are `M_DATA`, `M_PROTO`, and `M_PCPROTO`. These messages can also be passed between a process and the topmost module in a Stream with the same message boundary alignment maintained on both sides of the kernel. This allows a user process to function, to some degree, as a module above the Stream and maintain a service interface. `M_PROTO` and `M_PCPROTO` messages are intended to carry service interface information among modules, drivers, and user processes. Some message types can only be used within a Stream and cannot be sent or received from the user level.

Modules and drivers do not interact directly with any system calls except **`open(2)`** and **`close(2)`**. The Stream head handles all message translation and passing between user processes and STREAMS components. Message transfer between processes and the Stream head can occur in different forms. For example, `M_DATA` and `M_PROTO` messages can be transferred in their direct form by the **`getmsg(2)`** and **`putmsg(2)`** system calls. Alternatively, **`write(2)`** causes one or more `M_DATA` messages to be created from the data buffer supplied in the call. `M_DATA` messages received at the Stream head are consumed by **`read(2)`** and copied into the user buffer. As another example, `M_SIG` causes the Stream head to send a signal to a process.

Any module or driver can send any message in either direction on a Stream. However, based on their intended use in STREAMS and their treatment by the Stream head, certain messages can be categorized as upstream, downstream, or bidirectional. `M_DATA`, `M_PROTO`, or `M_PCPROTO` messages, for example, can be sent in both directions. Other message types are intended to be sent upstream to be processed only by the Stream head. Messages intended to be sent downstream are silently discarded if received by the Stream head.

STREAMS enables modules to create messages and pass them to neighboring modules. However, the **`read(2)`** and **`write(2)`** system calls are not enough to enable a user process to generate and receive all such messages. First, **`read`** and **`write`** are byte-stream oriented with no concept of message boundaries. To support service interfaces, the message boundary of each service primitive must be preserved so that the beginning and end of each primitive can be located. Also, **`read`** and **`write`** offer only one buffer to the user for transmitting and receiving STREAMS messages. If control information and data were placed in a single buffer, the user would have to parse the contents of the buffer to separate the data from the control information.

The **`putmsg`** system call enables a user to create messages and send them downstream. The user supplies the contents of the control and data parts of the message in two separate buffers. The **`getmsg`** system call retrieves `M_DATA` or `M_PROTO` messages from a Stream and places the contents into two user buffers.

The format of **`putmsg`** is as follows:

int **putmsg**(int *fd*, struct strbuf **ctlptr*, struct strbuf **dataptr*, int *flags*)

where *fd* identifies the Stream to which the message is passed, *ctlptr* and *dataptr* identify the control and data parts of the message, and *flags* may be used to specify that a high-priority message (`M_PCPROTO`) should be sent. When a control part is present, setting *flags* to 0 generates an `M_PROTO` message. If *flags* is set to `RS_HIPRI`, an `M_PCPROTO` message is generated.

NOTE

> The Stream head guarantees that the control part of a message generated by **`putmsg(2)`** is at least 64 bytes in length. This promotes reusability of the buffer. When the buffer is a reasonable size, modules and drivers may reuse the buffer for other headers.

The `strbuf` structure is used to describe the control and data parts of a message, and has the following format:

```
struct strbuf {
    int  maxlen;/* maximum buffer length */
    int  len;     /* length of data */
    char *buf;    /* pointer to buffer */
}
```

where `buf` points to a buffer containing the data, `len` specifies the number of bytes of data in the buffer, and `maxlen` specifies the maximum number of bytes the given buffer can hold, and is only significant when retrieving information into the buffer using **`getmsg`**.

The **`getmsg`** system call retrieves `M_DATA`, `M_PROTO`, or `M_PCPROTO` messages available at the Stream head, and has the following format:

`int` **`getmsg`**`(int` *fd*`, struct strbuf *`*ctlptr*`, struct strbuf *`*dataptr*`, int *`*flagsp*`)`

The arguments to **`getmsg`** are the same as those of **`putmsg`** except that the *flagsp* parameter is a pointer to an `int`.

`putpmsg` and **`getpmsg`** (see **`putmsg(2)`** and **`getmsg(2)`**) support multiple bands of data flow. They are analogous to the system calls **`putmsg`** and **`getmsg`**. The extra parameter is the priority band of the message.

`putpmsg` has the following interface:

```
int putpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, intband, int flags)
```

The parameter *band* is the priority band of the message to put downstream. The valid values for *flags* are `MSG_HIPRI` and `MSG_BAND`. `MSG_BAND` and `MSG_HIPRI` are mutually exclusive. `MSG_HIPRI` generates a high-priority message (`M_PCPROTO`) and *band* is ignored. `MSG_BAND` causes an `M_PROTO` or `M_DATA` message to be generated and sent down the priority band specified by *band*. The valid range for *band* is from 0 to 255, inclusive.

The call

```
    putpmsg(fd, ctlptr, dataptr, 0, MSG_BAND);
```

is equivalent to the system call

```
    putmsg(fd, ctlptr, dataptr, 0);
```

and the call

```
    putpmsg(fd, ctlptr, dataptr, 0, MSG_HIPRI);
```

is equivalent to the system call

```
    putmsg(fd, ctlptr, dataptr, RS_HIPRI);
```

If `MSG_HIPRI` is set and *band* is nonzero, **`putpmsg`** fails with `EINVAL`.

`getpmsg` has the following format:

```
int getpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,int *bandp, int *flagsp)
```

where *bandp* is the priority band of the message. This system call retrieves a message from the Stream. If **flagsp* is set to `MSG_HIPRI`, **`getpmsg`** attempts to retrieve a high-priority message. If `MSG_BAND` is set, **`getpmsg`** tries to retrieve a message from priority band **bandp* or higher. If `MSG_ANY` is set, the first message on the Stream head read queue is retrieved. These three flags (`MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`) are mutually exclusive. On return, if a high priority message was retrieved, **flagsp* is set to `MSG_HIPRI` and **bandp* is set to `0`. Otherwise, **flagsp* is set to `MSG_BAND` and **bandp* is set to the band of the message retrieved.

The call

```
    int band = 0;
    int flags = MSG_ANY;

    getpmsg(fd, ctlptr, dataptr, &band, &flags);
```

is equivalent to

```
    int flags = 0;

    getmsg(fd, ctlptr, dataptr, &flags);
```

If `MSG_HIPRI` is set and **bandp* is nonzero, **`getpmsg`** fails with `EINVAL`.

Control of Stream Head Processing

The `M_SETOPTS` message allows a driver or module to exercise control over certain Stream head processing. An `M_SETOPTS` can be sent upstream at any time. The Stream head responds to the message by altering the processing associated with certain system calls. The options to be modified are specified by the contents of the **`stroptions`** structure contained in the message. See the *Device Driver Reference* for more information.

Six Stream head characteristics can be modified. Four characteristics correspond to fields contained in `queue` (minimum/maximum) packet sizes and high-/low- water marks). The other two are discussed here.

Read Options

The value for read options (`so_readopt`) corresponds to two sets of three modes a user can set using the **`I_SRDOPT ioctl`** (see **`streamio(7)`**) call. The first set deals with data and message boundaries:

byte-stream (`RNORM`)
: The **`read(2)`** call completes when the byte count is satisfied, the Stream head read queue becomes empty, or a zero length message is encountered. In the last case, the zero length message is put back on the queue. A subsequent `read` returns `0` bytes.

message non-discard (`RMSGN`)
: The **`read(2)`** call completes when the byte count is satisfied or at a message boundary, whichever comes first. Any data remaining in the message are put back on the Stream head read queue.

message discard (`RMSGD`)
: The **`read(2)`** call completes when the byte count is satisfied or at a message boundary. Any data remaining in the message are discarded.

Byte-stream mode nearly models pipe data transfer. Message non-discard mode nearly models a TTY in canonical mode.

The second set deals with the treatment of protocol messages by the **`read(2)`** system call:

normal protocol (`RPROTNORM`)
: The **`read(2)`** call fails with `EBADMSG` if an `M_PROTO` or `M_PCPROTO` message is at the front of the Stream head read queue. This is the default operation protocol.

protocol discard (`RPROTDIS`)
: The **`read(2)`** call discards any `M_PROTO` or `M_PCPROTO` blocks in a message, delivering the `M_DATA` blocks to the user.

protocol data (`RPROTDAT`)
: The **`read(2)`** call converts the `M_PROTO` and `M_PCPROTO` message blocks to `M_DATA` blocks, treating the entire message as data.

Write Offset

The value for write offset (`so_wroff`) is a hook to allow more efficient data handling. It works as follows: In every data message generated by a **`write(2)`** system call and in the first `M_DATA` block of the data portion of every message generated by a **`putmsg(2)`** call, the Stream head leaves `so_wroff` bytes of space at the beginning of the message block. Expressed as a C language construct:

```
    bp->b_rptr = bp->b_datap->db_base + write offset
```

The write offset value must be smaller than the maximum STREAMS message size, `STRMSGSZ`. In certain cases (for example, if a buffer large enough to hold the offset+data is not currently available), the write offset might not be included in the block. To handle all possibilities, modules and drivers should not assume that the offset exists in a message, but should always check the message.

The intended use of write offset is to leave room for a module or a driver to place a protocol header before user data in the message instead of allocating and prepending a separate message.

Message Queues and Priorities

Message queues grow when the STREAMS scheduler is delayed from calling a **`service`** procedure because of system activity, or when the procedure is blocked by flow control. When called by the scheduler the **`service`** procedure processes enqueued messages in a FIFO manner. However, expedited data support and certain conditions require that associated messages (for example, an `M_ERROR`) reach their Stream destination as rapidly as possible. This is done by associating priorities with the messages. These priorities imply a certain ordering of messages on the queue as shown in Figure 5-2. Each message has a priority band associated with it. Ordinary messages have a priority of zero. High-priority messages are high priority by nature of their message type. Their priority band is ignored. By convention, they are not affected by flow control. The **`putq`** utility routine places high-priority messages at the head of the message queue followed by priority band messages (expedited data) and ordinary messages.

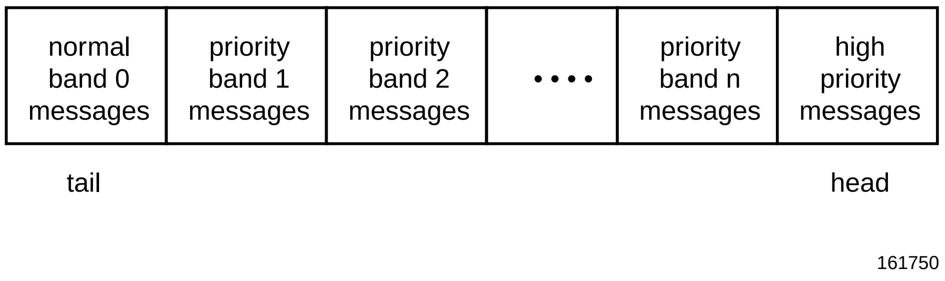


Figure 5-2. Message Ordering on a Queue

When a message is queued, it is placed after the messages of the same priority already on the queue (that is, FIFO within their order of queueing). This affects the flow control parameters associated with the band of the same priority. Message priorities range from 0 (normal) to 255 (highest). This provides up to 256 bands of message flow within a Stream. Expedited data can be implemented with one extra band of flow (priority band 1) of data. This is shown in Figure 5-3.

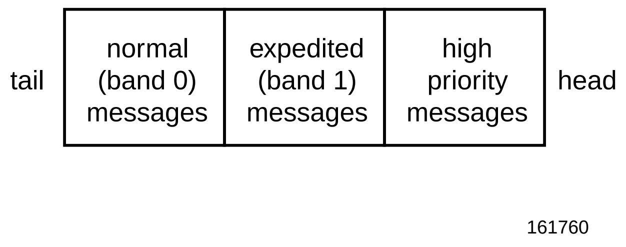


Figure 5-3. Message Ordering with One Priority Band

High-priority messages are not subject to flow control. When they are queued by **`putq`**, the associated queue is always scheduled (in the same way as any queue; following all other queues currently scheduled). When the **`service`** procedure is called by the scheduler, the procedure uses **`getq`** to retrieve the first message on queue, which will be a high-priority message, if present. **`service`** procedures must be implemented to act on high-priority messages immediately. The above mechanisms—priority message queueing, absence of flow control, and immediate processing by a procedure—result in rapid transport of high-priority messages between the originating and destination components in the Stream.

The following routines aid users in controlling each priority band of data flow:

- **`flushband`**,
- **`bcanputnext`**
- **`strqget`**
- **`strqset`**.

`flushband` is discussed in the section titled “Flush Handling,” and **`bcanputnext`** is discussed in the section titled “Flow Control.” The *Device Driver Reference* also has a description of these routines.

The **`strqget`** routine allows modules and drivers to obtain information about a queue or particular band of the queue. This insulates the STREAMS data structures from the modules and drivers. The format of the routine is:

```
int strqget(queue_t *q, qfields_t what, unsigned char pri, long *valp)
```

The information is returned in the `long` referenced by *valp*. The fields that can be obtained are defined by Screen 5-1:

```
typedef enum qfields {
        QHIWAT  = 0,  /* q_hiwat or qb_hiwat */
        QLOWAT  = 1,  /* q_lowat or qb_lowat */
        QMAXPSZ = 2,  /* q_maxpsz */
        QMINPSZ = 3,  /* q_minpsz */
        QCOUNT  = 4,  /* q_count or qb_count */
        QFIRST  = 5,  /* q_first or qb_first */
        QLAST   = 6,  /* q_last or qb_last */
        QFLAG   = 7,  /* q_flag or qb_flag */
        QBAD    = 8
} qfields_t;
```

Screen 5-1. Obtained Fields

This routine returns 0 on success and an error number on failure.

The routine **`strqset`** allows modules and drivers to change information about a queue or particular band of the queue. This also insulates the STREAMS data structures from the modules and drivers. Its format is

`int` **`strqset`**`(queue_t *`*q*`, qfields_t` *what*`, unsigned char` *pri*`, long` *val*`)`

The updated information is provided by *val*. **`strqset`** returns 0 on success and an error number on failure. If the field is intended to be read-only, then the error `EPERM` is returned and the field is left unchanged. The following fields are currently read-only: `QCOUNT`, `QFIRST`, `QLAST`, and `QFLAG`.

Note that the **`strqget`** and **`strqset`** routines must be bracketed by the **`freezestr(D3)`** and **`unfreezestr(D3)`** routines.

The **`ioctl`**s **`I_FLUSHBAND`**, **`I_CKBAND`**, **`I_GETBAND`**, **`I_CANPUT`**, and **`I_ATMARK`** support multiple bands of data flow. The **`ioctl I_FLUSHBAND`** allows a user to flush a particular band of messages. It is discussed in more detail in the section titled "Flush Handling." The **`ioctl I_CKBAND`** allows a user to check if a message of a given priority exists on the Stream head read queue. Its interface is

`ioctl``(`*fd*`,` **`I_CKBAND`**`,` *pri*`);`

This returns `1` if a message of priority *pri* exists on the Stream head read queue and `0` if no message of priority *pri* exists. If an error occurs, `-1` is returned. Note that *pri* should be of type `int`.

The **`ioctl I_GETBAND`** allows a user to check the priority of the first message on the Stream head read queue. The interface is

`ioctl``(`*fd*`,` **`I_GETBAND`**`,` *prip*`);`

This results in the integer referenced by *prip* being set to the priority band of the message on the front of the Stream head read queue.

The **`ioctl I_CANPUT`** allows a user to check if a certain band is writable. Its interface is

```
ioctl(fd, I_CANPUT, pri);
```

The return value is `0` if the priority band *pri* is flow controlled, `1` if the band is writable, and `-1` on error.

The field `b_flag` of the `msgb` structure can have a flag `MSGMARK` that allows a module or driver to *mark* a message. This is used to support TCP's (Transport Control Protocol) ability to show the user the last byte of out-of-band data. Once marked, a message sent to the Stream head causes the Stream head to remember the message. A user may check to see if the message on the front of its Stream head read queue is marked with the **`I_ATMARK`** **`ioctl`**. If a user is reading data from the Stream head and there are multiple messages on the read queue, and one of those messages is marked, the **`read(2)`** terminates when it reaches the marked message and returns the data only up to that marked message. The rest of the data may be obtained with successive reads.

The **`ioctl I_ATMARK`** has the following format:

```
ioctl(fd, I_ATMARK, flag);
```

where *flag* may be either `ANYMARK` or `LASTMARK`. `ANYMARK` indicates that the user merely wants to check if the message is marked. `LASTMARK` indicates that the user wants to see if the message is the only one marked on the queue. If the test succeeds, `1` is returned. On failure, `0` is returned. If an error occurs, `-1` is returned.

queue Structure

`service` procedures, message queues, message priority, and basic flow control are all intertwined in STREAMS. A queue generally does not use its message queue if there is no **`service`** procedure in the queue. The function of a **`service`** procedure is to process messages on its queue. Message priority and flow control are associated with message queues.

The operation of a queue revolves around the `queue` structure. See **`queue(D4DK)`** for details.

Queues are always allocated in pairs (read and write); one queue pair per module, driver, or Stream head. A queue contains a linked list of messages. When a `queue` pair is allocated, the following fields are initialized by STREAMS:

- `q_qinfo` - from `streamtab`
- `q_minpsz`, `q_maxpsz`, `q_hiwat`, `q_lowat` - from `module_info`

Copying values from `module_info` allows them to be changed in the `queue` without modifying the `streamtab` and `module_info` values.

`q_count` and `qb_count` are used in flow control calculations and represent the number of bytes in the various bands on the queue.

Using queue Information

Modules and drivers should use STREAMS utility routines to alter `q_first`, `q_last`, `q_count`, and `q_flag`. See the *Device Driver Reference* for more information.

Modules and drivers can change `q_ptr`. Modules and drivers can read but should not change `q_qinfo`, `q_bandp`, and `q_nband`.

Modules and drivers need locks for their private data structures (just as the STREAMS code protects the `q_next` pointer, for example).

See **`queue(D4DK)`** for a list of flags that you can test.

qband Structure

The queue flow information for each band is contained in a `qband` structure. This structure is not visible to a module/driver, although some information in it may be read and written using **`strqget`** and **`strqset`**.

`qband` includes a field analogous to the `queue`'s `q_count` field. However, the field only applies to the messages on the queue in the band of data flow represented by the corresponding `qband` structure. (In contrast, `q_count` only contains information regarding normal and high-priority messages.)

Each band has a separate high- and low-water mark. These are initially set to the `queue`'s `q_hiwat` and `q_lowat` respectively. Modules and drivers may change these values if desired through the **`strqset`** function.

The `qband` structures are not preallocated per queue. They are allocated when a message with a priority greater than zero is placed on the queue by **`putq`**, **`putbq`**, or **`insq`**. Because band allocation can fail, these routines return 0 on failure and 1 on success. Once a `qband` structure is allocated, it remains associated with the queue until the queue is freed. **`strqset`** and **`strqget`** will cause `qband` allocation to occur.

Using qband Information

Use the STREAMS utility routines when manipulating the fields in the `qband` structure. Use the routines **`strqset`** and **`strqget`** to access band information.

Message Processing

`put` procedures are generally required in pushable modules. **`service`** procedures are optional. If the **`put`** routine enqueues messages, you need a corresponding **`service`** routine to handle the enqueued messages. If the **`put`** routine does not enqueue messages, you do not need the **`service`** routine.

The general processing flow when both procedures are present is as follows:

1. A message is received by the **`put`** procedure in a queue, where some processing may be performed on the message.

2. The **put** procedure places the message on the queue with the **putq** utility routine for the **service** procedure to process further at some later time.
3. **putq** places the message on the queue based on its priority.
4. **putq** makes the queue ready for execution by the STREAMS scheduler.
5. After some indeterminate delay (intended to be short), the STREAMS scheduler calls the **service** procedure.
6. The **service** procedure gets the first message from the message queue with the **getq** utility.
7. The **service** procedure processes the message and passes it to the **put** procedure of the next queue with **putnext**.
8. The **service** procedure gets the next message and processes it.

This processing continues until the queue is empty (**getq** does not return a message) or flow control blocks further processing. The **service** procedure returns to the caller.

NOTE

A **service** or **put** procedure must never sleep since it has no user context. It must always return to its caller.

If no processing is required in the **put** procedure, the procedure does not have to be explicitly declared. However, **putq** can be placed in the `qinit` structure declaration for the appropriate queue side to queue the message for the **service** procedure, for example:

```
static struct qinit winit = { putq, modwsrv, ...... };
```

Typically, **put** procedures will, at a minimum, process high-priority messages to avoid queueing them. If `M_FLUSH` messages are queued there is a danger that a message queued after the `M_FLUSH` will be discarded when the `M_FLUSH` is processed.

The key attribute of a **service** procedure in the STREAMS architecture is delayed processing. When a **service** procedure is used in a module, the module developer is implying that there are other, more time-sensitive activities to be performed elsewhere in this Stream, in other Streams, or in the system in general. The presence of a **service** procedure is mandatory if the flow control mechanism is to be used by the queue.

The delay for STREAMS to call a **service** procedure varies with implementation and system activity.

If a module or driver wishes to recognize priority bands, the **service** procedure is written to the following algorithm:

```
.
.
while ((bp = getq(q)) != NULL) {
    if (pcmsg(bp->b_datap->db_type)) {
           putnext(q, bp);
    } else if (bcanputnext(q, bp->b_band)) {
               putnext(q, bp);
    } else {
       putbq(q, bp);
       return;
    }
}
.
.
```

NOTE

In this example, a race condition exists on a multiprocessor system between **bcanputnext** and **putnext**. By the time **putnext** is called, the destination queue may be full, potentially causing the high water mark to be exceeded. Although the queue may be full, the amount of “overwrite” bounded, and is therefore not usually a problem.

Flow Control

The STREAMS flow control mechanism is voluntary and operates between the two nearest queues in a Stream containing **service** procedures (see Figure 5-4). Messages are generally held on a queue only if a **service** procedure is present in the associated queue. Flow control is applied per band. Each band has its own high- and low-water marks.

Messages accumulate on a queue when the queue's **service** procedure processing does not keep pace with the message arrival rate, or when the procedure is blocked from placing its messages on the following Stream component by the flow control mechanism. Pushable modules contain independent upstream and downstream limits. The Stream head contains a preset upstream limit (which can be modified by a special message sent from downstream) and a driver may contain a downstream limit.

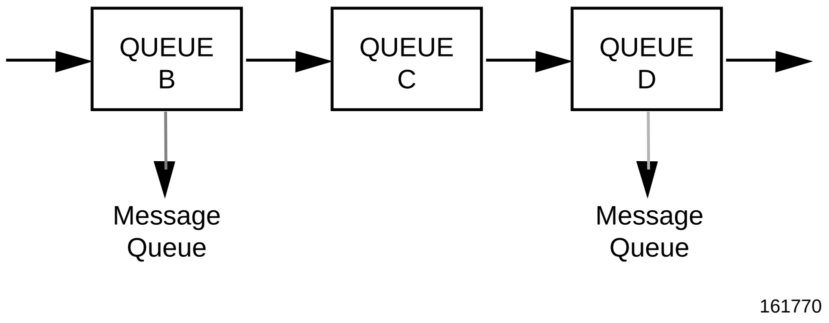


Figure 5-4. Flow Control

Flow control operates as follows:

1. Each time a STREAMS message handling routine (for example, **`putq`**) adds or removes a message from a message queue, the limits are checked. STREAMS calculates the total size of all message blocks (`bp->b_wptr - bp->b_rptr`) on the message queue.

2. The total is compared to the queue high-water and low-water values. If the total exceeds the high-water value, an internal full indicator is set for the queue. The operation of the **`service`** procedure in this queue is not affected if the indicator is set, and the **`service`** procedure continues to be scheduled.

3. The next part of flow control processing occurs in the nearest preceding queue that contains a **`service`** procedure. In Figure 5-4, if Queue D is full and Queue C has no **`service`** procedure, then Queue B is the nearest preceding queue.

4. The **`service`** procedure in Queue B uses a STREAMS utility routine to see if a queue ahead is marked full. If messages cannot be sent, the scheduler blocks the **`service`** procedure in Queue B from further execution. Queue B remains blocked until the low water mark of the full queue, Queue D, is reached.

5. While Queue B is blocked, any messages except high-priority messages arriving at Queue B will accumulate on its message queue.

NOTE

High-priority messages are not subject to flow control.

Eventually, Queue B may reach a full state and the full condition will propagate back to the previous module in the Stream.

6. When the **`service`** procedure processing on Queue D causes the message block total to fall below the high-water mark, the full indicator is turned off. When the message block total falls below the low-water mark, STREAMS automatically schedules the nearest preceding blocked queue (Queue B in this example), to restart processing. This automatic scheduling is known as back-enabling a queue.

Modules and drivers need to observe the message priority. High-priority messages, determined by the type of the first block in the message, are not subject to flow control. They are processed immediately and forwarded, as appropriate.

For ordinary messages, flow control must be tested before any processing is performed. The **`canputnext`** utility determines if the forward path from the queue is blocked by flow control.

This is the general flow control processing of ordinary messages:

1. Retrieve the message at the head of the queue with **`getq`**.
2. Determine if the message type is high priority and not to be processed here.
3. If so, pass the message to the **`put`** procedure of the following queue with **`putnext`**.
4. Use **`canputnext`** to determine if messages can be sent onward.
5. If messages should not be forwarded, put the message back on the queue with **`putbq`** and return from the procedure.
6. Otherwise, process the message.

The canonical representation of this processing within a **`service`** procedure is as follows:

```
while (getq != NULL)
        if (high priority message || no flow control)
                process message
                putnext
        else
                putbq
                return
```

Expedited data have their own flow control with the same general processing as that of ordinary messages. **`bcanputnext`** provides modules and drivers with a way to test flow control in the given priority band. It returns 1 if a message of the given priority can be placed on the queue, returns 0 if the priority band is flow controlled, and if the band does not yet exist on the queue in question, the routine returns 1.

Banded data has separate flow control. In other words, bands 1 through 255 operate totally independently. Any band greater than or equal to band 1, when flow controlled, will stop band 0 data (normal data).

Note that the call `bcanputnext(q, 0)` is equivalent to the call `canputnext(q)`.

NOTE

A **service** procedure must process all messages on its queue unless flow control prevents this.

A **service** procedure continues processing messages from its queue until getq returned NULL. When an ordinary message is enqueued by **putq**, **putq** causes the **service** procedure to be scheduled only if the queue was previously empty, and a previous **getq** call returns NULL (that is, the QWANTR flag is set). If there are messages on the queue, **putq** presumes the **service** procedure is blocked by flow control and the procedure is automatically rescheduled by STREAMS when the block is removed. If the **service** procedure cannot complete processing as a result of conditions other than flow control (for example, no buffers), it must ensure it will return later (for example, by use of the **bufcall** utility routine) or it must discard all messages on the queue. If this is not done, STREAMS never schedules the **service** procedure to be run unless the queue's **put** procedure enqueues a priority message with **putq**.

NOTE

High-priority messages are discarded only if there is already a high-priority message on the Stream head read queue. Only one high-priority message can be present on the Stream head read queue at any time.

putbq replaces messages at the beginning of the appropriate section of the message queue by their priority. This might not be the same position at which the message was retrieved by the preceding **getq**. A later **getq** might return a different message.

putq only looks at the priority band in the first message. If a high-priority message is passed to **putq** with a nonzero b_band value, b_band is reset to 0 before placing the message on the queue. If the message is passed to **putq** with a b_band value that is greater than the number of qband structures associated with the queue, **putq** tries to allocate a new qband structure for each band up to and including the band of the message.

The above also applies to **putbq** and **insq**. If an attempt is made to insert a message out of order in a queue by insq, the message is not inserted and the routine fails.

putq will not schedule a queue if **noenable**(*q*) has been previously called for this queue. **noenable** instructs **putq** to enqueue the message when called by this queue, but not to schedule the **service** procedure. **noenable** does not prevent the queue from being scheduled by a flow control back-enable. The inverse of **noenable** is **enableok(q)**.

Driver upstream flow control is explained next as an example. Although device drivers typically discard input when they are unable to send it to a user process, STREAMS allows driver read-side flow control, possibly for handling temporary upstream blockages, through a driver read **service** procedure that is disabled during the driver **open** with **noenable**. If the driver input interrupt routine determines messages can be sent upstream, it sends the message with **putnext**. Otherwise, it calls **putq** to queue the message. The message waits on the message queue (possibly with queue length checked when new messages are enqueued by the interrupt routine) until the upstream queue becomes unblocked. When the blockage abates, STREAMS back-enables the driver read **`service`** procedure, which then sends the messages upstream using the mechanisms described previously. This is similar to **`looprsrv`** (see the section titled "Loop-around Driver" where the **`service`** procedure is present only for flow control.

`qenable`, another flow control utility, allows a module or driver to cause one of its queues to be scheduled. **`qenable`** might also be used when a module or driver wants to delay message processing for some reason. An example is a buffer module that gathers messages in its message queue and forwards them as a single, larger message. This module uses **`noenable`** to inhibit its **`service`** procedure and queues messages with its **`put`** procedure until a certain byte count or "in queue" time has been reached. When either condition is met, the module calls **`qenable`** to cause its **`service`** procedure to run.

Another example is a communication line discipline module that implements end-to-end (that is, to a remote system) flow control. Outbound data is held on the write-side message queue until the read-side receives a transmit window from the remote end of the network.

NOTE

STREAMS routines are called at different priority levels. Interrupt routines are called at the interrupt priority of the interrupting device. **`service`** routines are called with interrupts enabled (hence, **`service`** routines for STREAMS drivers can be interrupted by their own interrupt routines). Write side **`put`** procedures may also be interrupted by their own interrupt routines.

Service Interfaces

STREAMS can implement a service interface between any two components in a Stream, and between a user process and the topmost module in the Stream. A service interface is defined at the boundary between a service user and a service provider. A service interface is a set of primitives and the rules that define a service and the allowable state transitions that result as these primitives are passed between the user and the provider. These rules are typically represented by a state machine. In STREAMS, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in `M_PROTO` and `M_PCPROTO` messages.

`PROTO` messages (`M_PROTO` and `M_PCPROTO`) can be multiblock, with the second through last blocks of type `M_DATA`. The first block in a `PROTO` message contains the control part of the primitive in a form agreed on by the user and provider. The block is not intended to carry protocol headers. (Although its use is not recommended, upstream `PROTO` messages can have multiple `PROTO` blocks at the start of the message. **`getmsg(2)`** compacts the blocks into a single control part when sending to a user process). The `M_DATA` block(s) contains any data part associated with the primitive. The data part may be processed in a module that receives it, or it may be sent to the next Stream component along with any data generated by the module. The contents of `PROTO` messages and their allowable sequences are determined by the service interface.

`PROTO` messages can be sent bidirectionally (upstream and downstream) on a Stream and between a Stream and a user process. **`putmsg(2)`** and **`getmsg(2)`** system calls are analogous, respectively, to **`write(2)`** and **`read(2)`** except that the former allow both data and control parts to be (separately) passed, and they retain the message boundaries across the user-Stream interface. **`putmsg(2)`** and **`getmsg(2)`** separately copy the control part (`M_PROTO` or `M_PCPROTO` block) and data part (`M_DATA` blocks) between the Stream and user process.

An `M_PCPROTO` message is normally used to acknowledge primitives composed of other messages. `M_PCPROTO` ensures that the acknowledgment reaches the service user before any other message. If the service user is a user process, the Stream head only stores a single `M_PCPROTO` message, and discards subsequent `M_PCPROTO` messages until the first one is read with **`getmsg(2)`**.

A STREAMS message format has been defined to simplify the design of service interfaces. System calls, **`getmsg(2)`** and **`putmsg(2)`**, are available for sending messages downstream and receiving messages that are available at the Stream head.

This section describes the system calls **`getmsg`** and **`putmsg`** in the context of a service interface example. First, a brief overview of STREAMS service interfaces is presented.

Service Interface Benefits

A principal advantage of the STREAMS mechanism is its modularity. From the user level, kernel-resident modules can be dynamically interconnected to implement any reasonable processing sequence. This modularity reflects the layering characteristics of contemporary network architectures.

One benefit of modularity is the ability to interchange modules of like functions. For example, two distinct transport protocols, implemented as STREAMS modules, may provide a common set of services. An application or higher layer protocol that requires those services can use either module. This ability to substitute modules enables user programs and higher level protocols to be independent of the underlying protocols and physical communication media.

Each STREAMS module provides a set of processing functions, or services, and an interface to those services. The service interface of a module defines the interaction between that module and any neighboring modules, and is a necessary component for providing module substitution. By creating a well-defined service interface, applications and STREAMS modules can interact with any module that supports that interface, as shown in Figure 5-5.

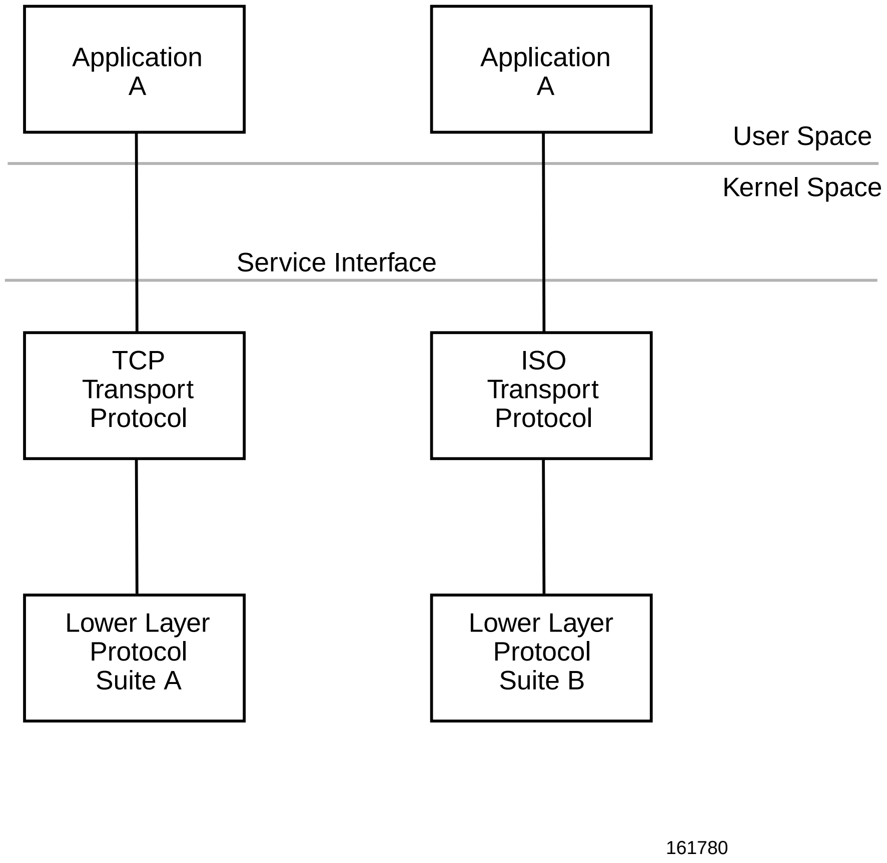


Figure 5-5. Protocol Substitution

By defining a service interface through which applications interact with a transport protocol, it is possible to substitute a different protocol below that service interface in a way completely transparent to the application. In this example, the same application can run over the Transmission Control Protocol (TCP) and the ISO transport protocol. Of course, the service interface must define a set of services common to both protocols.

The three components of any service interface are the service user, the service provider, and the service interface itself, as shown in Figure 5-6.

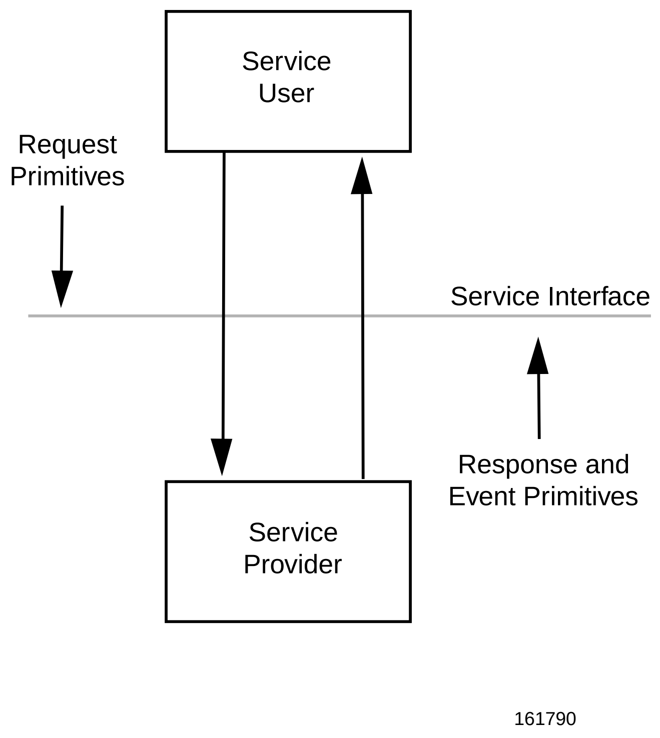


Figure 5-6. Service Interface

Typically, a user makes a request of a service provider using some well-defined service primitive. Responses and event indications are also passed from the provider to the user using service primitives.

Each service interface primitive is a distinct STREAMS message that has two parts: a control part and a data part. The control part contains information that identifies the primitive and includes all necessary parameters. The data part contains user data associated with that primitive.

An example of a service interface primitive is a transport protocol connect request. This primitive requests the transport protocol service provider to establish a connection with another transport user. The parameters associated with this primitive may include a destination protocol address and specific protocol options to be associated with that connection. Some transport protocols also allow a user to send data with the connect request. A STREAMS message would be used to define this primitive. The control part would identify the primitive as a connect request and would include the protocol address and options. The data part would contain the associated user data.

Service Interface Library

The service interface library example presented in Screen 5-2 through Screen 5-7 includes four functions that enable a user to do the following:

- Establish a Stream to the service provider and bind a protocol address to the Stream. See Screen 5-2, Screen 5-3 and Screen 5-4.
- Send data to a remote user. See Screen 5-6.
- Receive data from a remote user. See Screen 5-7.
- Close the Stream connected to the provider. See Screen 5-5.

Screen 5-2 shows the structure and constant definitions required by the library. These typically will reside in a header file associated with the service interface.

```
/*
 * Primitives initiated by the service user.
 */
#define BIND_REQ      1   /* bind request */
#define UNITDATA_REQ  2   /* unitdata request */

/*
 * Primitives initiated by the service provider.
 */
#define OK_ACK        3   /* bind acknowledgment */
#define ERROR_ACK     4   /* error acknowledgment */
#define UNITDATA_IND  5   /* unitdata indication */

/*
 * The following structure definitions define the format of the
 * control part of the service interface message of the above
 * primitives.
 */

struct bind_req {     /* bind request */
    long PRIM_type;   /* always BIND_REQ */
    long BIND_addr;   /* addr to bind */
};

struct unitdata_req { /* unitdata request */
    long PRIM_type;   /* always UNITDATA_REQ */
    long DEST_addr;   /* destination addr */
};

struct ok_ack {       /* positive acknowledgment */
    long PRIM_type;   /* always OK_ACK */
};

struct error_ack {    /* error acknowledgment */
    long PRIM_type;   /* always ERROR_ACK */
    long UNIX_error;  /* UNIX system error code  */
};

struct unitdata_ind {/* unitdata indication */
    long PRIM_type;   /* always UNITDATA_IND */
    long SRC_addr;    /* source addr */
};

/* union of all primitives */

union primitives {
    long                 type;
    struct bind_req      bind_req;
    struct unitdata_req  unitdata_req;
    struct ok_ack        ok_ack;
    struct error_ack     error_ack;
    struct unitdata_ind  unitdata_ind;
};
```

Screen 5-2. Service Interface Library Example

```
/* header files needed by library */
#include <stropts.h>
#include <stdio.h>
#include <errno.h>
```

Five primitives have been defined. The first two represent requests from the service user to the service provider. These are as follows:

`BIND_REQ`	Asks the provider to bind a specified protocol address (that is, give it a name on the network.). It requires an acknowledgment from the provider to verify that the contents of the request are syntactically correct.
`UNITDATA_REQ`	Asks the provider to send data to the specified destination address. It does not require an acknowledgment from the provider.

The three other primitives represent acknowledgments of requests, or indications of incoming events, and are passed from the service provider to the service user. These are as follows:

`OK_ACK`	Informs the user that a previous bind request was received successfully by the service provider.
`ERROR_ACK`	Informs the user that a non-fatal error was found in the previous bind request. It indicates that no action was taken with the primitive that caused the error.
`UNITDATA_IND`	Indicates that data destined for the user have arrived.

The defined structures describe the contents of the control part of each service interface message passed between the service user and service provider. The first field of each control part defines the type of primitive being passed.

Accessing the Service Provider

The first routine presented, **`inter_open`**, opens the protocol driver device file specified by *path* and binds the protocol address contained in *addr* so that it may receive data. On success, the routine returns the file descriptor associated with the open Stream; on failure, it returns `-1` and sets `errno` to indicate the appropriate error value.

```
inter_open(char *path, int oflags, int addr)
{
    int fd;
    struct bind_req bind_req;
    struct strbuf ctlbuf;
    union primitives rcvbuf;
    struct error_ack *error_ack;
    int flags;

    if ((fd = open(path, oflags)) < 0)
        return(-1);

    /* send bind request msg down stream */

    bind_req.PRIM_type = BIND_REQ;
    bind_req.BIND_addr = addr;
    ctlbuf.len = sizeof(struct bind_req);
    ctlbuf.buf = (char *)&bind_req;

    if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {
        close(fd);
        return(-1);
    }
```

Screen 5-3. Accessing the Service Provider

After opening the protocol driver, **`inter_open`** packages a bind request message to send downstream. **`putmsg`** is called to send the request to the service provider. The bind request message contains a control part that holds a *bind_req* structure, but it has no data part. *ctlbuf* is a structure of type `strbuf`, and it is initialized with the primitive type and address. Notice that the `maxlen` field of `ctlbuf` is not set before calling **`putmsg`**, because **`putmsg`** ignores this field. The *dataptr* argument to **`putmsg`** is set to `NULL` to indicate that the message contains no data part. Also, the *flags* argument is 0, which specifies that the message is not a high-priority message.

After **`inter_open`** sends the bind request, it must wait for an acknowledgment from the service provider, as shown in Screen 5-4:

```
    /* wait for ack of request */

    ctlbuf.maxlen = sizeof(union primitives);
    ctlbuf.len = 0;
    ctlbuf.buf = (char *)&rcvbuf;
    flags = RS_HIPRI;

    if (getmsg(fd, &ctlbuf, NULL, &flags) < 0) {
        close(fd);
        return(-1);
    }

    /* did we get enough to determine type */
    if (ctlbuf.len < sizeof(long)) {
        close(fd);
        errno = EPROTO;
        return(-1);
    }

    /* switch on type (first long in rcvbuf) */
    switch(rcvbuf.type) {
        default:
            close(fd);
            errno = EPROTO;
            return(-1);

        case OK_ACK:
            return(fd);

        case ERROR_ACK:
            if (ctlbuf.len < sizeof(struct error_ack)) {
                close(fd);
                errno = EPROTO;
                return(-1);
            }
            error_ack = (struct error_ack *)&rcvbuf;
            close(fd);
            errno = error_ack->UNIX_error;
            return(-1);
    }
}
```

Screen 5-4. Acknowledgment from Service Provider

getmsg is called to retrieve the acknowledgment of the bind request. The acknowledgment message consists of a control part that contains either an `ok_ack` or `error_ack` structure, and no data part.

The acknowledgment primitives are defined as priority messages. Messages are queued in a FIFO sequence within their priority at the Stream head; high-priority messages are placed at the front of the Stream head queue followed by priority band messages and ordinary messages. The STREAMS mechanism allows only one high-priority message per Stream at the Stream head at one time; any further high-priority messages are freed until the message at the Stream head is processed. (Only one high priority message can be present on the Stream head read queue at any time.) High-priority messages are particularly suitable for acknowledging service requests when the acknowledgment should be placed ahead of any other messages at the Stream head.

Before calling **getmsg**, this routine must initialize the `strbuf` structure for the control part. *buf* should point to a buffer large enough to hold the expected control part, and *maxlen* must be set to show the maximum number of bytes this buffer can hold.

Because neither acknowledgment primitive contains a data part, the *dataptr* argument to **getmsg** is set to NULL. The *flagsp* argument points to an integer containing the value RS_HIPRI. This flag indicates that **getmsg** should wait for a STREAMS high-priority message before returning. It is set because we want to catch the acknowledgment primitives that are priority messages. Otherwise, if the flag is zero, the first message is taken. With RS_HIPRI set, even if a normal message is available, **getmsg** will block until a high-priority message arrives.

On return from **getmsg**, the *len* field is checked to ensure that the control part of the retrieved message is an appropriate size. The example then checks the primitive type and takes appropriate actions. An OK_ACK indicates a successful bind operation, and **inter_open** returns the file descriptor of the open Stream. An ERROR_ACK indicates a bind failure, and errno is set to identify the problem with the request.

Closing the Service Provider

The next routine in the service interface library example is **inter_close**, which closes the Stream to the service provider.

```
inter_close(int fd)
{
    close(fd);
}
```

Screen 5-5. Closing the Service Provider

The routine simply closes the given file descriptor. This routine causes the protocol driver to free any resources associated with that Stream. For example, the driver may unbind the protocol address that had previously been bound to that Stream, thereby freeing that address for use by some other service user.

Sending Data to the Service Provider

The third routine, **inter_snd**, passes data to the service provider for transmission to the user at the address specified in *addr.* The data to be transmitted are contained in the buffer pointed to by *buf* and contains *len* bytes. On successful completion, this routine returns the number of bytes of data passed to the service provider; on failure, it returns -1 and sets errno to an appropriate error value.

```
inter_snd(int fd, char *buf, int len, long addr)
{
    struct strbuf ctlbuf;
    struct strbuf databuf;
    struct unitdata_req unitdata_req;

    unitdata_req.PRIM_type = UNITDATA_REQ;
    unitdata_req.DEST_addr = addr;
    ctlbuf.len = sizeof(struct unitdata_req);
    ctlbuf.buf = (char *)&unitdata_req;
    databuf.len = len;
    databuf.buf = buf;

    if (putmsg(fd, &ctlbuf, &databuf, 0) < 0) {
        errno = EIO;
        return(-1);
    }

    return(len);
}
```

Screen 5-6. Sending Data

In this example, the data request primitive is packaged with both a control part and a data part. The control part contains a *unitdata_req* structure that identifies the primitive type and the destination address of the data. The data to be transmitted are placed in the data part of the request message.

Unlike the bind request, the data request primitive requires no acknowledgment from the service provider. In the example, this choice was made to minimize the overhead during data transfer. If the **`putmsg`** call succeeds, this routine assumes all is well and returns the number of bytes passed to the service provider.

Receiving Data from the Service Provider

The final routine in this example, **`inter_rcv`**, retrieves the next data. *buf* points to a buffer where the data should be stored, *len* shows the size of that buffer, and *addr* points to a long integer where the source address of the data will be placed. On successful completion, **`inter_rcv`** returns the number of bytes in the retrieved data; on failure, it returns `-1` and sets the appropriate PowerMAX OS System error value.

```
inter_rcv(int fd, char *buf, int len, long *addr)
    {
    struct strbuf ctlbuf;
    struct strbuf databuf;
    struct unitdata_ind unitdata_ind;
    int retval;
    int flagsp;

    ctlbuf.maxlen = sizeof(struct unitdata_ind);
    ctlbuf.len = 0;
    ctlbuf.buf = (char *)&unitdata_ind;
    databuf.maxlen = len;
    databuf.len = 0;
    databuf.buf = buf;
    flagsp = 0;

    if ((retval = getmsg(fd, &ctlbuf, &databuf, &flagsp)) < 0) {
        errno = EIO;
        return(-1);
    }
    if (retval) {
        errno = EIO;
        return(-1);
    }
    if (unitdata_ind.PRIM_type != UNITDATA_IND) {
        errno = EPROTO;
        return(-1);
    }
    *addr = unitdata_ind.SRC_addr;
    return(databuf.len);
}
```

Screen 5-7. Receiving Data

getmsg is called to retrieve the data indication primitive, where that primitive contains both a control and data part. The control part consists of a `unitdata_ind` structure that identifies the primitive type and the source address of the data sender. The data part contains the data itself.

In `ctlbuf`, `buf` must point to a buffer where the control information will be stored, and `maxlen` must be set to indicate the maximum size of that buffer. Similar initialization is done for `databuf`.

The integer pointed at by *flagsp* in the **getmsg** call is set to zero, indicating that the next message should be retrieved from the Stream head, regardless of its priority. Data will arrive in normal priority messages. If no message currently exists at the Stream head, **getmsg** will block until a message arrives.

The user's control and data buffers should be large enough to hold any incoming data. If both buffers are large enough, **getmsg** processes the data indication and return `0`, indicating that a full message was retrieved successfully. However, if either buffer is not large enough, **getmsg** only retrieves the part of the message that fits into each user buffer. The remainder of the message is saved for later retrieval (if in message non-discard mode), and a positive, non-zero value is returned to the user, `MORECTL` indicates that more control information is waiting for retrieval, `MOREDATA` indicates that more data is waiting for retrieval, and (`MORECTL | MOREDATA`) indicates that data from both parts of the message remains. In the example, if the user buffers are not large enough (that is, **getmsg** returns a positive, non-zero value), the function will set `errno` to EIO and fail.

The type of the primitive returned by **`getmsg`** is checked to make sure it is a data indication (`UNITDATA_IND` in the example). The source address is then set and the number of bytes of data is returned.

The example presented is a simplified service interface. The state transition rules for such an interface were not presented for the sake of brevity. The intent was to show typical uses of the **`putmsg`** and **`getmsg`** system calls. See **`putmsg(2)`** and **`getmsg(2)`** for further details. For simplicity, this example did not also consider expedited data.

Multiprocessor/Driver-Kernel Interface driver locks are used to protect against race conditions on multiprocessor systems with respect to the current state.

Module Service Interface

Screen 5-8 and Screen 5-9 show an example of part of a module that illustrates the concept of a service interface. The module implements a simple service interface and mirrors the service interface library example given earlier. The following rules pertain to service interfaces:

- Modules and drivers that support a service interface must act on all `PROTO` messages and not pass them through.
- Modules may be inserted between a service user and a service provider to manipulate the data part as it passes between them. However, these modules may not alter the contents of the control part (`PROTO` block, first message block) nor alter the boundaries of the control or data parts. The message blocks comprising the data part may be changed, but the message may not be split into separate messages nor combined with other messages.

In addition, modules and drivers must observe the rule that high-priority messages are not subject to flow control and forward them accordingly.

The service interface primitives are defined in the declarations as shown in Screen 5-8:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/errno.h>

/* Primitives initiated by the service user */

#define BIND_REQ1/* bind request */
#define UNITDATA_REQ2/* unitdata request */

/* Primitives initiated by the service provider */

#define OK_ACK3  /* bind acknowledgment */
#define ERROR_ACK4/* error acknowledgment */
#define UNITDATA_IND5/* unitdata indication */
/*
 * The following structures define the format of the
 * stream message block of the above primitives.
 */
struct bind_req {  /* bind request */
    long PRIM_type;/* always BIND_REQ */
    long BIND_addr;/* addr to bind*/
};
struct unitdata_req {/* unitdata request */
    long PRIM_type;/* always UNITDATA_REQ */
    long DEST_addr;/* dest addr */
};
struct ok_ack {/* ok acknowledgment */
    long PRIM_type;/* always OK_ACK */
};
struct error_ack {/* error acknowledgment */
    long PRIM_type;/* always ERROR_ACK */
    long UNIX_error;/* UNIX system error code */
};
struct unitdata_ind {/* unitdata indication */
    long PRIM_type;/* always UNITDATA_IND */
    long SRC_addr;/* source addr */
};
union primitives {/* union of all primitives */
    long  type;
    struct bind_req  bind_req;
    struct unitdata_req  unitdata_req;
    struct ok_ack  ok_ack;
    struct error_ack  error_ack;
    struct unitdata_ind  unitdata_ind;
};
struct dgproto {/* structure per minor device */
    short state;/* current provider state */
    long addr;/* net address */
    lck_t *lck;
};
/* Provider states */
#define IDLE   0
#define BOUND  1
```

Screen 5-8. Module Service Interface Declaration

In general, the `M_PROTO` or `M_PCPROTO` block is described by a data structure containing the service interface information. In this example, union primitives is that structure.

Two commands are recognized by the module:

`BIND_REQ`	Give this Stream a protocol address (that is, give it a name on the network). After a `BIND_REQ` is completed, data from other senders will find their way through the network to this particular Stream.
`UNITDATA_REQ`	Send data to the specified address.

Three messages are generated:

`OK_ACK`	A positive acknowledgment (*ack*) of `BIND_REQ`.
`ERROR_ACK`	A negative acknowledgment (*nak*) of `BIND_REQ`.
`UNITDATA_IND`	Data from the network have been received (this code is not shown).

The acknowledgment of a `BIND_REQ` informs the user that the request was syntactically correct (or incorrect if `ERROR_ACK`). The receipt of a `BIND_REQ` is acknowledged with an `M_PCPROTO` to ensure that the acknowledgment reaches the user before any other message. For example, a `UNITDATA_IND` could come through before the bind has completed, and the user would get confused.

The driver uses a per-minor device data structure, `dgproto`, which contains the following:

`state`	Current state of the service provider `IDLE` or `BOUND`
`addr`	Network address that has been bound to this Stream
`lck`	A spin lock to protect state information

It is assumed (though not shown) that the module open procedure sets the write queue `q_ptr` to point at the appropriate private data structure.

Service Interface Procedure

The write **`put`** procedure is shown in Screen 5-9:

```
int protowput(queue_t *q, mblk_t  *mp)
{
    union primitives *proto;
    struct dgproto *dgproto;
    int err;
    pl_t oldpri;

    dgproto = (struct dgproto *) q->q_ptr;

    switch (mp->b_datap->db_type) {

    default:
        /* don't understand it */
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
        *mp->b_wptr++ = EPROTO;
        qreply(q, mp);
        break;

    case M_FLUSH:
        /* standard flush handling goes here ... */
        break;

    case M_PROTO:
        /* Protocol message -> user request */
        proto = (union primitives *) mp->b_rptr;
        switch (proto->type) {
        default:
            mp->b_datap->db_type = M_ERROR;
            mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
            *mp->b_wptr++ = EPROTO;
            qreply(q, mp);
            return;

        case BIND_REQ:
            oldpri = LOCK(dgproto->lck, plstr);
            if (dgproto->state != IDLE) {
                err = EINVAL;
                goto error_ack;
            }
            if (mp->b_wptr - mp->b_rptr != sizeof(struct bind_req)) {
                err = EINVAL;
                goto error_ack;
            }
            if (err = chkaddr(proto->bind_req.BIND_addr))
                goto error_ack;

            dgproto->state = BOUND;
            dgproto->addr = proto->bind_req.BIND_addr;
            UNLOCK(dgproto->lck, oldpri);
            mp->b_datap->db_type = M_PCPROTO;
            proto->type = OK_ACK;
```

Screen 5-9. Write Procedure

```
            mp->b_wptr = mp->b_rptr + sizeof(struct ok_ack);
            qreply(q, mp);
            break;

        error_ack:
            UNLOCK(dgproto->lck, oldpri);
            mp->b_datap->db_type = M_PCPROTO;
            proto->type = ERROR_ACK;
            proto->error_ack.UNIX_error = err;
            mp->b_wptr = mp->b_rptr + sizeof(struct error_ack);
            qreply(q, mp);
            break;
        case UNITDATA_REQ:
            oldpri = LOCK(dgproto->lck, plstr);
            if (dgproto->state != BOUND)
                goto bad;
            if (mp->b_wptr - mp->b_rptr != sizeof(struct unitdata_req))
                goto bad;
            if (err = chkaddr(proto->unitdata_req.DEST_addr))
                goto bad;

                /* start device or mux output ... */

            UNLOCK(dgproto->lck, oldpri);
            putq(q, mp);
            break;

        bad:
            UNLOCK(dgproto->lck, oldpri);
            freemsg(mp);
            break;
    }
}
```

The write **`put`** procedure switches on the message type. The only types accepted are `M_FLUSH` and `M_PROTO`. For `M_FLUSH` messages, the driver performs the canonical flush handling (not shown). For `M_PROTO` messages, the driver assumes the message block contains a union primitive and switches on the `type` field. Two types are understood: `BIND_REQ` and `UNITDATA_REQ`.

For a `BIND_REQ`, the current state is checked; it must be `IDLE`. Next, the message size is checked. If it is the correct size, the passed-in address is verified for legality by calling **`chkaddr`**. If everything checks, the incoming message is converted into an `OK_ACK` and sent upstream. If there is any error, the incoming message is converted into an `ERROR_ACK` and sent upstream.

For `UNITDATA_REQ`, the state is also checked; it must be `BOUND`. As above, the message size and destination address are checked. If there is any error, the message is simply discarded. If all is well, the message is put on the queue, and the lower half of the driver is started.

If the write **`put`** procedure receives a message type that it does not understand, either a bad `b_datap->db_type` or bad `proto->type,` the message is converted into an `M_ERROR` message and sent upstream.

The generation of `UNITDATA_IND` messages (not shown in the example) normally occurs in the device interrupt if this is a hardware driver or in the lower read **`put`** procedure if this is a multiplexor. The algorithm is simple: The data part of the message is prepended by an `M_PROTO` message block that contains a `unitdata_ind` structure and sent upstream.

Message Allocation and Freeing

The **allocb** utility routine allocates a message and the space to hold the data for the message. **allocb** returns a pointer to a message block containing a data buffer of at least the size requested, providing there is enough memory available. It returns null on failure. Note that **allocb** always returns a message of type `M_DATA`. The type may then be changed if required. `b_rptr` and `b_wptr` are set to `db_base` (see `msgb` and `datab`), which is the start of the memory location for the data.

allocb may return a buffer larger than the size requested. If **allocb** indicates buffers are not available (**allocb** fails), the **put**/**service** procedure may not call **sleep** to wait for a buffer to become available. Instead, the **bufcall** utility can defer processing in the module or the driver until a buffer becomes available.

If message space allocation is done by the **put** procedure and **allocb** fails, the message is usually discarded. If the allocation fails in the **service** routine, the message is returned to the queue. **bufcall** is called to enable to the **service** routine when a message buffer becomes available, and the **service** routine returns.

The **freeb** utility routine releases (deallocates) the message block descriptor and the corresponding data block, if the reference count (see `datab` structure) is equal to 1. If the reference counter exceeds 1, the data block is not released.

The **freemsg** utility routine releases all message blocks in a message. It uses **freeb** to free all message blocks and corresponding data blocks.

In Screen 5-10, **allocb** is used by the **bappend** subroutine that appends a character to a message block:

```
/*
 * Append a character to a message block.
 * If (*bpp) is null, it will allocate a new block
 * Returns 0 when the message block is full, 1 otherwise
 */

#define MODBLKSZ128/* size of message blocks */

static bappend(mblk_t **bpp, int ch)
{
    mblk_t *bp;

    if ((bp = *bpp) != NULL) {
        if (bp->b_wptr >= bp->b_datap->db_lim)
            return 0;
    } else if ((*bpp = bp = allocb(MODBLKSZ, BPRI_MED)) == NULL)
        return 1;
    *bp->b_wptr++ = ch;
    return 1;
}
```

Screen 5-10. Appending a Character to a Message Block

bappend receives a pointer to a message block pointer and a character as arguments. If a message block is supplied `(*bpp != NULL)`, then **bappend** checks if there is room for more data in the block. If not, it fails. If there is no message block, a block of at least `MOD-BLKSZ` is allocated through **`allocb`**.

If the **`allocb`** fails, **bappend** returns success, silently discarding the character. This may or may not be acceptable. For TTY-type devices, it is generally accepted. If the original message block is not full or the **`allocb`** is successful, **bappend** stores the character in the block.

Screen 5-11 shows subroutine **`modwput`** which processes all the message blocks in any downstream data (type `M_DATA`) messages. **`freemsg`** deallocates messages.

```
/* Write side put procedure */
static modwput( queue_t *q, mblk_t *mp)
{
     switch (mp->b_datap->db_type) {
     default:
          putnext(q, mp);/* Don't do these, pass them along */
          break;

     case M_DATA: {
          register mblk_t *bp;
          struct mblk_t *nmp = NULL, *nbp = NULL;

          for (bp = mp; bp != NULL; bp = bp->b_cont) {
               while (bp->b_rptr < bp->b_wptr) {
                    if (*bp->b_rptr == '\n')
                         if (!bappend(&nbp, '\r'))
                              goto newblk;
                    if (!bappend(&nbp, *bp->b_rptr))
                         goto newblk;

                    bp->b_rptr++;
                    continue;

               newblk:
                    if (nmp == NULL)
                         nmp = nbp;
                    else linkb(nmp, nbp); /* link message block
                                   to tail of nmp */
                    nbp = NULL;
               }
          }

          if (nmp == NULL)
               nmp = nbp;
          else linkb(nmp, nbp);
          freemsg(mp); /* de-allocate message */
          if (nmp)
               putnext(q, nmp);
          break;
     }
     }
}
```

Screen 5-11. Processing Message Blocks

In Screen 5-11, data messages are scanned and filtered. **`modwput`** copies the original message into a new block(s), modifying as it copies; *nbp* points to the current new message block; and *nmp* points to the new message being formed as multiple `M_DATA` message blocks. The outer `for` loop goes through each message block of the original message, while the inner `while` loop goes through each byte. **bappend** is used to add characters to the current or new block; if it fails, the current new block is full. If *nmp* is `NULL`, *nmp* is pointed at the new block. If *nmp* is not `NULL`, the new block is linked to the end of *nmp* with the **`linkb`** utility.

At the end of the loops, the final new block is linked to *nmp*. The original message (all message blocks) is returned to the pool by **`freemsg`**. If a new message exists, it is sent downstream.

Recovering from No Buffers

The **`bufcall`** utility can recover from an **`allocb`** failure. The call syntax is as follows:

`bufcall``(int` *size*`, int` *pri*`, int (*`*func*`)(), long` *arg*`)`

`bufcall` calls (**func*)(*arg*) when a buffer of *size* bytes is available. When *func* is called, it has no user context and must return without sleeping. Also, because of interrupt processing, and multiprocessor contention for resources, there is no guarantee that when *func* is called, a buffer will actually be available (someone else may steal it).

On success, **`bufcall`** returns a nonzero identifier that can be used as a parameter to **`unbufcall`** to cancel the request later. On failure, 0 is returned and the requested function will never be called.

NOTE

Make sure you avoid deadlock when holding resources while waiting for **`bufcall`** to call (**func*)(*arg*). Use **`bufcall`** sparingly.

Two examples, Screen 5-12 and Screen 5-13, are provided. Screen 5-12 is a device receive interrupt handler:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>

dev_rintr(int dev)
{
    /* process incoming message ... */

    /* allocate new buffer for device */
    dev_re_load(dev);
}
/*
 * Reload device with a new receive buffer
 */
dev_re_load(int dev)
{
    mblk_t *bp;

    if ((bp = allocb(DEVBLKSZ, BPRI_MED)) == NULL) {
        cmn_err(CE_WARN, “dev: allocb failure (size %d)\n”, DEVBLKSZ);
        /*
         * Allocation failed.  Use bufcall to
         * schedule a call to ourselves.
         */
        (void) bufcall(DEVBLKSZ, BPRI_MED, dev_re_load, dev);
        return;
    }

    /* pass buffer to device ... */
}
```

Screen 5-12. Device Receive Interrupt Handler

`dev_rintr` is called when the device has posted a receive interrupt. The code retrieves the data from the device (not shown). **`dev_rintr`** must then give the device another buffer to fill by a call to **`dev_re_load`**, which calls **`allocb`**. If **`allocb`** fails, **`dev_re_load`** uses **`bufcall`** to call itself when STREAMS determines a buffer is available.

NOTE

Because **`bufcall`** may fail, there is still a chance that the device may hang. A better strategy, if **`bufcall`** fails, is to discard the current input message and resubmit that buffer to the device. Losing input data is generally better than hanging.

Screen 5-13 is a write **`service`** procedure, **`mod_wsrv`**, which needs to prepend each output message with a header. **`mod_wsrv`** illustrates a case for potential deadlock:

```
static int mod_wsrv(queue_t  *q)
{
    int qenable;
    mblk_t *mp, *bp;

    while (mp = getq(q)) {

        /* check for priority messages and canput ... */

        /* Allocate a header to prepend to the message.  If
         * the allocb fails, use bufcall to reschedule.
         */
        if ((bp = allocb(HDRSZ, BPRI_MED)) == NULL) {
            if (!bufcall(HDRSZ, BPRI_MED, qenable, q)) {
               itimeout(qenable, q, HZ*2, plstr);
            }
            /* Put the message back and exit, we will be re-enabled later */
            putbq(q, mp);
            return;
        }
        /* process message .... */
    }
}
```

Screen 5-13. Write Service Procedure

However, if **allocb** fails, **mod_wsrv** wants to recover without loss of data and calls **bufcall**. In this example, the routine passed to **bufcall** is **qenable**. When a buffer is available, the **service** procedure is automatically re-enabled. Before exiting, the current message is put back on the queue. This example deals with **bufcall** failure by resorting to the **itimeout** operating system utility routine. This routine schedules the given function to be run with the given argument in the given number of clock ticks (there are HZ clock ticks per second). In this example, if **bufcall** fails, the system runs **qenable** after two seconds have passed.

Extended STREAMS Buffers

Some hardware using the STREAMS mechanism supports memory-mapped I/O that allows the sharing of buffers between users, kernel, and the I/O card.

If the hardware supports memory-mapped I/O, data received from the network are placed in the DARAM (dual access RAM) section of the I/O card. Because DARAM is a shared memory between the kernel and the I/O card, data transfer between the kernel and the I/O card is eliminated. Once in kernel space, you can manipulate the data buffer as if it were a kernel resident buffer. Similarly, data being sent downstream is placed in DARAM and then forwarded to the network.

In a typical network arrangement, data is received from the network by the I/O card. The block of data is read into the card's internal buffer. It interrupts the host computer to denote that data have arrived. The STREAMS driver gives the controller the kernel address where the data block is to go and the number of bytes to transfer. After the controller reads the data into its buffer and verifies the checksum, it copies the data into main memory to the address specified by the direct memory access (DMA) memory address. Once in the kernel space, the data is packaged into message blocks and processed in the usual way.

When data is transmitted from user process to the network, the data is copied from the user space to the kernel space, and packaged as a message block and sent to the downstream driver. The driver interrupts the I/O card signaling that data is ready to be transmitted to the network. The controller copies the data from the kernel space to the internal buffer on the I/O card, and from there data is placed on the network.

The STREAMS buffer allocation mechanism enables the allocation of message and data blocks to point directly to a client-supplied (non-STREAMS) buffer. Message and data blocks allocated this way are indistinguishable (for the most part) from the normal data blocks. The client-supplied buffers are processed as if they were normal STREAMS data buffers.

Drivers may not only attach non-STREAMS data buffers but also free them. This is done as follows:

- *Allocation* - if the drivers are to use DARAM without wasting STREAMS resources and without being dependent on upstream modules, a data and message block can be allocated without an attached data buffer. The routine to use is called **`esballoc`**. This returns a message block and data block without an associated STREAMS buffer. The buffer used is the one supplied by the caller.
- *Freeing* - each driver using non-STREAMS resources in a STREAMS environment must fully manage those resources, including freeing them. However, to make this as transparent as possible, a driver-dependent routine is executed if **`freeb`** is called to free a message and data block with an attached non-STREAMS buffer.

  `freeb` detects if a buffer is a client supplied, non-STREAMS buffer. If it is, **`freeb`** finds the **`free_rtn`** structure associated with that buffer. After calling the driver-dependent routine (defined in **`free_rtn`**) to free the buffer, the **`freeb`** routine frees the message and data block.

NOTE

The free routine must not reference any dynamically allocated data structures that become freed when the driver is closed, because messages can exist in a Stream after the driver is closed. This can occur, for example, when a Stream is closed down. The driver close routine is called and the driver's private data structure may be deallocated. If the driver sends a message created by **`esballoc`** upstream, that message may still be on the Stream head read queue. The Stream head read queue is then flushed, freeing the message and calling the driver's free routine after the driver has been closed.

The format of the **`free_rtn`** structure is as follows:

```
struct free_rtn {
    void (*free_func) (); /* driver dependent free routine */
    char *free_arg;       /* argument for free_rtn */
};
typedef struct free_rtn frtn_t;
```

The structure has two fields: a pointer to a function and a location for any argument passed to the function. Instead of defining a specific number of arguments, **`free_arg`** is defined as a `char *`. Drivers can then pass pointers to structures if more than one argument is needed.

The STREAMS utility routine, **`esballoc`**, provides a common interface for allocating and initializing data blocks. It makes the allocation as transparent to the driver as possible and provides a way to change the fields of the data block, since modification should only be performed by STREAMS. The driver calls this routine when it wants to attach its own data buffer to a newly allocated message and data block. If the routine successfully completes the allocation and assigns the buffer, it returns a pointer to the message block. The driver is responsible for supplying the arguments to **`esballoc`**, namely, a pointer to its data buffer, the size of the buffer, the priority of the data block, and a pointer to the **`free_rtn`** structure. All arguments should be non-`NULL`. See the *Device Driver Reference* for a detailed description of **`esballoc`**.

Message Types

All the STREAMS messages are defined in **`sys/stream.h`**. The messages differ in their intended purpose and their queueing priority. The contents of certain message types can be transferred between a process and a Stream by system calls.

Below, the message types are briefly described and classified according to their queueing priority.

Ordinary Messages (also called “normal” messages):

`M_BREAK`	Request to a Stream driver to send a “break”
`M_CTL`	Control/status request used for intermodule communication
`M_DATA`	User data message for I/O system calls
`M_DELAY`	Request a real-time delay on output
`M_IOCTL`	Control/status request generated by a Stream head
`M_PASSFP`	File pointer passing message
`M_PROTO`	Protocol control information
`M_RSE`	Reserved for internal use
`M_SETOPTS`	Set options at the Stream head, sent upstream

`M_SIG`	Signal sent from a module/driver to a user

High Priority Messages:

`M_COPYIN`	Copy in data for transparent **`ioctl`**s, sent upstream
`M_COPYOUT`	Copy out data for transparent **`ioctl`**s, sent upstream
`M_ERROR`	Report downstream error condition, sent upstream
`M_FLUSH`	Flush module queue
`M_HANGUP`	Set a Stream head hangup condition, sent upstream
`M_IOCACK`	Positive **`ioctl(2)`** acknowledgment
`M_IOCDATA`	Data for transparent **`ioctl`**s, sent downstream
`M_IOCNAK`	Negative **`ioctl(2)`** acknowledgment
`M_PCPROTO`	Protocol control information
`M_PCRSE`	Reserved for internal use
`M_PCSIG`	Signal sent from a module/driver to a user
`M_READ`	Read notification, sent downstream
`M_START`	Restart stopped device output
`M_STARTI`	Restart stopped device input
`M_STOP`	Suspend output
`M_STOPI`	Suspend input

NOTE

Transparent **`ioctl`**s support applications developed before the introduction of STREAMS.

Defined STREAMS message types differ in their intended purposes, their treatment at the Stream head, and in their message queueing priority.

STREAMS does not prevent a module or driver from generating any message type and sending it in any direction on the Stream. However, established processing and direction rules should be observed. Stream head processing according to message type is fixed, although certain parameters can be altered.

Detailed Description of Message Types

The message types are classified according to their message queueing priority. Ordinary messages are described first, with high priority messages following. In certain cases, two message types may perform similar functions, differing only in priority. The use of the word “module” generally implies “module or driver.”

Ordinary messages are also called normal or non-priority messages. Ordinary messages are subject to flow control whereas high priority messages are not.

Ordinary Messages

M_BREAK

Sent to a driver to request that BREAK be transmitted on whatever media the driver is controlling.

The message format is not defined by STREAMS and its use is developer dependent. This message may be considered a special case of an `M_CTL` message. An `M_BREAK` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_CTL

Generated by modules that want to send information to a particular module or type of module. `M_CTL` messages are typically used for inter-module communication, as when adjacent STREAMS protocol modules negotiate the terms of their interface. An `M_CTL` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_DATA

Intended to contain ordinary data. Messages allocated by the **`allocb`** routine are type `M_DATA` by default. `M_DATA` messages are generally sent bidirectionally on a Stream and their contents can be passed between a process and the Stream head. In the **`getmsg(2)`** and **`putmsg(2)`** system calls, the contents of `M_DATA` message blocks are referred to as the data part. Messages composed of multiple message blocks will typically have `M_DATA` as the message type for all message blocks following the first.

M_DELAY

Sent to a media driver to request a real-time delay on output. The data buffer associated with this message is expected to contain an integer to show the number of machine ticks of delay desired. `M_DELAY` messages are typically used to prevent transmitted data from exceeding the buffering capacity of slower terminals.

The message format is not defined by STREAMS and its use is developer dependent. Not all media drivers may understand this message. This message may be considered a special case of an `M_CTL` message. An `M_DELAY` message cannot be generated by a user-level process and is always discarded if passed to the Stream head.

M_IOCTL

Generated by the Stream head in response to **`I_STR`**, **`I_LINK`**, **`I_UNLINK`**, **`I_PLINK`**, and **`I_PUNLINK`** (**`ioctl(2)`** STREAMS system calls, see **`streamio(7)`**), and in response to **`ioctl`** calls that contain a command argument value not defined in **`streamio(7)`**. When one of these **`ioctl`**s is received from a user process, the Stream head uses values supplied in the call and values from the process to create an `M_IOCTL` message containing them, and sends the message downstream. `M_IOCTL` messages are intended to perform the general **`ioctl`** functions of character device drivers.

For an **`I_STR`** **`ioctl`**, the user values are supplied in a structure of the following form, provided as an argument to the **`ioctl`** call (see **`I_STR`** in **`streamio(7)`**):

```
struct strioctl
{
    int   ic_cmd;    /* downstream request */
    int   ic_timout; /* ACK/NAK timeout */
    int   ic_len;    /* length of data arg */
    char *ic_dp;     /* ptr to data arg */
};
```

where `ic_cmd` is the request (or command) defined by a downstream module or driver, `ic_timout` is the time the Stream head will wait for acknowledgment to the `M_IOCTL` message before timing out, and `ic_dp` is a pointer to an optional data buffer. On input, `ic_len` contains the length of the data in the buffer passed in and, on return from the call, it contains the length of the data, if any, being returned to the user in the same buffer.

The `M_IOCTL` message format is one `M_IOCTL` message block followed by zero or more `M_DATA` message blocks. STREAMS constructs an `M_IOCTL` message block by placing an `iocblk` structure, defined in **`sys/stream.h`**, in its data buffer:

```
struct iocblk
{
    int      ioc_cmd;         /* ioctl command type */
    cred_t  *ioc_cr;          /* full credentials */
    uint     ioc_id;          /* ioctl identifier */
    uint     ioc_count;       /* byte count for ioctl data */
    int      ioc_error;       /* error code for M_IOCACK or M_IOCNAK */
    int      ioc_rval;        /* return value for M_IOCACK */
    long     ioc_filler[4];   /* reserved for future use */
};
```

For an **`I_STR`** **`ioctl`**, `ioc_cmd` corresponds to `ic_cmd` of the `strioctl` structure. `ioc_cr` points to a credentials structure defining the user process's permissions (see **`cred.h`**). Its contents can be tested to determine if the user issuing the **`ioctl`** call is authorized to do so. For an **`I_STR`** **`ioctl`**, `ioc_count` is the number of data bytes, if any, contained in the message and corresponds to `ic_len`.

`ioc_id` is an identifier generated internally, and is used by the Stream head to match each `M_IOCTL` message sent downstream with response messages sent upstream to the Stream head. The response message which completes the Stream head processing for the **`ioctl`** is an `M_IOCACK` (positive acknowledgment) or an `M_IOCNAK` (negative acknowledgment) message.

For an **`I_STR`** **`ioctl`**, if a user supplies data to be sent downstream, the Stream head copies the data, pointed to by `ic_dp` in the `strioctl` structure, into `M_DATA` message blocks and links the blocks to the initial `M_IOCTL` message block. `ioc_count` is copied from `ic_len`. If there is no data, `ioc_count` is zero.

If the Stream head does not recognize the command argument of an `ioctl`, it creates a transparent `M_IOCTL` message. The format of a transparent `M_IOCTL` message is one `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure is the same as above. However, `ioc_cmd` is set to the value of the command argument in the **`ioctl`** system call and `ioc_count` is set to `TRANSPARENT`, defined in **`sys/stream.h`**. `TRANSPARENT` distinguishes the case where an **`I_STR`** **`ioctl`** may specify a value of `ioc_cmd` equivalent to the command argument of a transparent **`ioctl`**. The `M_DATA` block of the message contains the value of the *arg* parameter in the **`ioctl`** call.

The first module or driver that understands the `ioc_cmd` request contained in the `M_IOCTL` acts on it. For an **`I_STR`** **`ioctl`**, this action generally includes an immediate upstream transmission of an `M_IOCACK` message. For transparent `M_IOCTL`s, this action generally includes the upstream transmission of an `M_COPYIN` or `M_COPYOUT` message.

Intermediate modules that do not recognize a particular request must pass the message on. If a driver does not recognize the request, or the receiving module can not acknowledge it, an `M_IOCNAK` message must be returned.

`M_IOCACK` and `M_IOCNAK` message types have the same format as an `M_IOCTL` message and contain an `iocblk` structure in the first block. An `M_IOCACK` block may be linked to following `M_DATA` blocks. If one of these messages reaches the Stream head with an identifier that does not match that of the currently-outstanding `M_IOCTL` message, the response message is discarded. A common means of assuring that the correct identifier is returned is for the replying module to convert the `M_IOCTL` message into the appropriate response type and set ioc_count to 0, if no data is returned. Then, the **qreply** utility is used to send the response to the Stream head.

In an M_IOCACK or M_IOCNAK message, **ioc_error** holds any return error condition set by a downstream module. If this value is non-zero, it is returned to the user in errno. Note that both an M_IOCNAK and an M_IOCACK may return an error. However, only an M_IOCACK can have a return value. For an M_IOCACK, ioc_rval holds any return value set by a responding module. For an M_IOCNAK, ioc_rval is ignored by the Stream head.

If a module processing an **I_STR ioctl** wants to send data to a user process, it must use the M_IOCACK message that it constructs such that the M_IOCACK block is linked to one or more following M_DATA blocks containing the user data. The module must set ioc_count to the number of data bytes sent. The Stream head places the data in the address pointed to by ic_dp in the user **I_STR** strioctl structure.

If a module processing a transparent **ioctl** (that is, it received a transparent M_IOCTL) wants to send data to a user process, it can use only an M_COPYOUT message. For a transparent **ioctl**, no data can be sent to the user process in an M_IOCACK message. All data must have been sent in a preceding M_COPYOUT message. The Stream head will ignore any data contained in an M_IOCACK message (in M_DATA blocks) and will free the blocks.

No data can be sent with an M_IOCNAK message for any type of M_IOCTL. The Stream head will ignore and will free any M_DATA blocks.

The Stream head blocks the user process until an M_IOCACK or M_IOCNAK response to the M_IOCTL (same ioc_id) is received. For an M_IOCTL generated from an **I_STR ioctl**, the Stream head will time out if no response is received in ic_timout interval (the user may specify an explicit interval or specify use of the default interval). For M_IOCTL messages generated from all other **ioctl**s, the default (infinite) is used.

M_PASSFP

Used by STREAMS to pass a file pointer from the Stream head at one end of a Stream pipe to the Stream head at the other end of the same Stream pipe.

The message is generated as a result of an **I_SENDFD ioctl** (see **streamio(7)**) issued by a process to the sending Stream head. STREAMS places the M_PASSFP message directly on the destination Stream head's read queue to be retrieved by an **I_RECVFD ioctl** (see **streamio(7)**). The message is placed without passing it through the Stream (that is, it is not seen by any modules or drivers in the Stream). This message should never be present on any queue except the read queue of a Stream head. Consequently, modules and drivers do not need to recognize this message, and it can be ignored by module and driver developers.

M_PROTO

Intended to contain control information and associated data. The message format is one or more (see note) M_PROTO message blocks followed by zero or more M_DATA message blocks as shown in Figure 5-7. The semantics of the `M_DATA` and `M_PROTO` message block are determined by the STREAMS module that receives the message.

The `M_PROTO` message block will typically contain implementation dependent control information. `M_PROTO` messages are generally sent bidirectionally on a Stream, and their contents can be passed between a process and the Stream head. The contents of the first message block of an `M_PROTO` message is generally referred to as the control part, and the contents of any following `M_DATA` message blocks are referred to as the data part. In the **`getmsg(2)`** and **`putmsg(2)`** system calls, the control and data parts are passed separately.

NOTE

On the write-side, the user can only generate `M_PROTO` messages containing one `M_PROTO` message block.

Although its use is not recommended, the format of `M_PROTO` and `M_PCPROTO` (generically `PROTO`) messages sent upstream to the Stream head allows multiple `PROTO` blocks at the beginning of the message. **`getmsg(2)`** will compact the blocks into a single control part when passing them to the user process.

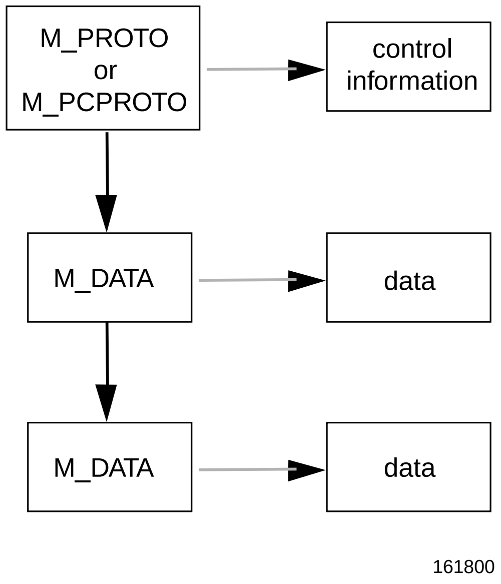


Figure 5-7. M_PROTO and M_PCPROTO Message Structure

M_RSE

Reserved for internal use. Modules that do not recognize this message must pass it on. Drivers that do not recognize it must free it.

M_SETOPTS

Used to alter some characteristics of the Stream head. It is generated by any downstream module, and is interpreted by the Stream head. The data buffer of the message has the following structure:

```
struct stroptions
{
    ulong  so_flags;          /* options to set */
    short  so_readopt;        /* read option */
    ushort so_wroff;          /* write offset */
    long   so_minpsz;         /* minimum read packet size */
    long   so_maxpsz;         /* maximum read packet size */
    ulong  so_hiwat;          /* read queue high-water mark */
    ulong  so_lowat;          /* read queue low-water mark */
    unsigned char so_band;    /* update water marks for this band */
};
```

where `so_flags` specifies which options are to be altered, and can be any combination of the following:

`SO_ALL`:	Update all options according to the values specified in the remaining fields of the `stroptions` structure.
`SO_READOPT`:	Set the read mode (see **`read(2)`**) as specified by the value of `so_readopt`:

RNORM	Byte stream
RMSGD	Message discard
RMSGN	Message non-discard
RPROTNORM	Normal protocol
RPROTDAT	Turn `M_PROTO` and `M_PCPROTO` messages into `M_DATA` messages.
RPROTDIS	Discard `M_PROTO` and `M_PCPROTO` blocks in a message and retain any linked `M_DATA` blocks.

`SO_WROFF`:	Direct the Stream head to insert an offset specified by `so_wroff` into the first message block of all `M_DATA` messages created as a result of a **`write(2)`** system call. The same offset is inserted into the first `M_DATA` message block, if any, of all messages created by a **`putmsg`** system call. The default offset is zero.

The offset must be less than the maximum message buffer size (system dependent). Under certain circumstances, a write offset may not be inserted. A module or driver must test that `b_rptr` in the `msgb` structure is greater than `db_base` in the `datab` structure to determine that an offset has been inserted in the first message block.

SO_MINPSZ:	Change the minimum packet size value associated with the Stream head read queue to so_minpsz. This value is advisory for the module immediately below the Stream head. It is intended to limit the size of M_DATA messages that the module should put to the Stream head. There is no intended minimum size for other message types. The default value in the Stream head is zero.
SO_MAXPSZ:	Change the maximum packet size value associated with the Stream head read queue to so_maxpsz This value is advisory for the module immediately below the Stream head. It is intended to limit the size of M_DATA messages that the module should put to the Stream head. There is no intended maximum size for other message types. The default value in the Stream head is INFPSZ, the maximum STREAMS allows.
SO_HIWAT:	Change the flow control high water mark (q_hiwat in the queue structure, qb_hiwat in the qband structure) on the Stream head read queue to the value specified in so_hiwat.
SO_LOWAT:	Change the flow control low water mark (q_lowat in the queue structure, qb_lowat in the qband structure) on the Stream head read queue to the value specified in so_lowat.
SO_MREADON:	Enable the Stream head to generate M_READ messages when processing a **read(2)** system call. If both SO_MREADON and SO_MREADOFF are set in so_flags, SO_MREADOFF will have precedence.
SO_MREADOFF:	Disable the Stream head generation of M_READ messages when processing a **read(2)** system call. This is the default. If both SO_MREADON and SO_MREADOFF are set in so_flags, SO_MREADOFF will have precedence.
SO_NDELON:	Set non-STREAMS tty semantics for O_NDELAY (or O_NONBLOCK) processing on **read(2)** and **write(2)** system calls. If O_NDELAY (or O_NONBLOCK) is set, a **read(2)** will return 0 if no data is waiting to be read at the Stream head. If O_NDELAY (or O_NONBLOCK) is clear, a **read(2)** will block until data becomes available at the Stream head.

Regardless of the state of O_NDELAY (or O_NONBLOCK), a **write(2)** will block on flow control and will block if buffers are not available.

If both SO_NDELON and SO_NDELOFF are set in so_flags, SO_NDELOFF will have precedence.

NOTE

For conformance with the POSIX standard, it is recommended that new applications use the O_NONBLOCK flag whose behavior is the same as that of O_NDELAY unless otherwise noted.

SO_NDELOFF: Set STREAMS semantics for O_NDELAY (or O_NONBLOCK) processing on **read(2)** and **write(2)** system calls. If O_NDELAY (or O_NONBLOCK) is set, a **read(2)** will return -1 and set EAGAIN if no data is waiting to be read at the Stream head. If O_NDELAY (or O_NONBLOCK) is clear, a **read(2)** will block until data becomes available at the Stream head. (See the note above.)

If O_NDELAY (or O_NONBLOCK) is set, a **write(2)** will return -1 and set EAGAIN if flow control is in effect when the call is received. It will block if buffers are not available. If O_NDELAY (or O_NONBLOCK) is set and part of the buffer has been written and a flow control or buffers not available condition is encountered, **write(2)** will terminate and return the number of bytes written.

If O_NDELAY (or O_NONBLOCK) is clear, a **write(2)** will block on flow control and will block if buffers are not available.

This is the default. If both SO_NDELON and SO_NDELOFF are set in so_flags, SO_NDELOFF will have precedence.

In the STREAMS-based pipe mechanism, the behavior of **read(2)** and **write(2)** is different for the O_NDELAY and O_NONBLOCK flags. See **read(2)** and **write(2)** for details.

SO_BAND: Set water marks in a band. If the SO_BAND flag is set with the SO_HIWAT or SO_LOWAT flag, the so_band field contains the priority band number the so_hiwat and so_lowat fields pertain to.

If the SO_BAND flag is not set and the SO_HIWAT and SO_LOWAT flags are on, the normal high and low water marks are affected. The SO_BAND flag has no effect if SO_HIWAT and SO_LOWAT flags are off.

Only one band's water marks can be updated with a single M_SETOPTS message.

SO_ISTTY: Inform the Stream head that the Stream is acting like a controlling terminal.

SO_ISNTTY: Inform the Stream head that the Stream is no longer acting like a controlling terminal.

For SO_ISTTY, the Stream may or may not be allocated as a controlling terminal via an M_SETOPTS message arriving upstream during open processing. If the Stream head is opened before receiving this message, the Stream will not be allocated as a controlling terminal until it is queued again by a session leader.

SO_TOSTOP: Stop on background writes to the Stream.

SO_TONSTOP: Do not stop on background writes to the Stream.

SO_TOSTOP and SO_TONSTOP are used with job control.

M_SIG

Sent upstream by modules or drivers to post a signal to a process. When the message reaches the front of the Stream head read queue, it evaluates the first data byte of the message as a signal number, defined in **sys/signal.h**. (Note that the signal is not generated until it reaches the front of the Stream head read queue.) The associated signal will be sent to process(es) under the following conditions:

If the signal is SIGPOLL, it will be sent only to those processes that have explicitly registered to receive the signal (see **I_SETSIG** in **streamio(7)**).

If the signal is not SIGPOLL and the Stream containing the sending module or driver is a controlling tty, the signal is sent to the associated process group. A Stream becomes the controlling tty for its process group if, on **open(2)**, a module or driver sends an M_SETOPTS message to the Stream head with the SO_ISTTY flag set.

If the signal is not SIGPOLL and the Stream is not a controlling tty, no signal is sent, except in case of SIOCSPGRP and TIOCSPGRP. These two ioctls set the process group field in the Stream head so the Stream can generate signals even if it is not a controlling tty.

High Priority Messages

M_COPYIN

Generated by a module or driver and sent upstream to request that the Stream head perform a **copyin** for the module or driver. It is valid only after receiving an M_IOCTL message and before an M_IOCACK or M_IOCNAK.

The message format is one M_COPYIN message block containing a copyreq structure, defined in **sys/stream.h**:

```
struct copyreq {
    int      cq_cmd;      /* ioctl command (from ioc_cmd) */
    cred_t  *cq_cr;       /* full credentials */
    uint     cq_id;       /* ioctl id (from ioc_id) */
    caddr_t  cq_addr;     /* address to copy data to/from */
    uint     cq_size;     /* number of bytes to copy */
    int      cq_flag;     /* reserved */
    mblk_t  *cq_private;  /* private state information */
    long     cp_filler[4];/* reserved for future use */
};
```

The first four members of the structure correspond to those of the iocblk structure in the M_IOCTL message which allows the same message block to be reused for both structures. The Stream head will guarantee that the message block allocated for the M_IOCTL message is large enough to contain a copyreq structure. The cq_addr field contains the user space address from which the data is to be copied. The cq_size field is the number of bytes to copy from user space. The `cq_flag` field is reserved for future use and should be set to zero.

The `cq_private` field can be used by a module to point to a message block containing the module's state information relating to this **`ioctl`**. The Stream head will copy (without processing) the contents of this field to the `M_IOCDATA` response message so that the module can resume the associated state. If an `M_COPYIN` or `M_COPYOUT` message is freed, STREAMS will not free any message block pointed to by `cq_private`. This is the module's responsibility.

This message should not be queued by a module or driver unless it intends to process the data for the **`ioctl`**.

M_COPYOUT

Generated by a module or driver and sent upstream to request that the Stream head perform a **`copyout`** for the module or driver. It is valid only after receiving an `M_IOCTL` message and before an `M_IOCACK` or `M_IOCNAK`.

The message format is one `M_COPYOUT` message block followed by one or more `M_DATA` blocks. The `M_COPYOUT` message block contains a `copyreq` structure as described in the `M_COPYIN` message with the following differences: The `cq_addr` field contains the user space address to which the data is to be copied. The `cq_size` field is the number of bytes to copy to user space.

Data to be copied to user space is contained in the linked `M_DATA` blocks.

This message should not be queued by a module or driver unless it intends to process the data for the **`ioctl`** in some way.

M_ERROR

Sent upstream by modules or drivers to report some downstream error condition. When the message reaches the Stream head, the Stream is marked so that all subsequent system calls issued to the Stream, excluding **`close(2)`** and **`poll(2)`**, will fail with `errno` set to the first data byte of the message. `POLLERR` is set if the Stream is being `poll`ed (see **`poll(2)`**). All processes sleeping on a system call to the Stream are awakened. An `M_FLUSH` message with `FLUSHRW` is sent downstream.

The Stream head maintains two error fields, one for the read-side and one for the write-side. The one-byte format `M_ERROR` message sets both of these fields to the error specified by the first byte in the message.

The second style of the `M_ERROR` message is two bytes long. The first byte is the read error and the second byte is the write error. This allows modules to set a different error on the read-side and write-side. If one of the bytes is set to `NOERROR`, then the field for the corresponding side of the Stream is unchanged. This allows a module to just an error on one side of the Stream. For example, if the Stream head was not in an error state and a module sent an `M_ERROR` message upstream with the first byte set to `EPROTO` and the second byte set to `NOERROR`, all subsequent read-like system calls (for example, **`read`**,

getmsg) will fail with EPROTO, but all write-like system calls (for example, **write**, **putmsg**) will still succeed. If a byte is set to 0, the error state is cleared for the corresponding side of the Stream. The values NOERROR and 0 are not valid for the one-byte form of the M_ERROR message.

M_FLUSH

Requests all modules and drivers that receive it to flush their message queues (discard all messages in those queues) as indicated in the message. All modules that enqueue messages must identify and process this message type.

An M_FLUSH can originate at the Stream head, or in any module or driver. The first byte of the message contains flags that specify one of the following actions:

- FLUSHR: Flush the read queue of the module.
- FLUSHW: Flush the write queue of the module.
- FLUSHRW: Flush both the read queue and the write queue of the module.
- FLUSHBAND: Flush the message according to the priority associated with the band.

Each module passes this message to its neighbor after flushing its appropriate queue(s), until the message reaches one end of the Stream.

Drivers are expected to include the following processing for M_FLUSH messages. When an M_FLUSH message is sent downstream through the write queues in a Stream, the driver at the Stream end should flush its queues according to the flag settings as follows:

- If only FLUSHW is set, the write queue is flushed and the message is discarded.
- If the message indicates that the read queues are to be flushed, the driver should flush its read queue, shut off the FLUSHW flag, and send the message up the Stream's read queues.

When a flush message is sent up a Stream's read-side, the Stream head checks whether the write-side of the Stream is to be flushed:

- If only FLUSHR is set, the Stream head discards the message.
- If FLUSHW is set, the Stream head turns off the FLUSHR flag and sends the message down the Stream's write side.

The lower side of a multiplexing driver should process M_FLUSH messages the same as the Stream head.

If FLUSHBAND is set, the second byte of the message contains the value of the priority band to flush.

M_HANGUP

Sent upstream by a driver to report that it can no longer send data upstream. As example, this might be because of an error, or to a remote line connection being dropped. When the message reaches the Stream head, the Stream is marked so that all subsequent **write(2)** and **putmsg(2)** system calls issued to the Stream will fail and return an EIO error. Those **ioctl**s that cause messages to be sent downstream are also failed. POLLHUP is set if the Stream is being polled (see **poll(2)**).

However, subsequent **read(2)** or **getmsg(2)** calls to the Stream will not generate an error. These calls will return any messages (according to their function) that were on, or in transit to, the Stream head read queue before the M_HANGUP message was received. When all such messages have been read, **read(2)** will return 0 and **getmsg(2)** will set each of its two length fields to 0.

This message also causes a SIGHUP signal to be sent to the controlling process instead of the foreground process group, since the allocation and deallocation of controlling terminals to a session is the responsibility of the controlling process.

M_IOCACK

Signals the positive acknowledgment of a previous M_IOCTL message. The message format is one M_IOCACK block (containing an iocblk structure, see M_IOCTL) followed by zero or more M_DATA blocks. The iocblk data structure may contain a value in ioc_rval to be returned to the user process. It may also contain a value in ioc_error to be returned to the user process in errno.

If this message is responding to an **I_STR ioctl** (see **streamio(7)**), it may contain data from the receiving module or driver to be sent to the user process. In this example, message format is one M_IOCACK block followed by one or more M_DATA blocks containing the user data. The Stream head returns the data to the user if there is a corresponding outstanding M_IOCTL request. Otherwise, the M_IOCACK message is ignored and all blocks in the message are freed.

Data can not be returned in an M_IOCACK message responding to a transparent M_IOCTL. The data must have been sent with preceding M_COPYOUT message(s). If any M_DATA blocks follow the M_IOCACK block, the Stream head will ignore and free them.

The format and use of this message type is described further under M_IOCTL.

M_IOCDATA

Generated by the Stream head and sent downstream as a response to an M_COPYIN or M_COPYOUT message. The message format is one M_IOCDATA message block followed by zero or more M_DATA blocks. The M_IOCDATA message block contains a copyresp structure, defined in **sys/stream.h**.

```
struct copyresp {
   int     cp_cmd;       /* ioctl command (from ioc_cmd) */
   cred_t  *cp_cr;       /* full credentials */
   uint    cp_id;        /* ioctl id (from ioc_id) */
   caddr_t cp_rval;      /* status of request: 0 -> success
                            non_zero -> failure */
   uint    cp_pad1;      /* reserved */
   int     cp_pad2;      /* reserved */
   mblk_t  *cp_private;  /* private state info from cq_private */
   long    cp_filler[4]; /* reserved for future use */
};
```

The first three members of the structure correspond to those of the iocblk structure in the M_IOCTL message which allows the same message blocks to be reused for all of the related transparent messages (M_COPYIN, M_COPYOUT, M_IOCACK, M_IOCNAK). The cp_rval field contains the result of the request at the Stream head. Zero indicates success and non-zero indicates failure. If failure is indicated, the module should not generate an M_IOCNAK message. It must abort all **ioctl** processing, clean up its data structures, and return. The cp_private field is copied from the cp_private field in the associated M_COPYIN or M_COPYOUT message. It is included in the M_IOCDATA message so the message can be self-describing. This is intended to simplify **ioctl** processing by modules and drivers.

If the message is in response to an M_COPYIN message and success is indicated, the M_IOCDATA block will be followed by M_DATA blocks containing the data copied in.

If an M_IOCDATA block is reused, any unused fields defined for the resultant message block should be cleared (particularly in an M_IOCACK or M_IOCNAK).

This message should not be queued by a module or driver unless it intends to process the data for the **ioctl** in some way.

M_IOCNAK

Signals the negative acknowledgment (failure) of a previous M_IOCTL message. Its form is one M_IOCNAK block containing an iocblk data structure (see M_IOCTL). The iocblk structure may contain a value in ioc_error to be returned to the user process in errno. Unlike the M_IOCACK, no user data or return value can be sent with this message. If any M_DATA blocks follow the M_IOCNAK block, the Stream head will ignore and free them. When the Stream head receives an M_IOCNAK, the outstanding **ioctl** request, if any, will fail. The format and usage of this message type is described further under M_IOCTL.

M_PCPROTO

Similar to the M_PROTO message type, except for the priority and the following additional attributes.

When an M_PCPROTO message is placed on a queue, its **service** procedure is always enabled. The Stream head will allow only one M_PCPROTO message to be placed in its read queue at a time. If an `M_PCPROTO` message is already in the queue when another arrives, the second message is silently discarded and its message blocks freed.

This message is intended to allow data and control information to be sent outside the normal flow control constraints.

The **`getmsg(2)`** and **`putmsg(2)`** system calls refer to `M_PCPROTO` messages as high priority messages.

M_PCRSE

Reserved for internal use. Modules that do not recognize this message must pass it on. Drivers that do not recognize it must free it.

M_PCSIG

Similar to the `M_SIG` message, except for the priority.

`M_PCSIG` is often preferable to the `M_SIG` message especially in tty applications, because `M_SIG` may be queued while `M_PCSIG` is more guaranteed to get through quickly. For example, if one generates an `M_SIG` message when the DEL (delete) key is pressed on the terminal and one has already typed ahead, the `M_SIG` message becomes queued and the user doesn't get the call until it's too late; it becomes impossible to kill or interrupt a process by pressing a delete key.

M_READ

Generated by the Stream head and sent downstream for a **`read(2)`** system call if no messages are waiting to be read at the Stream head and if read notification has been enabled. Read notification is enabled with the `SO_MREADON` flag of the `M_SETOPTS` message and disabled by use of the `SO_MREADOFF` flag.

The message content is set to the value of the *nbyte* parameter (the number of bytes to be read) in the **`read(2)`** call.

`M_READ` is intended to notify modules and drivers of the occurrence of a `read`. It is also intended to support communication between Streams that reside in separate processors. The use of the `M_READ` message is developer dependent. Modules may take specific action and pass on or free the `M_READ` message. Modules that do not recognize this message must pass it on. All other drivers may or may not take action and then free the message.

This message cannot be generated by a user-level process and should not be generated by a module or driver. It is always discarded if passed to the Stream head.

M_START and M_STOP

Request devices to start or stop their output. They are intended to produce momentary pauses in a device's output, not to turn devices on or off.

The message format is not defined by STREAMS and its use is developer dependent. These messages may be considered special cases of an `M_CTL` message. These messages cannot be generated by a user-level process and each is always discarded if passed to the Stream head.

M_STARTI and M_STOPI

Similar to `M_START` and `M_STOP` except that `M_STARTI` and `M_STOPI` are used to start and stop input.

6
Overview: STREAMS Modules and Drivers

6
Overview: STREAMS Modules and Drivers

Introduction

Modules and drivers are processing elements in STREAMS. A Stream device driver is similar to a conventional UNIX system driver. It is opened like a conventional driver and is responsible for the system interface to the device.

STREAMS modules and drivers are structurally similar. The call interfaces to driver routines are identical to interfaces used for modules. Drivers and modules must declare `streamtab`, `qinit`, and `module_info` structures. Within the STREAMS mechanism, drivers are required elements, but modules are optional. However, in the STREAMS-based pipe mechanism only the Stream head is required.

One consequence of the flexibility and modularity of STREAMS is the tendency to split up the processing formerly done by drivers and distribute it among a number of STREAMS modules and a driver. For example, where a TTY driver might directly call a line discipline routine, a STREAMS configuration would isolate the line discipline processing in a module. While STREAMS drivers may be cleaner and less complicated to write, the driver writer may have the additional responsibility of writing modules as well.

Furthermore, the user-level program establishing access to a STREAMS device has the option of building a stream with whatever modules are available. This flexibility implies that a module's functionality must be well documented by the developer so that an applications programmer can be confident of correctly including it in a stream.

Differences Between Modules and Drivers

The following list summarizes the major differences between STREAMS modules and drivers:

- Drivers are always positioned at the end of a stream. Consequently, for hardware devices, the driver must handle interrupts, but modules do not.
- Drivers may be at the stream end for more than one stream at a time, whereas a module can only be part of one stream.
- A module is not assigned a special device file and must be pushed onto a *stream*, while a driver is opened.
- Modules have no user context, and cannot access the `user` structure. This is also true for drivers, with the exception of the **`open(D3)`** and **`close(D3)`** routines.

Similarities Between Modules and Drivers

While the differences are significant, the similarities between modules and drivers are also important.

- Both modules and drivers are built on `queue` structures.
- The manner in which the entry point routines are called is similar. STREAMS devices are considered a subset of character devices, so they are accessed through the **`cdevsw`** switch table. If the device were a simple character device, the entry point routine would be looked up in this table. If a STREAMS device has a non-null value in the `d_str` field of the `cdevsw` table, the designated **`streamtab(D4)`** table should be used instead. The `streamtab` structure, in turn, contains pointers to `qinit` structures defining the entry points.
- STREAMS drivers and modules both have `streamtab` tables for access to routines, and so both have the same choice of routines, most of which are different from those found in the `cdevsw` table for character devices. See "STREAMS Entry Points" for a discussion of these routines.
- Both modules and drivers pass the same objects, (pointers to `queues` and to messages). Consequently, both modules and drivers make extensive use of the STREAMS-specific functions described in the *Device Driver Reference* manual.

User context is not generally available to STREAMS module procedures and drivers. The exception is during execution of the **`open`** and **`close`** routines. Driver and module **`open`** and **`close`** routines have user context and may access the `u_area` structure, although this is discouraged. The **`open`** and **`close`** routines may use blocking primitives as defined in the DDI.

NOTE

STREAMS driver and module **`put`** procedures and **`service`** procedures have no user context. They cannot access the `u_area` structure of a process and must not sleep.

The module and driver **`open`**/**`close`** interface has been modified for UNIX System V Release 4. However, the system defaults to UNIX System V Release 3.0 interface unless *prefix*`devflag` is defined. Examples and descriptions in this chapter reflect the Release 4 interface.

This release of the operating system does not support code that does not conform to the DDI/DKI standard.

Module and Driver Declarations

A module and driver contains, at a minimum, declarations of the form as shown in Screen 6-1:

```
#include <sys/types.h>    /* required in all modules and drivers */
#include <sys/stream.h>   /* required in all modules and drivers */
#include <sys/param.h>
#include <sys/cred.h>
#include <sys/synch.h>
#include <sys/ddi.h>      /* required in all modules and drivers */

static struct module_info rminfo = { 0x08, “mod”, 0, INFPSZ, 0, 0 };
static struct module_info wminfo = { 0x08, “mod”, 0, INFPSZ, 0, 0 };
static int modopen(), modput(), modclose();

static struct qinit rinit = {
    modput, NULL, modopen, modclose, NULL, &rminfo, NULL };

static struct qinit winit = {
    modput, NULL, NULL, NULL, NULL, &wminfo, NULL };

struct streamtab modinfo = { &rinit, &winit, NULL, NULL };

int moddevflag = D_MT;
```

Screen 6-1. Module and Driver Declarations

The contents of these declarations are constructed for the null module example in this section. This module does no processing. Its only purpose is to show linkage of a module into the system. The descriptions in this section are general to all STREAMS modules and drivers unless they specifically reference the example.

The declarations shown are the header set; the read and write queue (`rminfo` and `wminfo`) `module_info` structures; the module open, read/write-put, and close procedures; the read and write (`rinit` and `winit`) `qinit` structures; and the `streamtab` structure.

The header files **`ddi.h`**, **`types.h`**, and **`stream.h`**, are always required for modules and drivers. The header file **`param.h`**, contains definitions for `NULL` and other values for STREAMS modules and drivers.

NOTE

When configuring a STREAMS module or driver the `streamtab` structure must be externally accessible. The `streamtab` structure name must be the prefix appended with `info`. Also, the driver flag must be externally accessible. The flag name must be the prefix appended with `devflag`.

The `streamtab` contains `qinit` values for the read and write queues. The `qinit` structures in turn point to a `module_info` and an optional `module_stat` structure. The two required structures are shown in Screen 6-2:

```
struct qinit {
      int  (*qi_putp)();              /* put procedure */
      int  (*qi_srvp)();              /* service procedure */
      int  (*qi_qopen)();             /* called on each open or a push */
      int  (*qi_qclose)();            /* called on last close or a pop */
      int  (*qi_qadmin)();            /* reserved for future use */
      struct module_info  *qi_minfo;  /* information structure */
      struct module_stat  *qi_mstat;  /* statistics structure - optional */
};

struct module_info {
      ushort_t   mi_idnum;        /* module ID number */
      char       *mi_idname;      /* module name */
      long       mi_minpsz;       /* min packet size, for developer use */
      long       mi_maxpsz;       /* max packet size, for developer use */
      ulong_t    mi_hiwat;        /* hi-water mark */
      ulong_t    mi_lowat;        /* lo-water mark */
};
```

Screen 6-2. Required Structures

The `qinit` contains the queue procedures: **`put`**, **`service`**, **`open`**, and **`close`**. All modules and drivers with the same `streamtab` (that is, the same `fmodsw` or `cdevsw` entry) point to the same upstream and downstream `qinit` structure(s). The structure is meant to be software read-only, as any changes to it affect all instantiations of that module in all Streams. Pointers to the **`open`** and **`close`** procedures must be contained in the read `qinit` structure. These fields are ignored on the write-side. Our example has no **`service`** procedure on the read-side or write-side.

The `module_info` contains identification and limit values. All queues associated with a certain driver/module share the same `module_info` structures. The `module_info` structures define the characteristics of that driver/module's queues. As with the `qinit`, this structure is intended to be software read-only. However, the four limit values (`q_minpsz`, `q_maxpsz`, `q_hiwat`, `q_lowat`) are copied to a `queue` structure where they are modifiable. In the example, the flow control high- and low-water marks are zero since there is no **`service`** procedure and messages are not queued in the module.

Three names are associated with a module:

- The character string in `fmodsw`.
- The prefix for `streamtab`, used in configuring the module.
- The module name field in the `module_info` structure. The module name must match the entry for the module in the device driver/module configuration file. The name of this configuration file is machine specific; it is described in either the **`master(4)`** or **`mdevice(4)`** manual page, depending on your system.

Each module ID number and module name should be unique in the system. The module ID number is currently used only in logging and tracing. It is `0x08` in the example.

Minimum and maximum packet sizes are intended to limit the total number of characters contained in `M_DATA` messages passed to this queue. These limits are advisory except for the Stream head. For certain system calls that write to a Stream, the Stream head observes the packet sizes set in the write queue of the module immediately below it. Otherwise, the use of packet size is developer-dependent. In the example, INFPSZ indicates unlimited size on the read-side.

The module_stat is optional. Currently, there is no STREAMS support for statistical information gathering.

Null Module Example

The null module procedures are shown in Screen 6-3:

```
static int modopen(queue_t *q, dev_t *devp, int flag,
           int sflag, cred_t *credp)
{
    qprocson(q);/* enables put and srv routines */
    /* return success */
    return 0;
}

static int modput(queue_t *q, mblk_t *mp)
{
    putnext(q, mp);/* pass message through */
}

/* Note: we only need one put procedure that can be used for both
 * read-side and write-side.
 */

static int modclose(queue_t *q, int flag, cred_t *credp)
{
    qprocsoff(q);/* disables put and srv routines */
    return 0;
}
```

Screen 6-3. Null Module Procedure

The form and arguments of these procedures are the same in all modules and all drivers. Modules and drivers can be used in multiple Streams and their procedures must be re-entrant.

modopen illustrates the open call arguments and return value. The arguments are the read queue pointer (*q*), the pointer (*devp*) to the major/minor device number, the file flags (*flag*, defined in **sys/file.h**), the Stream open flag (*sflag*), and a pointer to a credentials structure (*credp*). The Stream open flag can take on the following values:

MODOPEN	Normal module open
0	Normal driver open
CLONEOPEN	Clone driver open

The return value from open is 0 for success and an error number for failure. If a driver is called with the CLONEOPEN flag, the device number pointed to by the *devp* should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major and minor device number). The **open** procedure for a module is called on the first **I_PUSH** and on all later **open** calls to the same Stream. During a push, a nonzero return value causes the **I_PUSH** to fail and the module to be removed from the Stream. If an error is returned by a module during an **open** call, the **open** fails, but the

Stream remains intact. In the null module example, **modopen** simply enables its **put** (and **service**) procedure(s) using **qprocson** and returns successfully.

If the Enhanced Security Utilities are installed, the module **open** fails if the calling process does not have the P_DEV privilege in its working set (see **intro(2)** for a list of privileges.)

On Enhanced Security versions of the system, permission checks in module and driver **open** routines should be done with the **drv_priv** routine; there is no need to check if `u.u_uid == 0`. This and the **suser** routine have been replaced with:

```
error = drv_priv(credp);
if (error)      /* not privileged */
    return errno;
```

modput illustrates the common interface to **put** procedures. The arguments are the read or write `queue` pointer, as appropriate, and the message pointer. The **put** procedure in the appropriate side of the queue is called when a message is passed from upstream or downstream. The **put** procedure has no return value, but it is defined as `int` (). In the example, no message processing is done. All messages are forwarded using **putnext**. See the *Device Driver Reference.* **putnext** calls the **put** procedure of the next queue in the proper direction.

The close routine is only called on an **I_POP ioctl** for modules, or on the last **close** call of the Stream for drivers. The arguments are the read queue pointer, the file flags as in **modopen**, and a pointer to a credentials structure.

qprocsoff is called to disable the **put** (and **service**) procedures.

The return value is `0` on success. A failure of **close** is ignored by the system.

Module and Driver ioctls

STREAMS is an addition to the UNIX system traditional character input/output (I/O) mechanism. In this section, the phrases “character I/O mechanism” and “I/O mechanism” refer only to that part of the mechanism that pre-existed STREAMS.

The character I/O mechanism handles all **ioctl(2)** system calls in a transparent manner. The kernel expects all **ioctl**s to be handled by the device driver associated with the character special file on which the call is sent. All **ioctl** calls are sent to the driver, which is expected to do all validation and processing other than file descriptor validity checking. The operation of any specific **ioctl** is dependent on the device driver. If the driver requires data to be transferred in from user space, it uses the kernel **copyin** function. It may also use **copyout** to transfer out any data results back to user space.

With STREAMS, there are a number of differences from the character I/O mechanism that affect **ioctl** processing.

First, there are a set of generic STREAMS **ioctl** command values (see **ioctl(2)**) recognized and processed by the Stream head. These are described in **streamio(7)**. The operation of the generic STREAMS **ioctl**s are generally independent of the presence of any specific module or driver on the Stream.

The second difference is the absence of user context in a module and driver when the information associated with the **ioctl** is received. This prevents use of **copyin** or **copyout** by the module. This also prevents the module and driver from associating any kernel data with the currently running process. (It is likely that by the time the module or driver receives the **ioctl**, the process generating it may no longer be running.)

A third difference is that for the character I/O mechanism, all **ioctl**s are handled by the single driver associated with the file. In STREAMS, there can be multiple modules on a Stream and each one can have its own set of **ioctl**s. The **ioctl**s that can be used on a Stream can change as modules are pushed and popped.

STREAMS provides the capability for user processes to perform control functions on specific modules and drivers in a Stream with **ioctl** calls. Most **streamio(7) ioctl** commands go no further than the Stream head. They are fully processed there and no related messages are sent downstream. However, certain commands and all unrecognized commands cause the Stream head to create an M_IOCTL message that includes the **ioctl** arguments, and send the message downstream to be received and processed by a specific module or driver. The M_IOCTL message is the initial message type that carries **ioctl** information to modules. Other message types are used to complete the **ioctl** processing in the Stream. In general, each module must uniquely recognize and take action on specific M_IOCTL messages.

STREAMS **ioctl** handling is equivalent to the transparent processing of the character I/O mechanism. STREAMS modules and drivers can process **ioctl**s generated by applications that are implemented for a non-STREAMS environment.

General ioctl Processing

STREAMS blocks a user process that issues an **ioctl** and causes the Stream head to generate an M_IOCTL message. The process remains blocked until either

- A module or a driver responds with an M_IOCACK (*ack*, positive acknowledgment) message or an M_IOCNAK (*nak*, negative acknowledgment) message.
- No message is received and the request “times out.”
- The **ioctl** is interrupted by the user process.
- An error condition occurs.

For the **ioctl I_STR**, the timeout period can be a user-specified interval or a default. For the other **M_IOCTL ioctl**s, the default value (infinite) is used.

For an **I_STR**, the STREAMS module or driver that generates a positive acknowledgment message can also return data to the process in that message. An alternate means to return data is provided with transparent **ioctl**s. If the Stream head does not receive a positive or negative acknowledgment message in the specified time, the **ioctl** call fails.

A module that receives an unrecognized M_IOCTL message should pass it on unchanged. A driver that receives an unrecognized M_IOCTL should produce a negative acknowledgment.

The form of an M_IOCTL message is a single M_IOCTL message block followed by zero or more M_DATA blocks. The M_IOCTL message block contains an iocblk structure, defined in **sys/stream.h**. For details, see **iocblk(D4)**.

For an **I_STR ioctl**, ioc_cmd (in the iocblk structure) contains the command supplied by the user in the strioctl structure defined in **streamio(7)**.

If a module or driver determines an M_IOCTL message is in error for any reason, it must produce the negative acknowledgment message by setting the message type to M_IOCNAK and sending the message upstream. No data or a return value can be sent to a user in this case. If ioc_error (in iocblk) is set to 0, the Stream head causes the **ioctl** call to fail with EINVAL. The driver has the option of setting ioc_error to an alternate error number if desired.

NOTE

ioc_error can be set to a nonzero value in both M_IOCACK and M_IOCNAK. This causes that value to be returned as an error number to the process that sent the **ioctl**.

If a module wants to look at what **ioctl**s of other modules are doing, the module should not look for a specific M_IOCTL on the write-side but look for M_IOCACK or M_IOCNAK on the read-side. For example, the module sees TCSETS (see **termios(7)**) going down and wants to know what is being set. The module should look at it and save the data but not use it. The read-side processing knows that the module is waiting for an answer for the **ioctl**. When the read-side processing sees an *ack* or *nak* next time, it checks if it is the same **ioctl** (here TCSETS) and if it is, the module may use the data previously saved.

The two STREAMS **ioctl** mechanisms, **I_STR** and transparent, are described next. (Here, **I_STR** means the **streamio(7) I_STR** command and implies the related STREAMS processing unless noted otherwise). **I_STR** has a restricted format and restricted addressing for transferring **ioctl**-related data between user and kernel space. It requires only a single pair of messages to complete **ioctl** processing. The transparent mechanism is more general and has almost no restrictions on **ioctl** data format and addressing. The transparent mechanism generally requires that multiple pairs of messages be exchanged between the Stream head and module to complete the processing.

I_STR ioctl Processing

The **I_STR ioctl** provides a capability for user applications to do module and driver control functions on STREAMS files. **I_STR** allows an application to specify the **ioctl** timeout. It requires that all user **ioctl** data (to be received by the destination module) be placed in a single block that is pointed to from the user strioctl structure. The module can also return data to this block.

If the module is looking at, for example, the **TCSETS**/**TCGETS** group of **ioctl** calls as they pass up or down a Stream, it must never assume that because TCSETS comes down that it actually has a data buffer attached to it. The user may have formed TCSETS as an **I_STR** call and accidentally given a null data buffer pointer. You should always check b_cont to see if it is NULL before using it as an index to the data block that goes with M_IOCTL messages.

The **TCGETA** call, if formed as an **I_STR** call with a data buffer pointer set to a value by the user, always has a data buffer attached to b_cont from the main message block. Check to see that the **ioctl** message does not have a buffer attached before allocating a new buffer and assigning b_cont to point at it. If you do not, the original buffer will be lost.

Figure 6-4 illustrates processing associated with an **I_STR ioctl**. **lpdoioctl** is called to process trapped M_IOCTL messages:

```
TYPE
lpdoioctl(struct lp *lp, mblk_t *mp)
{
     struct iocblk *iocp;
     queue_t *q;

     q = lp->qptr;    /*its own write queue*/

     /* 1st block contains iocblk structure */
     iocp = (struct iocblk *)mp->b_rptr;

     switch (iocp->ioc_cmd) {
     case SET_OPTIONS:
          /* Count should be exactly one short's worth
          (for this example) */
          if (iocp->ioc_count != sizeof(short))
               goto iocnak;
          if (mp->b_cont == NULL)
               goto lognak; /* not shown in this example */
          /* Actual data is in 2nd message block */
          lpsetopt(lp, *(short *)mp->b_cont->b_rptr);
                   /*hypothetical routine*/

          /* ACK the ioctl */
          mp->b_datap->db_type = M_IOCACK;
          iocp->ioc_count = 0;
          qreply(q, mp);
          break;
     default:
     iocnak:
          /* NAK the ioctl */
          mp->b_datap->db_type = M_IOCNAK;
          qreply(q, mp);
     }
}
```

Screen 6-4. I_STR ioctl Processing

lpdoioctl illustrates driver M_IOCTL processing, which also applies to modules. However, at case default, a module would not *nak* an unrecognized command, but would pass the message on. In this example, only one command is recognized, SET_OPTIONS. ioc_count contains the number of user-supplied data bytes. For this example, it must equal the size of a short. The user data is sent directly to the printer interface using **lpsetopt**. Next, the M_IOCTL message is changed to type M_IOCACK and the ioc_count field is set to zero to show that no data is to be returned to the user. Finally, the message is sent upstream using **qreply**. If ioc_count was left nonzero, the Stream head copies that many bytes from the 2nd through Nth message blocks into the user buffer.

Transparent ioctl Processing

The transparent STREAMS **`ioctl`** mechanism allows application programs to perform module and driver control functions with **`ioctl`**s other than **`I_STR`**. It is intended to transparently support applications developed before the introduction of STREAMS, and alleviates the need to recode and recompile the user level software to run over STREAMS files.

The mechanism extends the data transfer capability for STREAMS **`ioctl`** calls beyond that provided in the **`I_STR`** form. Modules and drivers can transfer data between their kernel space and user space in any **`ioctl`** that has a value of the `command` argument not defined in **`streamio(7)`**. These **`ioctl`**s are known as transparent **`ioctl`**s to differentiate them from the **`I_STR`** form. Transparent processing support is necessary when existing user level applications perform **`ioctl`**s on a non-STREAMS character device and the device driver is converted to STREAMS. The **`ioctl`** data can be in any format mutually understood by the user application and module.

The transparent mechanism also supports STREAMS applications that want to send **`ioctl`** data to a driver or module in a single call, where the data may not be in a form readily embedded in a single user block. For example, the data may be contained in nested structures, different user space buffers, and so forth.

This mechanism is needed because user context does not exist in modules and drivers when **`ioctl`** processing occurs. This prevents them from using the kernel **`copyin`** and **`copyout`** functions. For example, consider the following **`ioctl`** call:

`ioctl` `(`*stream_fildes*`,` *user_command*`,` &*ioctl_struct*`);`

where *ioctl_struct* is a structure containing the members:

```
int stringlen;  /* string length */
char *string;
struct other_struct *other1;
```

To read (or write) the elements of *ioctl_struct*, a module would have to do a series of **`copyin`**/**`copyout`** calls using pointer information from a prior **`copyin`** to transfer additional data. A non-STREAMS character driver could directly execute these copy functions because user context exists during all PowerMAX OS system calls to the driver. However, in STREAMS, user context is only available to modules and drivers in their open and close routines.

The transparent mechanism enables modules and drivers to request that the Stream head do a **`copyin`** or **`copyout`** on their behalf to transfer **`ioctl`** data between their kernel space and various user space locations. The related data is sent in message pairs exchanged between the Stream head and the module. A pair of messages is required so that each transfer can be acknowledged. In addition to `M_IOCTL`, `M_IOCACK`, and `M_IOCNAK` messages, the transparent mechanism also uses `M_COPYIN`, `M_COPYOUT`, and `M_IOCDATA` messages.

The general processing by which a module or a driver reads data from user space for the transparent case involves pairs of request/response messages, as follows:

1. The Stream head does not recognize the `command` argument of an **`ioctl`** call and creates a transparent `M_IOCTL` message. The `iocblk` structure has a `TRANSPARENT` indicator containing the value of the `arg` argument

in the call. It sends the `M_IOCTL` message downstream. See “Transparent ioctl Messages” for more details.

2. A module receives the `M_IOCTL` message, recognizes the `ioc_cmd`, and determines that it is `TRANSPARENT`.
3. If the module requires user data, it creates an `M_COPYIN` message to request a **`copyin`** of user data. The message contains the address of user data to copy in and how much data to transfer. It sends the message upstream.
4. The Stream head receives the `M_COPYIN` message and uses the contents to **`copyin`** the data from user space into an `M_IOCDATA` response message that it sends downstream. The message also contains an indicator of whether the data transfer succeeded. The **`copyin`** might fail, for instance, because of an `EFAULT` condition. See **`intro(2)`**.
5. The module receives the `M_IOCDATA` message and processes its contents.

   The module may use the message contents to generate another `M_COPYIN`. Steps 3 through 5 may be repeated until the module has requested and received all the user data to be transferred.
6. When the module completes its data transfer, it does the **`ioctl`** processing and sends an `M_IOCACK` message upstream to notify the Stream head that **`ioctl`** processing has successfully completed.

Writing data from a module to user space is similar except that the module uses an `M_COPYOUT` message to request the Stream head to write data into user space. In addition to length and user address, the message includes the data to be copied out. In this case, the `M_IOCDATA` response will not contain user data, only show success or failure.

The module may intermix `M_COPYIN` and `M_COPYOUT` messages in any order. However, each message must be sent one at a time; the module must receive the associated `M_IOCDATA` response before any subsequent `M_COPYIN`/`M_COPYOUT` request or *ack*/*nak* message is sent upstream. After the last `M_COPYIN`/`M_COPYOUT` message, the module must send an `M_IOCACK` message (or `M_IOCNAK` for a detected error condition).

NOTE

For a transparent `M_IOCTL`, user data cannot be returned with an `M_IOCACK` message. The data must have been sent with a preceding `M_COPYOUT` message.

Transparent ioctl Messages

The form of the `M_IOCTL` message generated by the Stream head for a transparent **`ioctl`** is a single `M_IOCTL` message block followed by one `M_DATA` block. The form of the `iocblk` structure in the `M_IOCTL` block is the same as described under “General ioctl Processing.” However, `ioc_cmd` is set to the value of the `command` argument in the **`ioctl`** system call and `ioc_count` is set to `TRANSPARENT`, defined in **`sys/stream.h`**. `TRANSPARENT` distinguishes the case where an **`I_STR`** **`ioctl`** may specify a value of `ioc_cmd` equivalent to the `command` argument of a transparent **`ioctl`**. The `M_DATA` block of the message contains the value of the `arg` parameter in the call.

NOTE

Modules that process a specific `ioc_cmd` that did not validate the `ioc_count` field of the `M_IOCTL` message will break if transparent `ioctl`s with the same command are done from user space.

Transparent ioctl Examples

The following are three examples of transparent **`ioctl`** processing. Screen 6-5 and Screen 6-6 illustrate `M_COPYIN`. Screen 6-7 illustrates `M_COPYOUT`. Screen 6-8 and Screen 6-9 show a more complex example with state transitions combining both `M_COPYIN` and `M_COPYOUT`.

M_COPYIN Example

In this example, the contents of a user buffer are transferred into the kernel as part of an **`ioctl`** call of the form

`ioctl``(` *fd*`, SET_ADDR, &`*bufadd*`)`

where *bufadd* is a structure declared as

```
struct address {
   int ad_len;/* buffer length in bytes */
   caddr_t ad_addr;/* buffer address */
};
```

This requires two pairs of messages (request/response) following receipt of the `M_IOCTL` message. The first will **`copyin`** the structure and the second will **`copyin`** the buffer. Screen 6-5 illustrates processing that supports only the transparent form of **`ioctl`**. ***`xxx-wput`*** is the write-side **`put`** procedure for the module or driver ***`xxx`***:

```
struct address {      /* same members as in user space */
    int ad_len;           /* length in bytes */
    caddr_t ad_addr;      /* buffer address */
};

/* state values (overloaded in private field) */
#define GETSTRUCT     0        /* address structure */
#define GETADDR       1        /* byte string from ad_addr */

xxxwput(queue_t *q, mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;

    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *)mp->b_rptr;
        switch (iocbp->ioc_cmd) {

        case SET_ADDR:

            if (iocbp->ioc_count != TRANSPARENT) {/* fail if I_STR */
                if (mp->b_cont) {     /* return buffer to pool ASAP */
                    freemsg(mp->b_cont);
                    mp->b_cont = NULL;
                }
                mp->b_datap->db_type = M_IOCNAK;/* EINVAL */
                qreply(q, mp);
                break;
            }
            /* Reuse M_IOCTL block for M_COPYIN request */

            cqp = (struct copyreq *)mp->b_rptr;

            /* Get user space structure address from linked M_DATA block */

            cqp->cq_addr = (caddr_t) *(long *)mp->b_cont->b_rptr;
            freemsg(mp->b_cont);          /* MUST free linked blocks */
            mp->b_cont = NULL;
            cqp->cq_private = (mblk_t *)GETSTRUCT;  /* to identify response */

            /* Finish describing M_COPYIN message */

            cqp->cq_size = sizeof(struct address);
            cqp->cq_flag = 0;
            mp->b_datap->db_type = M_COPYIN;
            mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
            qreply(q, mp);
            break;
```

Screen 6-5. Request/Response Messages

```
        default: /* M_IOCTL not for us */
            /* if module, pass on */
            /* if driver, nak ioctl */
            break;

        }   /* switch (iocbp->ioc_cmd) */

        break;

    case M_IOCDATA:
        xxxioc(q, mp);/* all M_IOCDATA processing done here */
        break;
        .
        .
        .
    }    /* switch (mp->b_datap->db_type) */
}
```

xxx**wput** verifies that the `SET_ADDR` is `TRANSPARENT` to avoid confusion with an **I_STR** **`ioctl`**, which uses a value of `ioc_cmd` equivalent to the `command` argument of a transparent **`ioctl`**. When sending an `M_IOCNAK`, freeing the linked `M_DATA` block is not mandatory as the Stream head frees it. However, this returns the block to the buffer pool more quickly.

In this and all the following examples in this section, the message blocks are reused to avoid the overhead of deallocating and allocating.

NOTE

The Stream head guarantees that the size of the message block containing an `iocblk` structure is large enough also to hold the `copyreq` and `copyresp` structures.

`cq_private` is set to contain state information for `ioctl` processing (tells us what the subsequent `M_IOCDATA` response message contains). Keeping the state in the message makes the message self-describing and simplifies the **`ioctl`** processing. `M_IOCDATA` processing is done in *xxx*`ioc`. Two `M_IOCDATA` types are processed, `GETSTRUCT` and `GETADDR`:

```
xxxioc(queue_t *q, mblk_t *mp)/* M_IOCDATA processing */
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct address *ap;

    csp = (struct copyresp *)mp->b_rptr;
    iocbp = (struct iocblk *)mp->b_rptr;
    switch (csp->cp_cmd) {/* validate M_IOCDATA for this module */

    case SET_ADDR:
        if (csp->cp_rval) {/* GETSTRUCT or GETADDR failed */
            freemsg(mp);
            return;
        }
        switch ((int)csp->cp_private) {/* determine state */

        case GETSTRUCT:/* user structure has arrived */
            mp->b_datap->db_type = M_COPYIN; /* reuse M_IOCDATA block */
            cqp = (struct copyreq *)mp->b_rptr;
            ap = (struct address *)mp->b_cont->b_rptr; /* user structure */
            cqp->cq_size = ap->ad_len;/* buffer length */
            cqp->cq_addr = ap->ad_addr;/* user space buffer address */
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            cqp->cq_flag = 0;
            csp->cp_private = (mblk_t *)GETADDR;/* next state */
            qreply(q, mp);
            break;

        case GETADDR: /* user address is here */
            if (xxx_set_addr(mp->b_cont) == FAILURE){/*hypothetical routine*/
                mp->b_datap->db_type = M_IOCNAK;
                iocbp->ioc_error = EIO;
            } else {
                mp->b_datap->db_type = M_IOCACK;/* success */
                iocbp->ioc_error = 0;/* may have been overwritten */
                iocbp->ioc_count = 0;/* may have been overwritten */
                iocbp->ioc_rval = 0;/* may have been overwritten */
            }
            mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            qreply(q, mp);
            break;

        default:  /* invalid state: can't happen */
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            mp->b_datap->db_type = M_IOCNAK;
            mp->b_wptr = mp->rptr + sizeof(struct iocblk);
```

Screen 6-6. GETSTRUCT and GETADDR

```
                iocbp->ioc_error = EINVAL;  /* may have been overwritten */
                qreply(q, mp);
                ASSERT (0);/* panic if debugging mode */
                break;
            }
            break;/* switch (cp_private) */

        default:  /* M_IOCDATA not for us */
            /* if module, pass message on */
            /* if driver, free message */
            break;
        }   /* switch (cp_cmd) */
}
```

***xxx*_set_addr** is a routine (not shown in the example) that processes the user address from the **ioctl**. Because the message block has been reused, the fields that the Stream head examines (denoted by *may have been overwritten*) must be cleared before sending an M_IOCNAK.

M_COPYOUT Example

In this example, the user wants option values for this Stream device to be placed into the user's options structure (see beginning of example code). This can be done by use of a transparent **ioctl** call of the form

ioctl(*fd*, GET_OPTIONS, &*optadd*)

or, alternately, by use of a **streamio** call

ioctl(*fd*, **I_STR**, &*opts_strioctl*)

In the first case, *optadd* is declared struct options. In the **I_STR** case, *opts_strioctl* is declared struct strioctl, where *opts_strioctl*.ic_dp points to the user options structure.

Screen 6-7 illustrates support of both the **I_STR** and transparent forms of an **ioctl**. The transparent form requires a single M_COPYOUT message following receipt of the M_IOCTL to **copyout** the contents of the structure. ***xxx*wput** is the write-side **put** procedure for module or driver ***xxx***:

```
struct options {/* same members as in user space */
    int  op_one;
    int  op_two;
    shortop_three;
    long op_four;
};

xxxwput(queue_t *q, mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    int transparent = 0;

    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *)mp->b_rptr;
        switch (iocbp->ioc_cmd) {

        case GET_OPTIONS:

            if (iocbp->ioc_count == TRANSPARENT) {
                transparent = 1;
                cqp = (struct copyreq *)mp->b_rptr;
                cqp->cq_size = sizeof(struct options);
                /* Get structure address from linked M_DATA block */
                cqp->cq_addr = (caddr_t) *(long *)mp->b_cont->b_rptr;
                cqp->cq_flag = 0;

                /* No state necessary - we will only ever get one
                 * M_IOCDATA from the Stream head indicating success
                 * or failure for the copyout */
            }
            if (mp->b_cont)
                freemsg(mp->b_cont);/* overwritten */
            if ((mp->b_cont = allocb(sizeof(struct options),
            BPRI_MED)) == NULL) {
                mp->b_datap->db_type = M_IOCNAK;
                iocbp->ioc_error = EAGAIN;
                qreply(q, mp);
                break;
            }
            xxx_get_options(mp->b_cont);   /* hypothetical routine */
            if (transparent) {
                mp->b_datap->db_type = M_COPYOUT;
                mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
            } else {
                mp->b_datap->db_type = M_IOCACK;
                iocbp->ioc_count = sizeof(struct options);
```

Screen 6-7. I_STR and Transparent ioctl

```
            }
            qreply(q, mp);
            break;

        default: /* M_IOCTL not for us */
            /* if module, pass on; if driver, nak ioctl */

            break;
        } /* switch (iocbp->ioc_cmd) */
        break;

    case M_IOCDATA:
        csp = (struct copyresp *)mp->b_rptr;
        if (csp->cmd != GET_OPTIONS) { /* M_IOCDATA not for us */
            /* if module, pass on; if driver, free message */

            break;
        }
        if (csp->cp_rval) {
            freemsg(mp);/* failure */
            return;
        }
        /* Data successfully copied out, ack */

        mp->b_datap->db_type = M_IOCACK;/* reuse M_IOCDATA for ack */
        mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
        iocbp->ioc_error = 0;/* may have been overwritten */
        iocbp->ioc_count = 0;/* may have been overwritten */
        iocbp->ioc_rval = 0;/* may have been overwritten */
        qreply(q, mp);
        break;
        .
        .
        .
    } /* switch (mp->b_datap->db_type) */
}
```

Bidirectional Transfer Example

Screen 6-8 and Screen 6-9 illustrate bidirectional data transfer between the kernel and user space during transparent **`ioctl`** processing. It also shows how more complex state information can be used.

The user wants to send and receive data from user buffers as part of a transparent **`ioctl`** call of the form

`ioctl``(`*fd*`,` *XXX*`_IOCTL,` &*addr_xxxdata*`)`

The user `addr_`*xxx*`data` structure defining the buffers is declared as `struct` *xxx*`data`, as shown. This requires three pairs of messages following receipt of the `M_IOCTL` message:

1. to **`copyin`** the structure
2. to **`copyin`** one user buffer
3. to **`copyout`** the second user buffer

***`xxx`*`wput`** is the write-side **`put`** procedure for the module or driver xxx:

```
struct xxxdata {          /* same members in user space */
    int      x_inlen;         /* number of bytes copied in */
    caddr_t  x_inaddr;        /* buffer address of data copied in */
    int      x_outlen;        /* number of bytes copied out */
    caddr_t  x_outaddr;       /* buffer address of data copied out */
};
/*  State information for ioctl processing */
struct state {
    int      st_state;        /* see below */
    struct xxxdatast_data;    /* see above */
};
/* state values */

#define GETSTRUCT     0       /* get xxxdata structure */
#define GETINDATA     1       /* get data from x_inaddr */
#define PUTOUTDATA    2       /* get response from M_COPYOUT */

static int
xxxwput(queue_t *q, mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct state *stp;
    mblk_t *tmp;
    switch (mp->b_datap->db_type) {
        .
        .
        .
    case M_IOCTL:
        iocbp = (struct iocblk *)mp->b_rptr;
        switch (iocbp->ioc_cmd) {

        case XXX_IOCTL:
            if (iocbp->ioc_count != TRANSPARENT) {/* fail if I_STR */
                if (mp->b_cont) {  /* return buffer to pool ASAP */
                    freemsg(mp->b_cont);
                    mp->b_cont = NULL;
                }
                mp->b_datap->db_type = M_IOCNAK;/* EINVAL */
                qreply(q, mp);
                break;
            }
            /* Reuse M_IOCTL block for M_COPYIN request */

            cqp = (struct copyreq *)mp->b_rptr;

            /* Get structure's user address from linked M_DATA block */

            cqp->cq_addr = (caddr_t) *(long *)mp->b_cont->b_rptr;
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
```

Screen 6-8. Write-Side put Procedure

```
            /* Allocate state buffer */

            if ((tmp = allocb(sizeof(struct state), BPRI_MED)) == NULL) {
                mp->b_datap->db_type = M_IOCNAK;
                iocbp->ioc_error = EAGAIN;
                qreply(q, mp);
                break;
            }
            tmp->b_wptr += sizeof(struct state);
            stp = (struct state *)tmp->b_rptr;
            stp->st_state = GETSTRUCT;
            cqp->cq_private = tmp;

            /* Finish describing M_COPYIN message */

            cqp->cq_size = sizeof(struct xxxdata);
            cqp->cq_flag = 0;
            mp->b_datap->db_type = M_COPYIN;
            mp->b_wptr = mp->b_rptr + sizeof(struct copyreq);
            qreply(q, mp);
            break;
        default: /* M_IOCTL not for us */
            /* if module, pass on */
            /* if driver, nak ioctl */
            break;

        } /* switch (iocbp->ioc_cmd) */
        break;

    case M_IOCDATA:
        xxxioc(q, mp);/* all M_IOCDATA processing done here */
        break;
        .
        .
        .
    } /* switch (mp->b_datap->db_type) */
}
```

***xxx*`wput`** allocates a message block to contain the state structure and reuses the `M_IOCTL` to create an `M_COPYIN` message to read in the *xxx*`data` structure. `M_IOCDATA` processing is done in ***xxx*`ioc`**:

```
xxxioc(queue_t *q, mblk_t *mp)  /* M_IOCDATA processing */
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct state *stp;
    mblk_t *xxx_indata();

    csp = (struct copyresp *)mp->b_rptr;
    iocbp = (struct iocblk *)mp->b_rptr;
    switch (csp->cp_cmd) {

    case XXX_IOCTL:
        if (csp->cp_rval) {/* failure */
            if (csp->cp_private)/* state structure */
                freemsg(csp->cp_private);
            freemsg(mp);
            return;
        }
        stp = (struct state *)csp->cp_private->b_rptr;
        switch (stp->st_state) {
        case GETSTRUCT:/* xxxdata structure copied in */
            /* save structure */

            stp->st_data = *(struct xxxdata *)mp->b_cont->b_rptr;
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            /* Reuse M_IOCDATA to copyin data */
            mp->b_datap->db_type = M_COPYIN;
            cqp = (struct copyreq *)mp->b_rptr;
            cqp->cq_size = stp->st_data.x_inlen;
            cqp->cq_addr = stp->st_data.x_inaddr;
            cqp->cq_flag = 0;
            stp->st_state = GETINDATA;/* next state */
            qreply(q, mp);
            break;

        case GETINDATA:/* data successfully copied in */
            /* Process input, return output */
            if ((mp->b_cont = xxx_indata(mp->b_cont)) == NULL) {
                                    /* hypothetical */
                mp->b_datap->db_type = M_IOCNAK; /* fail xxx_indata */
                mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
                iocbp->ioc_error = EIO;
                qreply(q, mp);
                break;
            }
            mp->b_datap->db_type = M_COPYOUT;
            cqp = (struct copyreq *)mp->b_rptr;
            cqp->cq_size = min(msgdsize(mp->b_cont),
                            stp->st_data.x_outlen);
            cqp->cq_addr = stp->st_data.x_outaddr;
```

Screen 6-9. Message Block Allocation

```
            cqp->cq_flag = 0;
            stp->st_state = PUTOUTDATA;/* next state */
            qreply(q, mp);
            break;

        case PUTOUTDATA:  /* data successfully copied out, ack ioctl */
            freemsg(csp->cp_private);/* state structure */
            mp->b_datap->db_type = M_IOCACK;
            mp->b_wtpr = mp->b_rptr + sizeof(struct iocblk);
            iocbp->ioc_error = 0;/* may have been overwritten */
            iocbp->ioc_count = 0;/* may have been overwritten */
            iocbp->ioc_rval = 0;/* may have been overwritten */
            qreply(q, mp);
            break;

        default:     /* invalid state: can't happen */
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            mp->b_datap->db_type = M_IOCNAK;
            mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
            iocbp->ioc_error = EINVAL;
            qreply(q, mp);
            ASSERT (0);/* panic if debugging mode */
            break;
        }  /* switch (stp->st_state) */
        break;
    default:  /* M_IOCDATA not for us */
        /* if module, pass message on */
        /* if driver, free message */
        break;
    }  /* switch (csp->cp_cmd) */
}
```

At `case GETSTRUCT`, the user *xxx*`data` structure is copied into the module's `state` structure (pointed at by `cp_private` in the message) and the `M_IOCDATA` message is reused to create a second `M_COPYIN` message to read in the user data. At `case GETINDATA`, the input user data is processed by the ***`xxx`*****`_indata`** routine (not supplied in the example), which frees the linked `M_DATA` block and returns the output data message block. The `M_IOCDATA` message is reused to create an `M_COPYOUT` message to write the user data. At `case PUTOUTDATA`, the message block containing the state structure is freed and an acknowledgment is sent upstream.

Care must be taken at the “can't happen” default case since the message block containing the state structure (`cp_private`) is not returned to the pool because it might not be valid. This might result in a lost block. The **`ASSERT`** helps find errors in the module if a “can't happen” condition occurs.

I_LIST ioctl

The **`ioctl I_LIST`** supports the **`strconf`** and **`strchg`** commands that are used to query or change the configuration of a Stream. Only the superuser or an owner of a STREAMS device may alter the configuration of that Stream. See **`strchg(1)`** for more information.

The **`strchg`** command does the following:

- Pushes one or more modules on the Stream
- Pops the topmost module off the Stream

- Pops all the modules off the Stream
- Pops all modules up to but not including a specified module

The **strconf** command does the following:

- Indicates if the specified module is present on the Stream
- Prints the topmost module of the Stream
- Prints a list of all modules and topmost driver on the Stream

If the Stream contains a multiplexing driver, the **strchg** and **strconf** commands will not recognize any modules below that driver.

The **ioctl I_LIST** performs two functions. When the third argument of the **ioctl** call is set to `NULL`, the return value of the call shows the number of modules, including the driver, present on the Stream. For example, if there are two modules above the driver, 3 is returned. On failure, `errno` may be set to a value specified in **streamio(7)**. The second function of the **I_LIST ioctl** is to copy the module names found on the Stream to the user-supplied buffer. The address of the buffer in user space and the size of the buffer are passed to the **ioctl** through a structure `str_list`, which is defined in Screen 6-10:

```
struct  str_mlist {
  char l_name[FMNAMESZ+1];       /* space for holding a module name */
};
struct str_list {
   int sl_nmods;         /* # of modules for which space is allocated */
   struct str_mlist  *sl_modlist;/* address of buffer for names */
};
```

Screen 6-10. str_list Structure

where `sl_nmods` is the number of modules in the `sl_modlist` array that the user has allocated. Each element in the array must be at least `FMNAMESZ`+1 bytes long. `FMNAMESZ` is defined by **sys/conf.h**.

The user can find out how much space to allocate by first invoking the **ioctl I_LIST** with `arg` set to `NULL`. The **I_LIST** call with `arg` pointing to the `str_list` structure returns, in the `sl_nmods` member, the number of entries that have been filled into the `sl_modlist` array. Note that the number of entries includes the driver. If there is not enough space in the `sl_modlist` array (see note) or `sl_nmods` is less than 1, the **I_LIST** call fails and `errno` is set to `EINVAL`. If *arg* or the `sl_modlist` array points outside the allocated address space, `EFAULT` is returned.

NOTE

It is possible, but unlikely, that another module was pushed on the Stream after the user invoked the **I_LIST ioctl** with the NULL argument and before the **I_LIST ioctl** with the structure argument was invoked.

Flush Handling

All modules and drivers are expected to handle `M_FLUSH` messages. An `M_FLUSH` message can originate at the Stream head or from a module or a driver. The first byte of the `M_FLUSH` message is an option flag that can have the following values:

`FLUSHR`	Flush read queue
`FLUSHW`	Flush write queue
`FLUSHRW`	Flush both, read and write, queues
`FLUSHBAND`	Flush a specified priority band only

Screen 6-11 shows line discipline module flush handling:

```
static int
ld_put(queue_t *q, mblk_t *mp)
{

    int qflag;
    pl_t pl;

    switch (mp->b_datap->db_type) {

        default:
            /*
             * queue everything except flush
             */
            putq(q, mp);
            return;

        case M_FLUSH:
            pl = freezestr(q);
            (void)strqget(q, QFLAG, 0, &qflag);/* get q_flag */
            unfreezestr(q, pl);

            if (*mp->b_rptr & FLUSHW)/* flush write queue */
                flushq(qflag & QREADR ? WR(q) : q, FLUSHDATA);

            if (*mp->b_rptr & FLUSHR)/* flush read queue */
                flushq(qflag & QREADR ? q : RD(q), FLUSHDATA);

            putnext(q, mp);
            return;
    }
}
```

Screen 6-11. Line Discipline Flush Handling

The Stream head turns around the `M_FLUSH` message if `FLUSHW` is set (`FLUSHR` will be cleared).

A driver turns around `M_FLUSH` if `FLUSHR` is set (should mask off `FLUSHW`). The Stream head turns around the `M_FLUSH` message if `FLUSHW` is set (`FLUSHR` will be cleared).

A driver turns around `M_FLUSH` if `FLUSHR` is set (should mask off `FLUSHW`).

Screen 6-12 example shows the line discipline module flushing because of break:

```
static int
ld_put(queue_t *q, mblk_t *mp)
{

    int qflag;
    pl_t pl;

    switch (mp->b_datap->db_type) {

        default:
            /*
             * queue everything except flush, break
             */
            putq(q, mp);
            return;

        case M_FLUSH:
            pl = freezestr(q);
            (void)strqget(q, QFLAG, 0, &qflag);/* get q_flag */
            unfreezestr(q, pl);

            if (*mp->b_rptr & FLUSHW)/* flush write queue */
                flushq(qflag & QREADR ? WR(q) : q, FLUSHDATA);

            if (*mp->b_rptr & FLUSHR)/* flush read queue */
                flushq(qflag & QREADR ? q : RD(q), FLUSHDATA);

            putnext(q, mp);
            return;

        case M_BREAK:
            pl=freezestr(q);
            (void)strqget(q, QFLAG, 0, &qflag);/* get q_flag */
            unfreezestr(q,pl);
            /*
             * read side only;
             * doesn't make sense for write side
             */
            if (qflag & QREADR) {
                putnextctl1(q, M_PCSIG, SIGINT);
                putnextctl1(q, M_FLUSH, FLUSHW);
                putnextctl1(WR(q), M_FLUSH, FLUSHR);
            } else
                freemsg(mp);
            return;
    }
}
```

Screen 6-12. Line Discipline Break Flushing

The next two figures further show flushing the entire Stream due to a line break. Figure 6-1shows the flushing of the write-side of a Stream, and Figure 6-2 shows the flushing of the read-side of a Stream. The dotted boxes depict flushed queues.

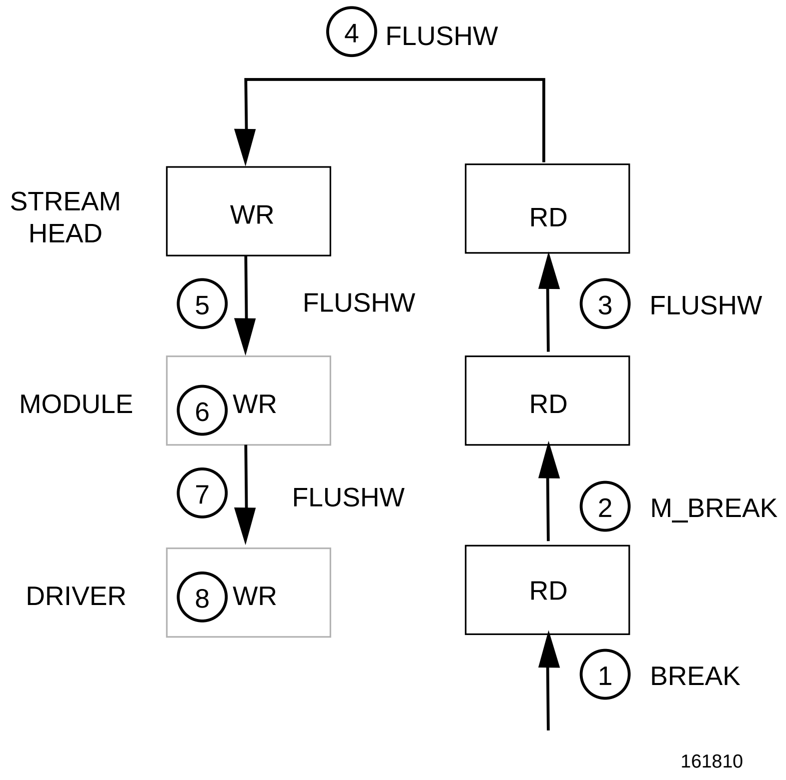


Figure 6-1. Flushing the Write-Side of a Stream

In Figure 6-1, the following is taking place:

1. A break is detected by a driver.
2. The driver generates an `M_BREAK` message and sends it upstream.
3. The module translates the `M_BREAK` into an `M_FLUSH` message with `FLUSHW` set and sends it upstream.
4. The Stream head does not flush the write queue (no messages are ever queued there).
5. The Stream head turns the message around (sends it down the write-side).
6. The module flushes its write queue.
7. The message is passed downstream.
8. The driver flushes its write queue and frees the message.

Figure 6-2 shows flushing the read-side of a Stream. The dotted boxes depict flushed queues.

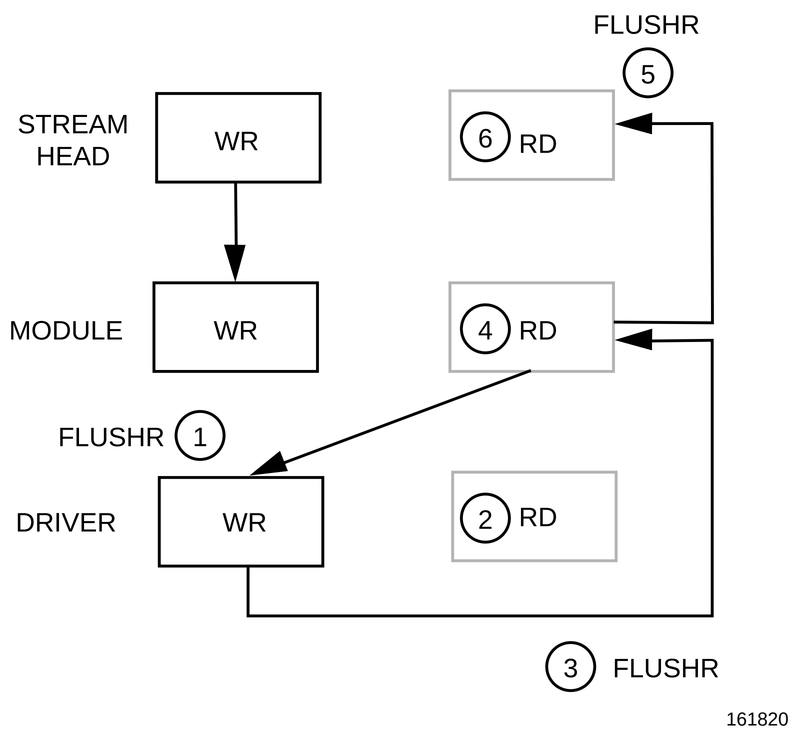


Figure 6-2. Flushing the Read-Side of a Stream

The events taking place in Figure 6-2 are as follows:

1. After generating the first `M_FLUSH` message, the module generates an `M_FLUSH` with `FLUSHR` set and sends it downstream.
2. The driver flushes its read queue.
3. The driver turns the message around (sends it up the read-side).
4. The module flushes its read queue.
5. The message is passed upstream.
6. The Stream head flushes the read queue and frees the message.

The **`flushband`** routine provides the module and driver with the capability to flush messages associated with a given priority band. See the *Device Driver Reference.*

A user can flush a particular band of messages by issuing:

`ioctl``(`*fd*`,` **`I_FLUSHBAND`**`,` *bandp*`);`

where *bandp* is a pointer to a structure `bandinfo` that has a format:

```
struct bandinfo {
        uchar_tbi_pri;
        intbi_flag;
};
```

The `bi_flag` field may be one of `FLUSHR`, `FLUSHW`, or `FLUSHRW`.

Screen 6-13 shows flushing according to the priority band:

```
queue_t  *rdq;/* read queue */
queue_t  *wrq;/* write queue */

case M_FLUSH:
    if (*bp->b_rptr & FLUSHBAND)  {
        if (*bp->b_rptr & FLUSHW)
            flushband(wrq, FLUSHDATA, *(bp->b_rptr + 1));
        if (*bp->b_rptr & FLUSHR)
            flushband(rdq, FLUSHDATA, *(bp->b_rptr + 1));
    } else {
        if (*bp->b_rptr & FLUSHW)
            flushq(wrq, FLUSHDATA);
        if (*bp->b_rptr & FLUSHR)
            flushq(rdq, FLUSHDATA);
    }
    /*
     * modules pass the message on;
     * drivers shut off FLUSHW and loop the message
     * up the read-side if FLUSHR is set; otherwise,
     * drivers free the message.
     */
    break;
```

Screen 6-13. Priority Band Flush Handling

Note that modules and drivers are not required to treat messages as flowing in separate bands. Modules and drivers can view the queue having only two bands of flow, normal and high priority. However, the latter alternative flushes the entire queue whenever an `M_FLUSH` message is received.

One use of the field `b_flag` of the `msgb` structure is provided to give the Stream head a way to stop `M_FLUSH` messages from being reflected forever when the Stream is being used as a pipe. When the Stream head receives an `M_FLUSH` message, it sets the `MSGNOLOOP` flag in the `b_flag` field before reflecting the message down the write-side of the Stream. If the Stream head receives an `M_FLUSH` message with this flag set, the message is freed rather than reflected.

Driver-Kernel Interface

The *Driver-Kernel Interface* is an interface between the PowerMAX OS system kernel and drivers. These drivers are block interface drivers, character interface drivers, and drivers and modules supporting a STREAMS interface. Each driver type supports an interface from the kernel to the driver. This kernel-to-driver interface consists of a set of driver-defined functions that are called by the kernel. These functions are the entry points into the driver.

One benefit of defining the DKI is increased portability of driver source code between various UNIX System V implementations. Another benefit is a gain in modularity that results in extending the potential for changes in the kernel without breaking driver code.

The interaction between a driver and the kernel can be described as occurring along two paths. See Figure 6-3.

One path includes those functions in the driver that are called by the kernel. These are entry points into the driver. The other path consists of the functions in the kernel that are called by the driver. Along both paths, information is exchanged between the kernel and drivers in the form of data structures. The DKI identifies these structures and specifies a set of contents for each.

The DKI defines data structure constraints (some fields are read/write, some are read-only, and some are neither readable nor writable). Be careful when you use DKI data structures; you must make sure that your code is portable, and that you do not corrupt the system. See the *Device Driver Reference* for more specific information.

NOTE

This release of the system does not support code that does not conform to the DDI/DKI.

The DKI also defines the common set of entry points expected to be supported in each driver type and their calling and return syntaxes. For each driver type, the DKI lists a set of kernel utility functions that can be called by that driver and also specifies their calling and return syntaxes.

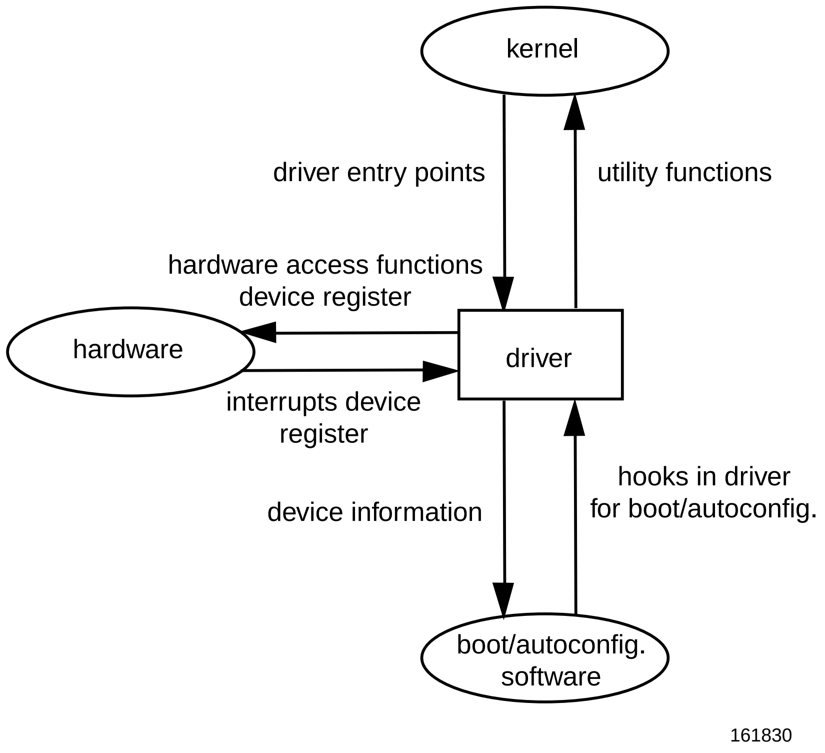

kernel
driver entry points
utility functions
hardware access functions device register
hardware
driver
interrupts device register
device information
hooks in driver for boot/autoconfig.
boot/autoconfig. software
161830

Figure 6-3. Interfaces Affecting Drivers

The set of STREAMS utilities available to drivers is listed in the *Device Driver Reference.* No system-defined macros that manipulate global kernel data or introduce structure size dependencies are permitted in these utilities. Therefore, some utilities that have been implemented as macros in the prior UNIX system releases are implemented as functions in PowerMAX OS. This does not preclude the existence of both macro and function versions of these utilities. Driver source code must include a header file that picks up function declarations while the core operating system source includes a header file that defines the macros. With the DKI interface, the following STREAMS utilities are implemented as C programming language functions: **datamsg**, **OTHERQ**, **putnext**, **RD**, **splstr**, and **WR**. See "Header Files" for more information.

Replacing macros such as **RD** with function equivalents in the driver source code allows driver objects to be insulated from changes in the data structures and their size, further increasing the useful lifetime of driver source code and objects.

The driver is insulated from implementation-specific details of multiprocessor STREAMS synchronization.

The DKI interface defines an interface suitable for drivers and there is no need for drivers to access global kernel data structures directly. The kernel functions **drv_getparm** and **drv_setparm** are provided for reading and writing information in these structures. This restriction has an important consequence. Because drivers are not permitted to access global kernel data structures directly, changes in the contents/offsets of information within these structures will not break objects. The **drv_getparm** and **drv_setparm** functions are described in more detail in the *Device Driver Reference.*

Device Driver and Driver-Kernel Interface

The DDI is an interface that facilitates driver portability across different UNIX system versions. The DKI is an interface that also facilitates driver source code portability across implementations of PowerMAX OS on all machines. DKI driver code, however, has to be recompiled on the machine on which it is to run.

The most important distinction between the DDI and the Driver-Kernel Interface lies in scope. The DDI addresses complete interfaces for block, character, and STREAMS interface drivers and modules. The DKI defines only driver interfaces with the kernel with the addition of the kernel interface for file system type (FST) modules. The DKI interface does not specify the system initialization driver interface (that is, **`init`** and **`start`** driver routines) nor hardware related interfaces.

NOTE

The “complete interface” refers to hardware- and boot/auto-configuration-related driver interface in addition to the interface with the kernel.

STREAMS Interface

The entry points from the kernel into STREAMS drivers and modules are through the **`qinit`** structures pointed to by the `streamtab` structure, *prefix*`info.` See the *Device Driver Reference.*

STREAMS drivers may need to define additional entry points to support the interface with boot/autoconfiguration software and the hardware (for example, an interrupt handler).

If the STREAMS module has prefix `mod`, then the declaration is of the form:

```
static int modrput(queue_t *, mblk_t *);
static int modrsrv(queue_t *);
static int modopen(queue_t *, dev_t *, int, int, cred_t *);
static int modclose(queue_t *, int, cred_t *);

static int modwput(queue_t *, register mblk_t *);
static int modwsrv(queue_t *);

static struct mod_minfo = {}
static struct qinit rdinit =
    {modrput, modrsrv, modopen, modclose, NULL, & m_info, NULL};

static struct qinit wrinit =
    {modwput, modwsrv, NULL, NULL, NULL, & m_info, NULL};

struct streamtab modinfo = { &rdinit, &wrinit, NULL, NULL };

int moddevflag = D_MT;
```

Screen 6-14. mod Declaration Form

where

- **`modrput`** is the module's read queue **`put`** procedure.
- **`modrsrv`** is the module's read queue **`service`** procedure.
- **`modopen`** is the **`open`** routine for the module.
- **`modclose`** is the **`close`** routine for the module.
- **`modwput`** is the **`put`** procedure for the module's write queue.
- **`modwsrv`** is the **`service`** procedure for the module's write queue.

Each `qinit` structure can point to four entry points. (An additional function pointer has been reserved for future use and must not be used by drivers or modules.) These four function pointer fields in the `qinit` structure are `qi_putp`, `qi_srvp`, `qi_qopen`, and `qi_close`.

The utility functions that can be called by STREAMS drivers and modules are listed in the *Device Driver Reference.* They must follow the call and return syntaxes specified in the manual. Manual pages relating to the DDI/DKI are provided in the *Device Driver Reference.*

Configuring the System for STREAMS Drivers and Modules

To configure the system to use a STREAMS software driver or module, you must edit a number of configuration files. The names of the files vary on different systems; see either the **`master(4)`** and **`system(4)`** manual pages or the **`mdevice(4)`** and **`sdevice(4)`** manual pages for descriptions of the configuration files for your system.

This section summarizes guidelines common to the design of STREAMS modules and drivers. Additional rules about modules and drivers can be found in “STREAMS Modules” and “STREAMS Drivers.”

Modules and Drivers

1. Modules and drivers cannot access information in the `u_area` of a process. Modules and drivers are not associated with any process, and therefore have no concept of process or user context, except during open and close routines (see the section titled “Rules for Open/Close Routines” later in This section).

   To configure the system to use a STREAMS software driver or module, you must edit a number of configuration files. The names of the files vary on different systems; see either the **`master(4)`** and **`system(4)`** manual pages or the **`mdevice(4)`** and **`sdevice(4)`** manual pages for descriptions of the configuration files for your system.

2. Every module and driver must process an `M_FLUSH` message according to the value of the argument passed in the message.

3. A module or a driver should not change the contents of a data block whose reference count is greater than 1 because other modules/drivers that have references to the block may not want the data changed. To avoid problems, data should be copied to a new block and then changed in the new one. See the *Device Driver Reference.*

4. Modules and drivers should manipulate queues and manage buffers only with the routines provided for that purpose, in conformance with DDI/DKI. See the *Device Driver Reference.*

5. Modules and drivers should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment.

6. Care must be taken when modules are mixed and matched, because one module may place different semantics on the priority bands than another module. The specific use of each band by a module should be included in the service interface specification.

   When designing modules and drivers that make use of priority bands one should keep in mind that priority bands merely provide a way to impose an ordering of messages on a queue. The priority band is not used to determine the service primitive. Instead, the service interface should rely on the data contained in the message to determine the service primitive.

Rules for Open/Close Routines

- **`open`** and **`close`** routines may use blocking primitives as defined in the DDI.
- The **`open`** routine should return zero on success or an error number on failure. If the **`open`** routine is called with the `CLONEOPEN` flag, the device number should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major/minor).
- If a module or a driver wants to allocate a controlling terminal, it should send an `M_SETOPTS` message to the Stream head with the `SO_ISTTY` flag set. Otherwise signaling will not work on the Stream.
- **`open`** and **`close`** routines have user context and can access some fields in the `u_area` using the **`drv_getparm`** and **`drv_setparm`** functions.

  A multithreaded driver or module must call **`qprocson`** to enable its put and service procedures and **`qprocsoff`** to disable them.

NOTE

The DKI interface provides the **`drv_getparm`** and **`drv_setparm`** functions to read/write kernel parameters, so the driver/module should not access them directly.

Rules for ioctls

- Do not change the `ioc_id, ioc_uid, ioc_gid,` or `ioc_cmd` fields in an `M_IOCTL` message.
- The above rule also applies to fields in an `M_IOCDATA`, `M_COPYIN`, and `M_COPYOUT` message. Field names are different; See the *Device Driver Reference.*
- Always validate `ioc_count` to see whether the **`ioctl`** is the transparent or **`I_STR`** form.

Rules for Put and Service Procedures

To ensure proper data flow between modules and drivers, the following rules should be observed in **`put`** and **`service`** procedures:

- **`Put`** and **`service`** procedures must not sleep.
- Return codes can be sent with STREAMS messages `M_IOCACK`, `M_IOCNAK`, and `M_ERROR`.
- Protect data structures common to **`put`** and **`service`** procedures by using **`splstr`**.

  Note that multithreaded drivers must protect all global driver data with DDI/DKI-defined locks or synchronization utility functions.
- **`Put`** and **`service`** procedures cannot access the information in the `u_area` of a process.
- Messages should be handled consistently. Any given message type should be handled completely by the **`put`** procedure, or deferred to the **`service`** procedure.

`Put` and **`service`** procedures must protect against race conditions using DDI/DKI locks. The basic model for **`put`** and **`service`** concurrency for a multithreaded driver is as follows: Only one instance of the **`service`** procedure for a specific queue may run at a time; this ensures FIFO ordering of messages is preserved. Multiple instances of the **`put`** procedure may run concurrently, and the **`put`** and **`service`** routines may run concurrently with each other. Strict adherence to the DDI/DKI rules governing system data structure access and use of DDI/DKI STREAMS utilities (for example, **`getq`**, **`strqget`**, **`putnext`**, and so forth) protects underlying STREAMS subsystem races. However, the driver writer must take care to protect driver-private data structures from potential race conditions because of **`put`** and **`service`** procedure concurrency. To protect against deadlock, the processor priority associated with a given driver lock must be high enough to prevent all interrupts that may need to acquire that lock.

NOTE

References to drivers apply to modules as well.

Put Procedures

- Generally, each queue defines a **`put`** procedure in its `qinit` structure for passing messages between modules.

  In some instances, drivers do not need put procedures; for example, messages are only passed upstream by the driver's interrupt routine, and therefore a read-side put procedure is not needed.

- A **`put`** procedure must use the **`putq`** utility to enqueue a message on its own queue. This is necessary to ensure that the various fields of the `queue` structure are maintained consistently. See the *Device Driver Reference.*

- When passing messages to a neighboring module, a module may not call **`putq`** directly, but must call its neighbor module's **`put`** procedure using the appropriate DDI/DKI STREAMS utility. See **`putnext`** in the *Device Driver Reference.*

  However, the `q_qinfo` structure that points to a module's **`put`** procedure may point to **`putq`** (that is, **`putq`** is used as the **`put`** procedure for that module). When a module calls a neighbor module's **`put`** procedure that is defined in this way, it will be calling **`putq`** indirectly. If any module uses **`putq`** as its **`put`** procedure in this way, the module must define a **`service`** procedure. Otherwise, no messages will ever be processed by the next module. Also, because **`putq`** does not process `M_FLUSH` messages, any module that uses **`putq`** as its **`put`** procedure must define a **`service`** procedure to process `M_FLUSH` messages.

- The **`put`** procedure of a queue with no **`service`** call its neighbor module's **`put`** procedure using the appropriate DDI/DKI STREAMS utility. See **`putnext`** in the *Device Driver Reference.*

- The **`put`** procedure of a queue with no **`service`** procedure must call the **`put`** procedure of the next queue using `putnext` if a message is to be passed to that queue.

- Processing many function calls with the **`put`** procedure could lead to interrupt stack overflow. In that case, switch to **`service`** procedure processing whenever appropriate to switch to a different stack.

- Although most drivers do not have a read-side **`put`** procedure, those that do must be called (for example, from the interrupt handler) with the multiprocessor DDI/DKI function **`put`**.

Service Procedures

1. If flow control is desired, a **`service`** procedure is required.

   The **`service`** procedure should use the **`canputnext`** or **`bcanputnext`** routines before doing **`putnext`** to honor flow control.

2. The **`service`** procedure must use **`getq`** to remove a message from its message queue, so that the flow control mechanism is maintained.

3. The **`service`** procedure should process all messages on its queue. The only exception is if the queue ahead is blocked (that is, **`canputnext`** fails) or some other failure like buffer allocation failure. Adherence to this rule is the only guarantee that STREAMS will enable (schedule for execu-

tion) the **`service`** procedure when necessary, and that the flow control mechanism will not fail.

If a **`service`** procedure exits for other reasons, it must take explicit steps to assure it will be re-enabled.

4. The **`service`** procedure should not put a high-priority message back on the queue, because of the possibility of getting into an infinite loop.
5. The **`service`** procedure must follow the steps below for each message that it processes. STREAMS flow control relies on strict adherence to these steps.

   a. Remove the next message from the queue using **`getq`**. It is possible that the **`service`** procedure could be called when no messages exist on the queue, so the **`service`** procedure should never assume that there is a message on its queue. If there is no message, RETURN.

   b. If all the following conditions are met:

      - Failure of functions **`canputnext`** or **`bcanputnext`**.
      - The message type is not a high priority type.
      - The message is to be put on the next queue.

      Continue to Step c. Otherwise, continue at Step d.

   c. The message must be replaced on the head of the queue from which it was removed using **`putbq`**. See the *Device Driver Reference.* Following this, the **`service`** procedure is exited. The **`service`** procedure should not be re-enabled at this point. It will be automatically back-enabled by flow control.

   d. If all the conditions of Step b are not met, the message should not be returned to the queue. It should be processed as necessary. Then, return to Step a.

Data Structures

Only the contents of `q_ptr`, `q_minpsz`, `q_maxpsz, q_hiwat`, and `q_lowat` in the `queue` structure may be altered. `q_minpsz`, `q_maxpsz`, `q_hiwat`, and `q_lowat` are set when the module or driver is opened, but they may be modified later only by using the DDI/DKI utility **`strqset`**.

Drivers and modules are allowed to change the `qb_hiwat` and `qb_lowat` fields of the `qband` structure. They may only read the `qb_count`, `qb_first`, `qb_last`, and `qb_flag` fields.

The routines **`strqget`** and **`strqset`** must be used to get and set the fields associated with the queue. They insulate modules and drivers from changes in the `queue` structure and from multiprocessor STREAMS implementation details, and also enforce the previous rules.

Dynamic Allocation of STREAMS Data Structures

Before PowerMAX OS, STREAMS data structures were statically configured to support a fixed number of Streams, read and write queues, message and data blocks, link block data structures, and Stream event cells. The only way to change this configuration was to reconfigure and reboot the system. Resources were also wasted because data structures were allocated but not necessarily needed.

With Release 4, the STREAMS mechanism has been enhanced to dynamically allocate the following STREAMS data structures: `stdata`, `queub_last`, and `qb_flag` fields. STREAMS allocates memory to cover these structures as needed.

Dynamic data structure allocation has the advantage of the kernel being initially smaller than a system with static configuration. The performance of the system may also improve because of better memory use and added flexibility. However, **`allocb`**, **`bufcall`**, and **`freeb`**, the routines that manage these data structures, may be slower at times because of extra overhead needed for dynamic allocation.

Header Files

The following header files are generally required in modules and drivers:

`types.h`	Contains type definitions used in the STREAMS header files.
`stream.h`	Contains required structure and constant definitions.
`stropts.h`	Primarily for users, but contains definitions of the arguments to the `M_FLUSH` message type also required by modules.
`ddi.h`	Contains definitions and declarations needed by drivers to use functions for the UNIX System V DDI or DKI. This header file should be the last header file included in the driver source code (after all `#include` statements).

One or more of the header files described next may also be included. No standard UNIX system header files should be included except as described in the following section. The intent is to prevent attempts to access data that cannot or should not be accessed.

`errno.h`	Defines various system error conditions, and is needed if errors are to be returned upstream to the user.
`sysmacros.h`	Contains miscellaneous system macro definitions (subject to DDI/DKI restrictions).
`param.h`	Defines various system parameters.
`signal.h`	Defines system signal values, and should be used if signals are to be processed or sent upstream.
`file.h`	Defines file open flags, and is needed if `O_NDELAY` (or `O_NONBLOCK`) is interpreted.

7
STREAMS Modules

7
STREAMS Modules

Introduction

A STREAMS module is a pair of queues and a defined set of kernel-level routines and data structures used to process data, status, and control information. A Stream may have zero or more modules. User processes push (insert) modules on a Stream using the **`I_PUSH`** **`ioctl`** and pop (remove) them using the **`I_POP`** **`ioctl`**. Pushing and popping of modules happens in a Last-In-First-Out (LIFO) fashion. Modules manipulate messages as they flow through the Stream.

Routines

STREAMS module routines (such as **`open`**, **`close`**, **`put`**, and **`service`**) have already been described in the previous sections. This section shows some examples and further describes attributes common to module **`put`** and **`service`** routines.

A module's **`put`** routine is called by the preceding module, driver, or Stream head and before the corresponding **`service`** routine. The **`put`** routine should do any processing that needs to be done immediately (for example, processing of high-priority messages). Any processing that can be deferred should be left for the corresponding **`service`** routine.

The **`service`** routine implements flow control, handles de-packetization of messages, performs deferred processing, and handles resource allocation. Once the **`service`** routine is enabled, it may be started but not necessarily completed before running user-level code.

The **`put`** and **`service`** routines must not call **`sleep`** and cannot access the `u_area` area, because they are executed asynchronously with respect to any process.

Screen 7-1 shows a STREAMS module read-side **`put`** routine:

```
static int modrput(queue_t *q, mblk_t *mp)
{
    struct mod_prv *modptr;

    modptr = (struct mod_prv *) q->q_ptr;  /* for state information */

    if (pcmsg(mp->b_datap->db_type)) { /* process priority message */
        putnext(q, mp);                /* and pass it on */
        return;
    }
    switch(mp->b_datap->db_type) {
    case M_DATA:                     /* may process message data */
        putq(q, mp);                   /* queue message for service routine */
        return;
    case M_PROTO:    /* handle protocol control message */
         .
         .
         .
    default:
        putnext(q, mp);
        return;
    }
}
```

Screen 7-1. Read Side put Procedure

The following briefly describes the code:

- A pointer to a queue defining an instance of the module and a pointer to a message are passed to the **put** routine.
- The **put** routine switches on the type of the message. For each message type, the **put** routine either enqueues the message for further processing by the module **service** routine, or passes the message to the next module in the Stream.
- High priority messages are processed immediately by the **put** routine and passed to the next module.
- Ordinary (or normal) messages are either enqueued or passed along the Stream.

Screen 7-2 shows a module write-side **put** routine:

```
static int modwput(queue_t *q, mblk_t *mp)
{
    struct mod_prv *modptr;
    modptr = (struct mod_prv *) q->q_ptr;/* for state information */
    if (pcmsg(mp->b_datap->db_type)) {/* process priority message */
        putnext(q, mp);       /* and pass it on */
        return;
    }
    switch(mp->b_datap->db_type) {
    case M_DATA:     /* may process message data */
        putq(q, mp); /* queue message for service routine */
                        /* or pass message along */
                        /* putnext(q, mp); */
        return;
    case M_PROTO:
        .
        .
        .
    case M_IOCTL:/* if command in message is recognized */
                /* process message and send back reply */
                /* else pass message downstream */
    default:
        putnext(q, mp);
        return;
    }
}
```

Screen 7-2. Write Side put Procedure

The write-side **`put`** routine, unlike the read-side, may be passed `M_IOCTL` messages. It is up to the module to recognize and process the **`ioctl`** command, or pass the message downstream if it does not recognize the command.

Screen 7-3 shows a general scenario employed by the module's **`service`** routine:

```
static int modrsrv(queue_t *q)
{
    mblk_t *mp;

    while ((mp = getq(q)) != (mblk_t *) NULL) {
        if (!pcmsg(mp->b_datap->db_type) &&
           !canputnext(q)) {    /* flow control check */
            putbq(q, mp);/* return message */
            return;
        }
                               /* process the message */
        switch(mp->b_datap->db_type) {
            .
            .
            .
            putnext(q, mp);/* pass the result */
        }
    } /* while */
}
```

Screen 7-3. Service Routine

The steps are as follows:

1. Retrieve the first message from the queue using **getq**.
2. If the message is high priority, process it immediately, and pass it along the Stream.
3. Otherwise, the **service** routine should use the **canputnext** routine to determine if the next module or driver that enqueues messages is within acceptable flow control limits. **canputnext** goes down the Stream (or up on the read-side) until it reaches a module, a driver, or the Stream head with a **service** routine. When it reaches one, it looks at the total message space currently allocated at that queue for enqueued messages. If the amount of space currently used at that queue exceeds the high-water mark, **canputnext** returns false (zero). If the next queue with a **service** routine is within acceptable flow control limits, it returns true (nonzero).
4. If **canputnext** returns false, the **service** routine should return the message to its own queue using the **putbq** routine. The **service** routine can do no further processing now, and it should return.
5. If **canputnext** returns true, the **service** routine should complete any processing of the message. This may involve retrieving more messages from the queue, (de)**-**allocating header and trailer information, and performing control function for the module.
6. When the **service** routine is finished processing the message, it may call the **putnext** routine to pass the resulting message to the next queue.
7. Above steps are repeated until there are no messages left on the queue (that is, until **getq** returns NULL) or **canputnext** returns false.

Filter Module Example

The module shown in Screen 7-4, **crmod**, is an asymmetric filter. On the write-side, newline is converted to carriage return followed by newline. On the read-side, no conversion is done. The declarations of this module are the same as those of the null module presented in the previous section:

```
/* Simple filter - converts newline -> carriage return, newline */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>

static struct module_info minfo = { 0x09, “crmod”, 0, INFPSZ, 512, 128 };

static int modopen, modrput, modwput, modwsrv, modclose;

static struct qinit rinit = {
    modrput, NULL, modopen, modclose, NULL, &minfo, NULL };

static struct qinit winit = {
    modwput, modwsrv, NULL, NULL, NULL, NULL, &minfo, NULL };

struct streamtab crmdinfo = { &rinit, &winit, NULL, NULL };

int moddevflag = D_MP;
```

Screen 7-4. Filter Module

`stropts.h` includes definitions of flush message options common to user level, modules and drivers. **`modopen`** and **`modclose`** are unchanged from the null module example shown earlier in this chapter. **`modrput`** is like **`modput`** from the null module.

Note that, in contrast to the null module example, a single `module_info` structure is shared by the read-side and write-side. The `module_info` includes the flow control high- and low-water marks (512 and 128) for the write queue. (Although the same `module_info` is used on the read queue side, the read-side has no **`service`** procedure, so flow control is not used.) The `qinit` contains the **`service`** procedure pointer.

The write-side **`put`** procedure, the beginning of the **`service`** procedure, and an example of flushing a queue are shown in Screen 7-5:

```
static int modwput(queue_t *q, register mblk_t *mp)
{
    if (pcmsg(mp->b_datap->db_type) && mp->b_datap->db_type != M_FLUSH)
        putnext(q, mp);
    else
        putq(q, mp);    /* Put it on the queue */
}

static int modwsrv(queue_t  *q)
{
    mblk_t *mp;

    while ((mp = getq(q) != NULL) {
        switch (mp->b_datap->db_type) {

            default:
                if (canputnext(q)) {
                    putnext(q, mp);
                    break;
                } else {
                    putbq(q, mp);
                return;
                    }

            case M_FLUSH:
                if (*mp->b_rptr & FLUSHW)
                     flushq(q, FLUSHDATA);
                putnext(q, mp);
                break;
```

Screen 7-5. Write Side put Procedure and Queue Flush

modwput, the write **put** procedure, switches on the message type. High-priority messages that are not type M_FLUSH are **putnext** to avoid scheduling. The others are queued for the **service** procedure. An M_FLUSH message is a request to remove messages on one or both queues. It can be processed in either the **put** or **service** procedure; it is preferable to use the **put** procedure, so that M_FLUSH is handled immediately.

modwsrv is the write **service** procedure. It takes a single argument, a pointer to the write queue. **modwsrv** processes only one high-priority message, M_FLUSH. No other high-priority messages should reach **modwsrv**.

For an M_FLUSH message, **modwsrv** checks the first data byte. If FLUSHW (defined in **stropts.h**) is set, the write queue is flushed with the **flushq** utility. See the *Device Driver Reference*. **flushq** takes two arguments, the queue pointer and a flag. The flag shows what should be flushed, data messages (FLUSHDATA) or everything (FLUSHALL). In Screen 7-6, data includes M_DATA, M_DELAY, M_PROTO, and M_PCPROTO messages. The choice of what types of messages to flush is module-specific.

If **canputnext**(*q*) returns false, ordinary messages are returned to the queue, indicating the downstream path is blocked. Screen 7-6 continues with the remaining part of **modwsrv** processing M_DATA messages:

```
            case M_DATA: {
                mblk_t *nbp = NULL;
                mblk_t *next;

                if (!canputnext(q)) {
                    putbq(q, mp);
                    return;
                }
                /* Filter data, appending to queue */
                for (; mp != NULL; mp = next) {
                    while (mp->b_rptr < mp->b_wptr) {
                        if (*mp->b_rptr == '\n´
                            if (!bappend(&nbp, '\r'))
                                goto push;
                        if (!bappend(&nbp, *mp->b_rptr))
                            goto push;
                        mp->b_rptr++;
                        continue;

                    push:
                        if (nbp)
                        putnext(q, nbp);
                        nbp = NULL;
                        if (!canputnext(q)) {
                            if (mp->b_rptr >= mp->b_wptr) {
                                next = mp->b_cont;
                                freeb(mp);
                                mp=next;
                            }
                            if (mp)
                                putbq(q, mp);
                            return;
                        }
                    } /* while */
                    next = mp->b_cont;
                    freeb(mp);
                } /* for */
                if (nbp)
                    putnext(q, nbp);
            } /* case M_DATA */
        } /* switch */
    } /* while */
}
```

Screen 7-6. M_DATA Message Processing

The differences in `M_DATA` processing between this and the example in the section titled “Message Allocation and Freeing” relate to the way the new messages are forwarded and flow controlled. To show alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with the **`putnext`** routine rather than being linked into a single, large message (as was done in the example). This alternative may not be desirable because message boundaries are altered and there is an additional overhead of handling and scheduling multiple messages.

When the filter processing is performed (following push), **`canputnext`** should check flow control after, rather than before, each new message is forwarded. This is because there is no provision to hold the new message until the queue becomes unblocked. If the downstream path is blocked, the remaining part of the original message is returned to the queue. Otherwise, processing continues.

Flow Control

To use the STREAMS flow control mechanism, modules must use **`service`** procedures, invoke **`canputnext`** before calling **`putnext`**, and use appropriate values for the high- and low-water marks.

Module flow control limits the amount of data that can be placed on a queue. It prevents depletion of buffers in the buffer pool. Flow control is advisory in nature and can be bypassed. It is managed by high- and low-water marks and regulated by `QWANTW` and `QFULL` flags. Module flow control is implemented by using the **`canputnext`**, **`getq`**, **`putq`**, **`putbq`**, **`insq`**, and **`rmvq`** routines.

During normal flow control, when a module and driver are in sync, the following steps are taken:

1. A driver sends data to a module using the **`putnext`** routine.
2. The module's **`put`** procedure queues data using **`putq`**.
3. The **`putq`** routine increments the module's `q_count` by the number of bytes in the message and enables the **`service`** procedure.
4. When STREAMS scheduling runs the **`service`** procedure, the **`service`** procedure retrieves the data by calling the **`getq`** utility.
5. **`getq`** decrements `q_count` by an appropriate value.

If the module cannot process data at the rate at which the driver is sending the data, the following steps occur:

1. The module's `q_count` goes above its high-water mark, and the `QFULL` flag is set by **`putq`**.
2. The driver's **`canputnext`** fails, and sets `QWANTW` flag in the module's queue.
3. The driver sends a command to the device to either stop input, queue the data in its own queue, or drop the data.
4. The module's `q_count` falls below its low-water mark because of **`getq`**.
5. **`getq`** finds the nearest back queue with a **`service`** procedure and enables it.
6. The scheduler runs the **`service`** procedure.

The procedure for banded data is the same, except that `qb_count` is used in place of `q_count`.

NOTE

Flow control does not prevent exceeding `q_hiwat` on a given queue. Flow control may exceed its maximum before **`canputnext`** detects `QFULL` and flow is stopped.

Screen 7-7 and Screen 7-8 show a line discipline module's flow control. Screen 7-7 is a read-side line discipline module:

```
/* read-side line discipline module flow control */

ld_read_srv(queue_t *q)
{
    mblk_t *mp;    /* original message */
    mblk_t *bp;    /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
       switch (mp->b_datap->db_type) {       /* type of message */
       case M_DATA:                     /* data message */
           if (canputnext(q)) {
             bp = read_canon(mp);
             putnext(q, bp);
           } else {
             putbq(q, mp);    /* put message back in queue */
             return;
           }
           break:

       default:
        if (pcmsg(mp->b_datap->db_type))
             putnext(q, mp);  /* high priority message */
        else {                          /* ordinary message */
             if (canputnext(q))
                 putnext(q, mp);
             else {
                 putbq(q, mp);
                 return;
             }
        }
        break;
    }
  }
}
```

Screen 7-7. Read Side Line Discipline

Screen 7-8 shows a write-side line discipline module:

```
/* write-side line discipline module flow control */

ld_write_srv(queue_t *q)
{
    mlbk_t *mp;/* original message */
    mblk_t *bp;/* canonicalized message */

    while ((mp = getq(q)) != NULL) {
       switch (mp->b_datap->db_type) {  /* type of message */
       case M_DATA:        /* data message */
        if (canputnext(q)) {
            bp = write_canon(mp);
            putnext(q, bp);
        } else {
            putbq(q, mp);
            return;
        }
            break;

       case M_IOCTL:
            ld_ioctl(q, mp);
        break:

       default:
        if (pcmsg(mp->b_datap->db_type))
            putnext(q, mp);/* high priority message */
           else {                    /* ordinary message */
            if (canputnext(q))
                putnext(q, mp);
            else {
                putbq(q, mp);
                return;
            }
        }
        break;
    }
    }
}
```

Screen 7-8. Write Side Line Discipline

Design Guidelines

Module developers should follow these guidelines:

- Message types that are not understood by the modules should be passed to the next module.
- The module that acts on an `M_IOCTL` message should send an `M_IOCACK` or `M_IOCNAK` message in response to the **`ioctl`**. If the module does not understand the **`ioctl`**, it should pass the `M_IOCTL` message to the next module.
- Modules should be designed in such a way that they do not pertain to any particular driver but can be used by all drivers.

- In general, modules should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. This makes it easier to arbitrarily push modules on top of each other in a sensible fashion. Not following this rule may limit module reusability.
- Filter modules pushed between a service user and a service provider may not alter the contents of the `M_PROTO` or `M_PCPROTO` block in messages. The contents of the data blocks may be manipulated, but the message boundaries must be preserved.
- A multithreaded module is responsible for protecting module-specific data against multiprocessor race conditions.

8
STREAMS Drivers

8
STREAMS Drivers

Introduction

A driver is software that provides an interface between the operating system and a device. The driver controls the device in response to kernel commands, and user-level programs access the device through system calls. The system calls interface with the file system and process control system, which in turn access the drivers. The driver provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller.

Unlike a module, a device driver may have an interrupt routine so that it is accessible from a hardware interrupt as well as from the Stream. A driver can have multiple Streams connected to it. Multiple connections occur when more than one minor device of the same driver is in use and for multiplexors. However, these particular differences are not recognized by the STREAMS mechanism. They are handled by developer-provided code included in the driver procedures.

This chapter describes the operation of a STREAMS driver, and discusses some of the processing typically required in drivers.

Driver Classification

In general, drivers are grouped according to the type of the device they control, the access method (the way data is transferred), and the interface between the driver and the device.

The type can be hardware or software. A hardware driver controls a physical device such as a disk. A software driver, also called a pseudo-device, controls software, which in turn may interface with a hardware device. The software driver may also support pseudo-devices that have no associated physical device.

Drivers can be character-type or block-type, but many support both access methods. In character-type transfer, data is read a character at a time or as a variable length stream of bytes, the size of which is determined by the device. In block-type access, data transfer is performed on fixed-length blocks of data. Devices that support both block- and character-type access must have a separate special device file for each access method. Character access devices can also use raw (also called unbuffered) data transfer that takes place directly between user address space and the device. Unbuffered data transfer is used mainly for administrative functions where the speed of the specific operation is more important than overall system performance.

The driver interface refers to the system structures and kernel interfaces used by the driver. For example, STREAMS is an interface.

Writing a Driver

All drivers are identified by a string of up to four characters called the prefix. The prefix is defined in the master file for the driver and is added to the name of the driver routines. For example, the **`open`** routine for the driver with the `xyz` prefix is ***`xyz`*****`open`**.

Writing a driver differs from writing other C programs in the following ways:

- A driver does not have a **`main`** routine. Driver entry points are given specific names and accessed through switch tables.
- A driver functions as a part of the kernel. Consequently, a poorly written driver can degrade system performance or corrupt the system.
- A driver cannot use system calls or the C library, because the driver functions at a lower level.
- A driver cannot use floating-point arithmetic.
- A driver cannot use archives or shared libraries, but frequently used subroutines can be put in separate files in the source code directory for the driver.

Driver code, like other system software, uses the advanced C language capabilities. These include bit-manipulation capabilities, casting of data types, and use of header files for defining and declaring global data structures.

Driver code includes a set of entry point routines:

- Initialization entry points that are accessed through `io_init` and `io_start` arrays during system initialization.
- Switch table entry points that are accessed through `bdevsw` (block-access) and `cdevsw` (character-access) switch tables when the appropriate system call is issued.
- Interrupt entry points that are accessed through the interrupt vector table when the hardware generates an interrupt.

The following lists rules of driver development:

- All drivers must have entries in the necessary configuration files. See "Configuring the System for STREAMS Drivers and Modules."
- All drivers should have `#include` system header files that define data structures used in the driver.
- Drivers may have an **`init`** and/or a **`start`** routine to initialize the driver.

  Software drivers usually have little to initialize, because there is no hardware involved. An **`init`** routine is used when a driver needs to initialize but does not need any system services. **`init`** routines are run before system services are initialized (like the kernel memory allocator, for example). When a driver needs to do initialization that requires system services, a **`start`** routine is used. The **`start`** routines are run after system services are initialized.

- Drivers have **`open`** and **`close`** routines.

- Most drivers have an interrupt handler routine.

  The driver developer is responsible for supplying an interrupt routine for the device's driver. The PowerMAX OS system provides a few interrupt handling routines for hardware interrupts, but the developer has to supply the specifics about the device.

  In general, a ***prefix*****`int`** interrupt routine should be written for any device that does not send separate transmit and receive interrupts. TTY devices that request separate transmit and receive interrupts can have two separate interrupt routines associated with them; ***prefix*****`xint`** to transmit an interrupt, and ***prefix*****`rint`** to receive an interrupt.

- Most drivers have `static` subordinate driver routines to provide the functionality for the specific device. The names of these routines should include the driver *prefix*, although this is not required since the routine is declared as `static`.

- A bootable object file and special device files are also needed for a driver to be fully functional.

Major and Minor Device Numbers

A device appears to the PowerMAX OS system as a special device file. The system accesses a device by opening, reading, writing, and closing the device's special device file.

The system identifies and accesses the special device file using the file's major and minor device numbers. The major number identifies a driver for a controller. The minor number identifies a specific device.

Major numbers are assigned by the installation and configuration software. Minor numbers are designated by the driver developer.

Minor numbers are determined differently for different types of devices. Typically, minor numbers are an encoding of information needed by the controller board.

Major and minor numbers can be external or internal.

- External major numbers are those visible to the user.

- Internal major numbers serve as an index into the `cdevsw` and `bdevsw` switch tables. These are assigned by the configuration process when drivers are loaded and they may change every time a full configuration boot is done.

  One driver may control several devices, but each device will have its own external major number and all those external major numbers are mapped to one internal major number for the driver.

- External minor numbers are controlled by a driver developer, although there are conventions enforced for some types of devices by some utilities. For example, a tape drive may interface with a hardware controller (device) to which several tape drives (subdevices) are attached. All tape drives attached to one controller will have the same external major number, but each drive will have a different external minor number.

- Internal minor numbers are used with hardware drivers to identify the logical controller that is being addressed. Because drivers that control multiple devices (controllers) usually require a data structure for each configured device, drivers address the per-controller data structure by the internal minor number rather than the external major number.

Logical controller numbers are assigned sequentially by the central controller firmware at self-configuration time.

The internal minor number for all software drivers is 0.

The switch tables will have only as many entries as required to support the drivers installed on the system. Switch table entry points are activated by system calls that reference a special device file that supplies the external major number and instructions on whether to use `bdevsw` or `cdevsw`. The routines **`getmajor`** and **`getminor`** return an internal major and minor number for the device. The routines **`getemajor`** and **`geteminor`** return an external major and minor number for the device.

STREAMS Drivers

At the interface to hardware devices, character I/O drivers have interrupt entry points; at the system interface, those same drivers generally have direct entry points (routines) to process **`open`**, **`close`**, **`read`**, **`write`**, **`poll`**, and **`ioctl`** system calls.

STREAMS device drivers have interrupt entry points at the hardware device interface and have direct entry points only for the **`open`** and **`close`** system calls. These entry points are accessed by STREAMS, and the call formats differ from traditional character device drivers. (STREAMS drivers are character drivers, too. We call the non-STREAMS character drivers traditional character drivers or non-STREAMS character drivers.) The **`put`** procedure is a driver's third entry point, but it is a message (not system) interface. The Stream head translates **`write`** and **`ioctl`** calls into messages and sends them downstream to be processed by the driver's write queue **`put`** procedure. **`read`** is seen directly only by the Stream head, which contains the functions required to process system calls. A driver does not know about system interfaces other than **`open`** and **`close`**, but it can detect the absence of a **`read`** indirectly if flow control propagates from the Stream head to the driver and affects the driver's ability to send messages upstream.

For input processing, when the driver is ready to send data or other information to a user process, it does not wake up the process. It prepares a message and sends it to the read queue of the appropriate (minor device) Stream. The driver's **`open`** routine generally stores the queue address corresponding to this Stream.

For output processing, the driver receives messages in place of a **`write`** call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Later output interrupts can remove messages from this queue.

When sending data to the device, the driver needs to handle the special cases that affect hardware access to the memory. For example, drivers that perform physical Direct Memory Access (DMA) to or from STREAMS message buffers should be aware that a STREAMS message buffer can cross page boundaries. This will happen if the buffer size is greater than the page size of the machine. Buffers smaller than the page size are usually allocated such that they will not cross a page boundary, but if the message was allocated via **`esballoc`**, the buffer could be positioned in an arbitrary location in memory.

Drivers using physical DMA should therefore transfer only those ranges of memory that are physically contiguous. The driver should check each message buffer to see if the data crosses a page boundary, and transfer separately each range of data that resides in a different page. If the hardware supports scatter-gather DMA, then the driver should generate a new base-length pair for each page.

Figure 8-1 shows multiple Streams (corresponding to minor devices) to a common driver. There are two distinct Streams opened from the same major device. Consequently, they have the same `streamtab` and the same driver procedures.

The configuration mechanism distinguishes between STREAMS devices and traditional character devices, because system calls to STREAMS drivers are processed by STREAMS routines, not by the PowerMAX OS system driver routines. In the `cdevsw` file, the field `d_str` provides this distinction.

Multiple instantiations (minor devices) of the same driver are handled during the initial open for each device. Typically, the `queue` address is stored in a driver-private structure array indexed by the minor device number. This is for use by the interrupt routine that needs to translate from device number to a particular Stream. The `q_ptr` of the `queue` points to the private data structure entry. When the messages are received by the queue, the calls to the driver **`put`** and **`service`** procedures pass the address of the `queue`, allowing the procedures to determine the associated device.

A driver is at the end of a Stream. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

STREAMS guarantees that only one **`open`** or **`close`** routine will be active at any time for any given major/minor pair.

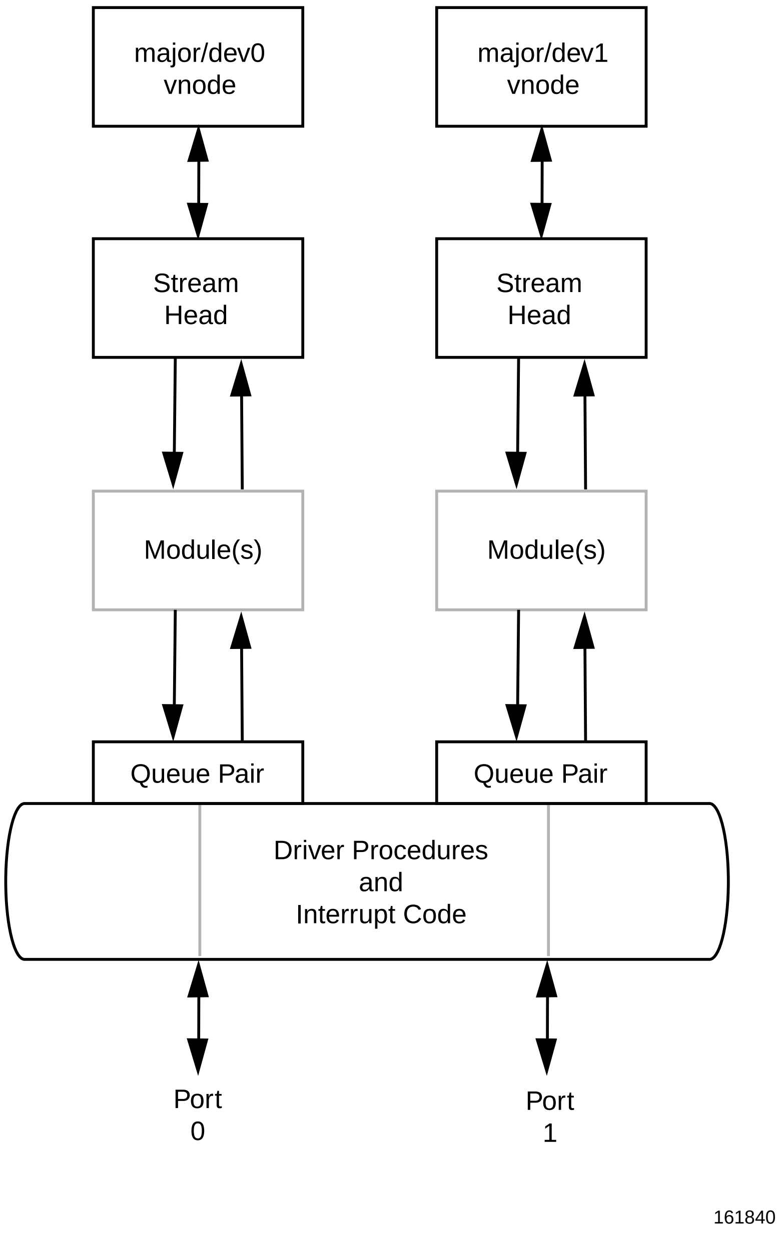


Figure 8-1. Device Driver Streams

Printer Driver Example

Screen 8-2 through Screen 8-6 show how a simple interrupt-per-character line printer driver could be written. The driver is unidirectional and has no read-side processing. It shows some differences between module and driver programming, including the following:

Open handling	A driver is passed a device number or is asked to select one.

Flush handling	A driver must loop `M_FLUSH` messages back upstream.
`ioctl` handling	A driver must send a negative acknowledgment for **`ioctl`** messages it does not understand. This is discussed under “Module and Driver ioctls.”

Declarations

The driver declarations are shown in Screen 8-1. For more information, see “Module and Driver Declarations.”

```
/* Simple line printer driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>

static struct module_info minfo = {
    0xaabb, “lp”, 0, INFPSZ, 150, 50 };

static int lpopen(queue_t *, dev_t *, int, int, cred_t *);
static int lpclose(queue_t *, int, cred_t *);
static int lpwput(queue_t *, mblk_t *);

static struct qinit rinit = {
    NULL, NULL, lpopen, lpclose, NULL, &minfo, NULL };

static struct qinit winit = {
    lpwput, NULL, NULL, NULL, NULL, &minfo, NULL };

struct streamtab lpinfo = { &rinit, &winit, NULL, NULL };

#define SET_OPTIONS(('l'<<8)|1)/* should be in a .h file */

lkinfo_t lp_lkinfo;

/* This is a private data structure, one per minor device number. */
/* Access to struct lp must be protected by DDI/DKI locks or */
/* synchronization primitives. */

struct lp {
    short flags;/* flags -- see below */
    mblk_t *msg;/* current message being output */
    queue_t *qptr;/* back pointer to write queue */
    lock_t *lck;
};
/* Flags bits */
#define BUSY  1/* device is running and interrupt is pending */

extern struct lp lp_lp[];/* per device lp structure array */
extern int lp_cnt;/* number of valid minor devices */

int lpdevflag = D_MT;
```

Screen 8-1. Line Printer Driver

Configuring a STREAMS driver requires only the `streamtab` structure to be externally accessible. For hardware drivers, the interrupt handler must also be externally accessible. All other STREAMS driver procedures would typically be declared `static`.

The `streamtab` structure must be defined as *prefix*info, where *prefix* is the value of the prefix field in the master file for this driver. The values in the module name and ID fields in the `module_info` structure should be unique in the system. Note that, as in character I/O drivers, `extern` variables are assigned values in the master file when configuring drivers or modules.

The private `lp` structure is indexed by the minor device number and contains these elements:

flags	A set of flags. Only one bit is used: `BUSY` indicates that output is active and a device interrupt is pending.
msg	A pointer to the current message being output.
qptr	A back pointer to the write queue. This is needed to find the write queue during interrupt processing.
lck	A DDI/DKI driver lock to prevent race conditions on the structure.

There is no read-side `put` or `service` procedure. The flow control limits for use on the write-side are 50 bytes for the low water mark and 150 bytes for the high water mark.

Driver Open

The STREAMS mechanism allows only one Stream per minor device. The driver open routine is called whenever a STREAMS device is opened. Opening also allocates a private data structure. The driver open, **`lpopen`** in Screen 8-2, has the same interface as the module open:

```
void lpinit()
{
    register struct lp *lp;

    /*
     * allocate multiprocessor lock for each minor device
     */
    for (lp = lp_lp; lp < &lp_lp[lp_cnt]; lp++)
        lp->lck = LOCK_ALLOC(1, plstr, &lp_lkinfo, KM_SLEEP);
}

int lpopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
{
    struct lp *lp;
    dev_t device;

    if (sflag)   /* check if non-driver open */
        return ENXIO;

    /* check if already open */
    device = getminor(*devp);
    if (device > lp_cnt)
        return ENXIO;
    if (q->q_ptr)
        return EBUSY;

     /* point q_ptr at driver structure */
    lp = &lp_lp[device];
    lp->qptr = WR(q);
    q->q_ptr = (char *)lp;
    WR(q)->q_ptr = (char *)lp;

    /* enable put and srv routines for queue pair */
    qprocson(q);

    return 0;
}
```

Screen 8-2. Driver Open

The Stream flag, *sflag*, must have the value 0, indicating a normal driver open. `devp` is a pointer to the major/minor device number for this port. After checking *sflag*, the STREAMS open flag, **`lpopen`** extracts the minor device pointed to by *devp*, using the **`getminor`** function. *credp* is a pointer to a credentials structure.

The minor device number selects a printer. The device number pointed to by *devp* must be less than `lp_cnt`, the number of configured printers. Otherwise, failure occurs.

The next check, `if (q->q_ptr) . . .` , determines if this printer is already open. If it is, `EBUSY` is returned to avoid merging printouts from multiple users. `q_ptr` is a driver/module private data pointer. It can be used by the driver for any purpose and is initialized to zero by STREAMS. In this example, the driver sets the value of `q_ptr`, in both the read and write `queue` structures, to point to a private data structure for the minor device, `lp_lp[device]`.

There are no physical pointers between queues. **`WR(q)`** generates the write pointer from the read pointer. **`RD(q)`** generates the read pointer from the write pointer, and **`OTHERQ(q)`** generates the mate pointer from either.

Driver Flush Handling

The following write **`put`** procedure, **`lpwput`**, illustrates driver `M_FLUSH` handling. Note that all drivers are expected to incorporate flush handling.

If `FLUSHW` is set, the write message queue is flushed, and (in this example) the leading message (`lp->msg`) is also flushed.

The line `oldpri = LOCK(lp->lck, plstr);` is used to protect the critical code, assuming the device interrupts at level below `plstr`.

Normally, if `FLUSHR` is set, the read queue would be flushed. However, in this example, no messages are ever placed on the read queue, so it is not necessary to flush it. The `FLUSHW` bit is cleared and the message is sent upstream using **`qreply`**. If `FLUSHR` is not set, the message is discarded.

The Stream head always performs the following actions on flush requests received on the read-side from downstream. If `FLUSHR` is set, messages waiting to be sent to user space are flushed. If `FLUSHW` is set, the Stream head clears the `FLUSHR` bit and sends the `M_FLUSH` message downstream. In this way, a single `M_FLUSH` message sent from the driver can reach all queues in a Stream. A module must send two `M_FLUSH` messages to have the same affect.

`lpwput` enqueues `M_DATA` and `M_IOCTL` messages and, if the device is not busy, starts output by calling **`lpout`**. Messages types that are not recognized are discarded.

```
int lpwput(queue_t *q, mblk_t *mp)
{
    register struct lp *lp;
    pl_t oldpri;

    lp = (struct lp *)q->q_ptr;

    switch(mp->b_datap->db_type) {

    default:
        freemsg(mp);
        break;

    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW) {
            /*
             * flush the queue;
             * also flush lp->msg since it is logically
             * at the head of the write queue.
             * access to lp must be locked to protect against
             * potential multiprocessor race.
             */
            flushq(q, FLUSHDATA);
            oldpri = LOCK(lp->lck, plstr);;
            if (lp->msg) {
                freemsg(lp->msg);
                lp->msg = NULL;
            }
            UNLOCK(lp->lck, oldpri);
        }

        if (*mp->b_rptr & FLUSHR) {
            *mp->b_rptr &= ~FLUSHW;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;

    case M_IOCTL:
    case M_DATA:
        putq(q, mp);
        oldpri = LOCK(lp->lck, plstr);
        if (!(lp->flags & BUSY))
            lpout(lp);
        UNLOCK(lp->lck, oldpri);
    }
}
```

Screen 8-3. Flush Handling

Driver Interrupt

`lpint` is the driver interrupt handler routine. **`lpout`** simply takes a character from the queue and sends it to the printer. For convenience, the message currently being output is stored in `lp->msg`. **`lpoutchar`** sends a character to the printer and interrupts when complete. Printer interface options need to be set before being able to print.

Screen 8-4 shows the interrupt routine in the printer driver.

```
/*
 * Device interrupt routine
 */
lpint(int device)
{
    register struct lp *lp;
    pl_t oldpri;
    lp = &lp_lp[device];
    oldpri = LOCK(lp->lck, plstr);
    if (!(lp->flags & BUSY)) {
        UNLOCK(lp->lck, oldpri);
        cmn_err(CE_WARN, “lp: unexpected interrupt\n”);
        return;

    }
    lp->flags &= ~BUSY;
    lpout(lp);
    UNLOCK(lp->lck, oldpri);
}
```

Screen 8-4. Device Interrupt

```
/* Start output to device -- called by put and interrupt routines */
/* argument lp is locked on entry */

lpout(struct lp *lp)
{
    register mblk_t *bp;
    queue_t *q;

    q = lp->qptr;

loop:
    if ((bp = lp->msg) == NULL) {/* no current message */
        if ((bp = getq(q)) == NULL) {
            lp->flags &= ~BUSY;
            return;
        }

        if (bp->b_datap->db_type == M_IOCTL) {
            lpioctl(lp, bp);
            goto loop;
        }

        lp->msg = bp;/* new message */
    }

    if (bp->b_rptr >= bp->b_wptr) {/* validate message */
        bp = lp->msg->b_cont;
        lp->msg->b_cont = null;
        freeb(lp->msg);
        lp->msg = bp;
        goto loop;
    }

    lpoutchar(lp, *bp->b_rptr++);/* output one character */
    lp->flags |= BUSY;
}
```

Driver Close Routine

The driver **`close`** routine is called by the Stream head. Any messages left on the queue are automatically removed by STREAMS. The Stream is dismantled and the data structures are deallocated.

```
static int lpclose(queue_t  *q, int flag, cred_t *credp)
{
    struct lp *lp;
    pl_t oldpri;
    /*
     * disable put and srv routines for q pair
     */
    qprocsoff(q);

    lp = (struct lp *) q->q_ptr;

    /* Free message, queue is automatically flushed by streans */

    oldpri = LOCK(lp->lck, plstr);
    if (lp->msg) {
        freemsg(lp->msg);
        lp->msg = NULL;
    }
    lp->flags = 0;
    UNLOCK(lp->lck, oldpri);
}
```

Screen 8-5. Driver Close Routine

Driver Flow Control

The same utilities and mechanisms used for module flow control are used by drivers.

When the message is queued, **`putq`** increments the value of `q_count` by the size of the message and compares the result against the driver's write high water limit (`q_hiwat`) value. If the count exceeds `q_hiwat`, the **`putq`** utility routine sets the internal `FULL` indicator for the driver write queue. This causes messages from upstream to be halted (**`canputnext`** returns `FALSE`) until the write queue count reaches `q_lowat`. The driver messages waiting to be output are dequeued by the driver output interrupt routine with **`getq`**, which decrements the count. If the resulting count is below `q_lowat`, the **`getq`** routine back-enables any upstream queue that had been blocked.

For priority band data, `qb_count`, `qb_hiwat`, and `qb_lowat` are used.

Device drivers typically discard input when unable to send it to a user process. However, STREAMS allows flow control to be used on the driver read-side to handle temporary upstream blocks.

To some extent, a driver or a module can control when its upstream transmission will become blocked. Control is available through the `M_SETOPTS` message to modify the Stream head read-side flow control limits.

Cloning

In many earlier examples, each user process connected a Stream to a driver by opening a particular minor device of that driver. Often, however, a user process had to connect a new Stream to a driver regardless of which minor device is used to access the driver. In the past, this typically forced the user process to poll the various minor device nodes of the driver for an available minor device. To alleviate this task, a facility called “clone open” is supported for STREAMS drivers. If a STREAMS driver is implemented as a clonable device, a single node in the file system may be opened to access any unused device that the driver controls. This special node guarantees that the user is allocated a separate Stream to the driver on every **`open`** call. Each Stream is associated with an unused major/minor device, so the total number of Streams that may be connected to a particular clonable driver is limited by the number of minor devices configured for that driver.

The clone device may be useful, for example, in a networking environment where a protocol pseudo-device driver requires each user to open a separate Stream over which it establishes communication.

Note, however, that a race can occur when simultaneous cloning opens and non-cloning opens are in progress. A clone driver must detect this race and return `ECLNRACE` (a system errno) for the non-cloning open. The FS layers above will detect this errno and restart the open.

NOTE

The decision to implement a STREAMS driver as a clonable device is made by the designers of the device driver.

Knowledge of clone driver implementation is not required. A description is presented here for completeness and to assist developers who must implement their own clone driver.

There are two ways to create a clone device node in the file system. The first is to have a node with the major number of the clone driver and with a minor number equal to the major number of the real device one wants to open. For example, **`/dev/net00`** might be major 40, minor 0 (normal open), and **`/dev/net`** might be major 4 (the major number of the clone driver) minor 40 (the major number of the real device).

The second way to create a clone device node is for the driver to designate a special minor device as its clone entry point. Here, **`/dev/net`** might be major 40, minor 0 (clone open).

The former example causes *sflag* to be set to `CLONEOPEN` in the open routine when **`/dev/net`** is opened. The latter will not. Instead, in the latter case the driver has decided to designate a special minor device as its clone interface. When the clone is opened, the driver knows that it should look for an unused minor device. This implies that the reserved minor for the clone entry point will never be given out.

In either case, the driver returns the new device number as

```
*devp = makedevice(getemajor(*devp), newminor);
```

NOTE

`makedevice` is unique to the DDI. If the DDI is not used, **`makedev`** can be used instead of **`makedevice`**.

Loop-around Driver

The loop-around driver is a pseudo-driver that loops data from one open Stream to another open Stream. The user processes see the associated files almost like a full-duplex pipe. The Streams are not physically linked. The driver is a simple multiplexor that passes messages from one Stream's write queue to the other Stream's read queue.

To create a connection, a process opens two Streams, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an **`I_STR ioctl(2)`** to the other Stream. For each **`open`**, the driver open places the passed `queue` pointer in a driver interconnection table, indexed by the device number. When the driver later receives the **`I_STR`** as an `M_IOCTL` message, it uses the device number to locate the other Stream's interconnection table entry, and stores the appropriate `queue` pointers in both of the Streams' interconnection table entries.

Subsequently, when messages other than `M_IOCTL` or `M_FLUSH` are received by the driver on either Stream's write-side, the messages are switched to the read queue following the driver on the other Stream's read-side. The resultant logical connection is shown in Figure 8-2. In Figure 8-2, the abbreviation QP represents a queue pair. Flow control between the two Streams must be handled by special code since STREAMS does not automatically propagate flow control information between two Streams that are not physically interconnected.

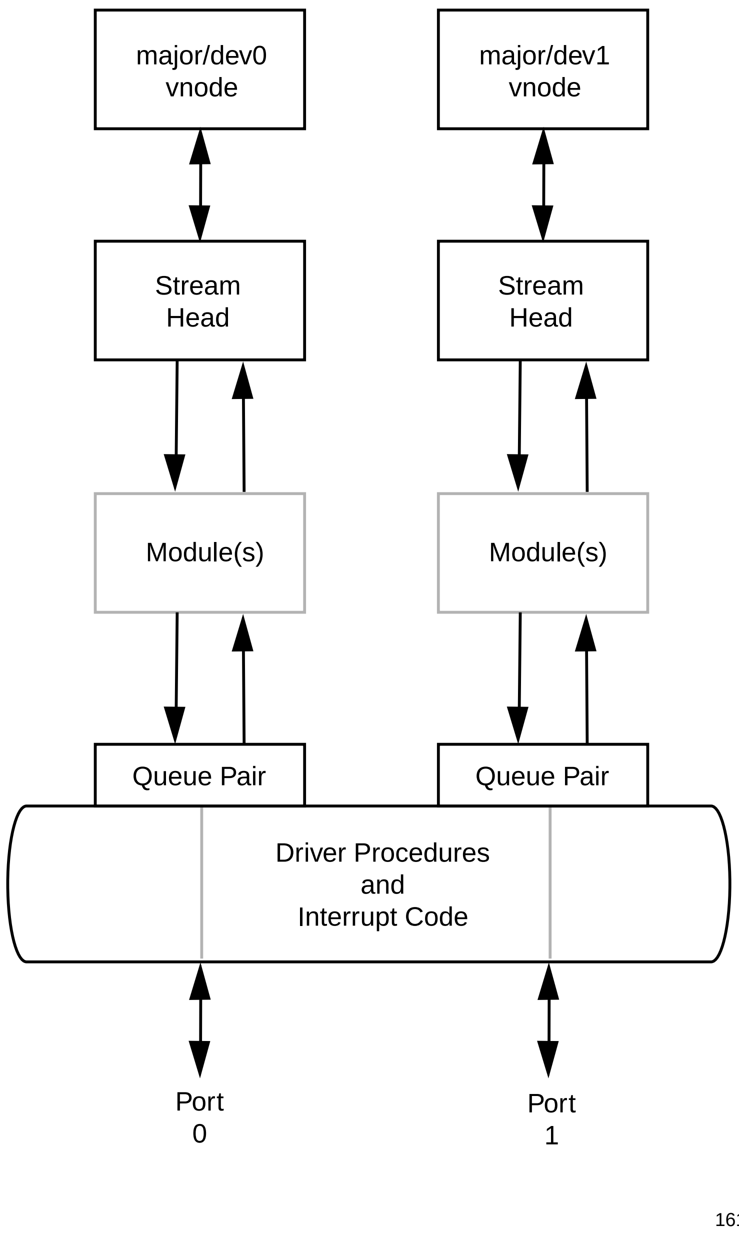


Figure 8-2. Loop-Around Streams

The next example shows the loop-around driver code. The `loop` structure contains the interconnection information for a pair of Streams. `loop_loop` is indexed by the minor device number. When a Stream is opened to the driver, the address of the corresponding `loop_loop` element is placed in `q_ptr` (private data structure pointer) of the read-side and write-side `queue`s. Because STREAMS clears `q_ptr` when the `queue` is allocated, a `NULL` value of `q_ptr` indicates an initial **open**. `loop_loop` verifies that this Stream is connected to another open Stream. This example driver uses coarse-grained locking for simplicity.

The declarations for the driver are shown in Screen 8-6:

```
/* Loop-around driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>

static struct module_info minfo = {
    0xee12, “loop”, 0, INFPSZ, 512, 128};

static int loopopen(queue_t *, dev_t *, int, int, cred_t *);
static int loopclose(quque_t *, int, cred_t *);
static int loopwput(queue_t *, mblk_t *);
static int loopwsrv(queue_t *);
static int looprsrv(queue_t *);

static struct qinit rinit = {
    NULL, looprsrv, loopopen, loopclose, NULL, &minfo, NULL};

static struct qinit winit = {
    loopwput, loopwsrv, NULL, NULL, NULL, &minfo, NULL};

struct streamtab loopinfo = {&rinit, &winit, NULL, NULL};

lkinfo_t loop_lkinfo;
lock_t *loop_lck;

struct loop {
    queue_t*qptr;/* back pointer to write queue */
    queue_t *oqptr;/* pointer to connected read queue */
}

#define LOOP_SET (('l'<<8)|1)/* should be in a .h file */

extern struct loop loop_loop[];
extern int loop_cnt;

int loopdevflag = D_MT;
```

Screen 8-6. Driver Declarations

The open procedure includes canonical clone processing that enables a single file system node to yield a new minor device/vnode each time the driver is opened as shown in Figure 8-7:

```
void loopinit()
{
    loop_lock = LOCK_ALLOC(LOOPHIER, plstr, &loop_lkinfo, KM_SLEEP);
}

int loopopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *credp)
{
    struct loop *loop;
    dev_t newminor;
    pl_t pl;

    if (q->q_ptr)/* already open */
        return(0);

/*
 * If CLONEOPEN, pick a minor device number to use.
 * Otherwise, check the minor device range.
 */
    pl = LOCK(loop_lock, plstr);
    if (sflag == CLONEOPEN) {
        for (newminor = 0; newminor < loop_cnt; newminor++) {
            if (loop_loop[newminor].qptr == NULL)
                break;
        }
    } else
        newminor = geteminor(*devp);

    if (newminor >= loop_cnt) {
        UNLOCK(loop_lock, pl);
        return(ENXIO);
    }

    /* build new device number and reset devp */
    /* getmajor gets the external major number, if (sflag == CLONEOPEN) */

    *devp = makedev(getemajor(*devp), newminor);
    loop = &loop_loop[newminor];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);
    loop->oqptr = NULL;
    UNLOCK(loop_lock, pl);

    /* enable put and srv routines for this queue pair */
    qprocson(q);
    return(0);
}
```

Screen 8-7. Open Procedure

In **loopopen**, *sflag* can be CLONEOPEN, indicating that the driver should pick an unused minor device (that is, the user does not care which minor device is used). In this example, the driver scans its private loop_loop data structure to find an unused minor device number. If *sflag* has not been set to CLONEOPEN, the passed-in minor device specified by geteminor (*devp) is used.

Because the messages are switched to the read queue following the other Stream's read-side, the driver needs a **put** procedure only on its write-side.

loopwput shows another use of an **I_STR ioctl** call (see “Module and Driver ioctls”). The driver supports a LOOP_SET value of ioc_cmd in the iocblk of the M_IOCTL message. LOOP_SET instructs the driver to connect the current open Stream to the Stream identified in the message. The second block of the M_IOCTL message holds an integer that specifies the minor device number of the Stream to connect to.

The driver performs the following sanity checks:

- The data in the second block is checked for the proper amount
- The range of the `to` device is checked
- The `to` device is checked to see if it is open
- The current Stream is checked to see if it is disconnected
- The `to` Stream is checked to see if it is disconnected

If everything checks out, the read `queue` pointers for the two Streams are stored in the respective `oqptr` fields. This cross-connects the two Streams indirectly, using `loop_loop`.

Canonical flush handling is incorporated in the **`put`** procedure.

Finally, **`loopwput`** enqueues all other messages (for example, `M_DATA` or `M_PROTO`) for processing by its **`service`** procedure. A check is made to see if the Stream is connected. If not, an `M_ERROR` is sent upstream to the Stream head.

```
int loopwput(queue_t *q, mblk_t *mp)
{
    register struct loop *loop;
    pl_t pl;

    loop = (struct loop *) q->q_ptr;

    switch (mp->b_datap->db_type) {

    case M_IOCTL: {
        struct iocblk *iocp;
        int error;

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case LOOP_SET: {
            int to;/* other minor device */

            /*
             * Sanity check.  ioc_count contains the amount of
             * user supplied data that must equal the size of
             * an int.
             */

            if (iocp->ioc_count != sizeof(int)) {
                error = EINVAL;
                goto iocnak;
            }

            /* fetch other dev from 2nd message block */
            to = *(int *)mp->b_cont->b_rptr;

            /*
             * More sanity checks.  The minor must be in range, open
             * already.  Also, this device and the other one must be
             * disconnected.
             */

            pl = LOCK(loop_lock, plstr);
            if (to >= loop_cnt || to < 0 || !loop_loop[to].qptr) {
                error = ENXIO;
                UNLOCK(loop_lock, pl);
                goto iocnak;
            }
            if (loop->oqptr || loop_loop[to].oqptr) {
                error = EBUSY;
                UNLOCK(loop_lock, pl);
                goto iocnak;
            }

            /* Cross connect streams using the loop structures */
```

Screen 8-8. Driver Sanity Checks

```
            loop->oqptr = RD(loop-loop[to].qptr);
            loop_loop[to].oqptr = RD(q);

            UNLOCK(loop_lock, pl);
            /*
             * Return successful ioctl.  Set ioc_count to zero,
             * since no data is returned.
             */
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
        }
        default:
            error = EINVAL;
iocnak:
            /*
             * Bad ioctl.  Setting ioc_error causes the ioctl
             * call to return that particular errno.  By default,
             * ioctl will return EINVAL on failure.
             */
            mp->b_datap->db_type = M_IOCNAK;
            iocp->ioc_error = error;/* set returned errno */
            qreply(q, mp);
        }
        break;
    }

    case M_FLUSH:
        pl = LOCK(loop_lock, plstr);
        if (*mp->b_rptr & FLUSHW) {
            flushq(q, FLUSHALL);/* write */
            if (loop->oqptr != NULL)
                flushq(loop->oqptr, FLUSHALL);
                /* read on other side equals write on this side */
        }
        if (*mp->b_rptr & FLUSHR) {
            flushq(RD(q), FLUSHALL);
            if (loop->oqptr != NULL)
                flushq(WR(loop->oqptr), FLUSHALL);
        }
        UNLOCK(loop_lock, pl);
        switch(*mp->b_rptr) {

        case FLUSHW:
            *mp->b_rptr = FLUSHR;
            break;

        case FLUSHR:
            *mp->b_rptr = FLUSHW;
            break;
        }
        pl = LOCK(loop_lock, plstr);

        if (loop->oqptr != NULL) {
            UNLOCK(loop_lock, pl);
            /*
             * loop->oqptr can only be cleared in loopclose, which
             * can not be called while the put procedure is executing
             */
            putnext(loop->oqptr, mp);
        }
        else
            UNLOCK(loop_lock, pl);
        break;

    default:/* If this Stream isn't connected, send M_ERROR upstream */
        pl = LOCK(loop_lock, plstr);
        if (loop->oqptr == NULL) {
            UNLOCK(loop_lock, pl);
            freemsg(mp);
            putnextctl1(RD(q), M_ERROR, ENXIO);
            break;
        }
        UNLOCK(loop_lock, pl);
        putq(q, mp);
    }
}
```

Certain message types can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user process(es). The messages may also change the state of the Stream head:

`M_ERROR`	Causes the Stream head to lock up. Message transmission between Stream and user processes is terminated. All subsequent system calls except **`close(2)`** and **`poll(2)`** will fail. Also causes an `M_FLUSH` clearing all message queues to be sent downstream by the Stream head.
`M_HANGUP`	Terminates input from a user process to the Stream. All subsequent system calls that would send messages downstream will fail. Once the Stream head read message queue is empty, EOF is returned on reads. Can also result in the `SIGHUP` signal being sent to the process group.
`M_SIG/M_PCSIG`	Causes a specified signal to be sent to a process.

`putnextctl1` and **`putnextctl`** are utilities that allocate a nondata (that is, not `M_DATA`, `M_DELAY`, `M_PROTO`, or `M_PCPROTO`) type message, place one byte in the message (for **`putnextctl1`**), and call the **`put`** procedure of the queue next to the specified queue.

`service` procedures are required in Screen 8-9 on both the write-side and read-side for flow control:

```
static int loopwsrv(queue_t *q)
{
    mblk_t *mp;
    register struct loop *loop;
    pl_t pl;

    loop = (struct loop *) q->q_ptr;

    while ((mp = getq(q)) != NULL) {
        /*
         * Check if we can put the message up the other Stream read
         * queue.
         */
        pl = LOCK(loop_lock, plstr);
        if (pcmsg(mp->b_datap->db_type) && !canputnext(loop->oqptr)) {
            UNLOCK(loop_lock, pl);
            putbq(q, mp);/* read-side is blocked */
            break;
        }
        /* send message */
        /*
         * loopwput verified that loop->oqptr was set and it can only
         * be cleared in the close routine, which can not be called
         * while this queue was enabled.
         */
        putnext(loop->oqptr, mp);/* To queue following other
                            Stream read queue */
    }
}

/*
 * read service routine
 * Enter only when “back enabled” by flow control
 */
static int looprsrv(queue_t *q)
{
    struct loop *loop;
    pl_t pl;

    loop = (struct loop *) q->q_ptr;
    pl = LOCK(loop_lock, plstr);
    if (loop->oqptr != NULL)
        /* manually enable write service procedure */
        UNLOCK(loop_lock, pl);
        qenable(WR(loop->oqptr));
    } else
        UNLOCK(loop_lock, pl);
}
```

Screen 8-9. Write and Read Side Flow Control

The write **`service`** procedure, **`loopwsrv`**, takes on the canonical form. The queue being written to is not downstream, but upstream (found by using `oqptr`) on the other Stream.

In Screen 8-10, there is no read-side **`put`** procedure so the read **`service`** procedure, **`looprsrv`**, is not scheduled by an associated **`put`** procedure, as has been done previously. **`looprsrv`** is scheduled only by being back-enabled when its upstream becomes unstuck from flow control blockage. The purpose of the procedure is to re-enable the writer (**`loopwsrv`**) by using `oqptr` to find the related `queue`. **`loopwsrv`** cannot be directly back-enabled by STREAMS because there is no direct `queue` linkage between the two Streams. Note that no message ever gets queued to the read **`service`** procedure. Messages are kept on the write-side so that flow control can propagate up to the Stream head. The **`qenable`** routine schedules the write-side **`service`** procedure of the other Stream.

`loopclose` breaks the connection between the Streams:

```
int loopclose(queue_t *q, int flag, cred_t *credp)
{
    register struct loop *loop;
    pl_t pl;

    /* disable put and srv routines for queue pair. */
    qprocsoff(q);
    pl = LOCK(loop_lock, plstr);
    loop = (struct loop *) q->q_ptr;
    loop->qptr = NULL;

    /*
     * If we are connected to another stream, break the linkage, and send
     * a hangup message.  The hangup message causes the stream head to fail
     * writes, allow the queued data to be read completely, and then
     * return EOF on subsequent reads.
     */
    if (loop->oqptr) {
        ((struct loop *)loop->oqptr->q_ptr)->oqptr = NULL;
        UNLOCK(loop_lock, pl);
        putnextctl(loop->oqptr, M_HANGUP);
        pl = LOCK(loop_lock, plstr);
        loop->oqptr = NULL;
    }
    UNLOCK(loop_lock, plstr);
}
```

Screen 8-10. Re-enabling the Writer

`loopclose` sends an `M_HANGUP` message up the connected Stream to the Stream head.

NOTE

A loop-around driver must never directly link the `q_next` pointers of the queue pairs of the two Streams.

Design Guidelines

Driver developers should follow these guidelines:

- Messages that are not understood by the drivers should be freed.
- A driver must process an `M_IOCTL` message. Otherwise, the Stream head blocks for an `M_IOCNAK` or `M_IOCACK` until the timeout (potentially infinite) expires.
- If a driver does not understand an **`ioctl`**, an `M_IOCNAK` message must be sent to upstream.
- Terminal drivers must always acknowledge the `EUC` **`ioctl`**s whether they understand them or not.

- If a driver wants to allocate a controlling terminal, it should send an `M_SETOPTS` message with the `SO_ISTTY` flag set upstream.
- A driver must be a part of the kernel for it to be opened.
- A multithreaded driver is responsible for protecting driver-specific data against multiprocessor race conditions.

NOTE

For information regarding the loadable STREAMS drivers, see the *Device Driver Programming* manual.

9
STREAMS Multiplexing

9 STREAMS Multiplexing

Introduction

This section describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers. A STREAMS multiplexor is a driver with multiple Streams connected to it. The primary function of the multiplexing driver is to switch messages among the connected Streams. Multiplexor configurations are created at user level by system calls.

STREAMS-related system calls set up the "plumbing," or Stream interconnections, for multiplexing drivers. The subset of these calls that allows a user to connect (and disconnect) Streams below a driver is referred to as the multiplexing facility. This type of connection is referred to as a 1-to-M, or lower, multiplexor configuration. This configuration must always contain a multiplexing driver, which is recognized by STREAMS as having special characteristics.

Multiple Streams can be connected above a driver by **`open(2)`** calls. This was done for the loop-around driver and for the driver handling multiple minor devices in "STREAMS Drivers." There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple Streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with Streams connected above is referred to as an N-to-1, or upper, multiplexor. STREAMS does not provide any facilities beyond **`open(2)`** and **`close(2)`** to connect or disconnect upper Streams for multiplexing purposes.

From the driver's perspective, upper and lower configurations differ only in how they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexor drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiplexed Streams.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver.

As discussed in "STREAMS Drivers," the multiple Streams that represent minor devices are actually distinct Streams in which the driver keeps track of each Stream attached to it. The STREAMS subsystem does not recognize any relationship between the Streams. The same is true for STREAMS multiplexors of any configuration. The multiplexed Streams are distinct and the driver must be implemented to do most of the work.

In addition to upper and lower multiplexors, more complex configurations can be created by connecting Streams containing multiplexors to other multiplexor drivers. With such a diversity of needs for multiplexors, it is not possible to provide general-purpose multiplexor drivers. STREAMS provides a general purpose multiplexing facility that allows users to set up the intermodule/driver plumbing to create multiplexor configurations of generally unlimited interconnection.

Building a Multiplexor

This section builds a protocol multiplexor with the multiplexing configuration shown in Figure 9-1. To free users from the need to know about the underlying protocol structure, a user-level daemon process is built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper Stream to one of two lower Streams. This driver supports two STREAMS connections beneath it. These connections are to two distinct networks; one for the IEEE 802.3 standard with the 802.3 driver, and the other to the IEEE 802.4 standard with the 802.4 driver. The TP driver multiplexes upper Streams over a single Stream to the IP driver.

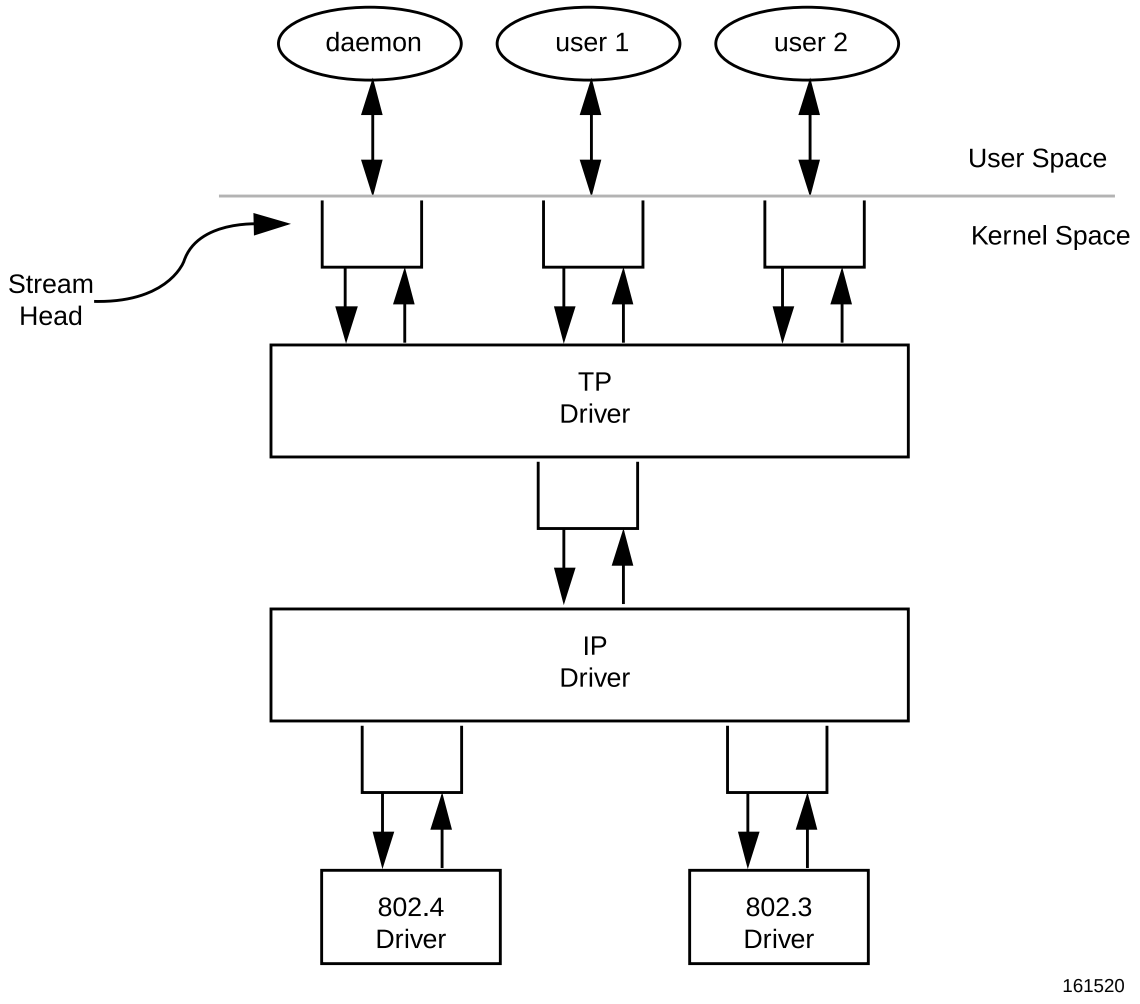


Figure 9-1. Protocol Multiplexor

The following example shows how this daemon process sets up the protocol multiplexor. The necessary declarations and initialization for the daemon program are shown in Screen 9-1:

```
#include <fcntl.h>
#include <stropts.h>

main()
{
    int fd_802_4,
        fd_802_3,
        fd_ip,
        fd_tp;

    /* daemonize this process */

    switch (fork()) {
    case 0:
        break;
    case -1:
        perror("fork failed");
        exit(2);
    default:
        exit(0);
    }
    setsid();
```

Screen 9-1. Daemon Program Declarations and Initialization

This multilevel multiplexed Stream configuration is built from the bottom up. Therefore, Screen 9-1 begins by first constructing the Internal Protocol (IP) multiplexor. This multiplexing device driver is treated like any other software driver. It owns a node in the PowerMAX OS file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate Streams above each driver as shown in Figure 9-2 The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the **`I_LINK ioctl`** call.

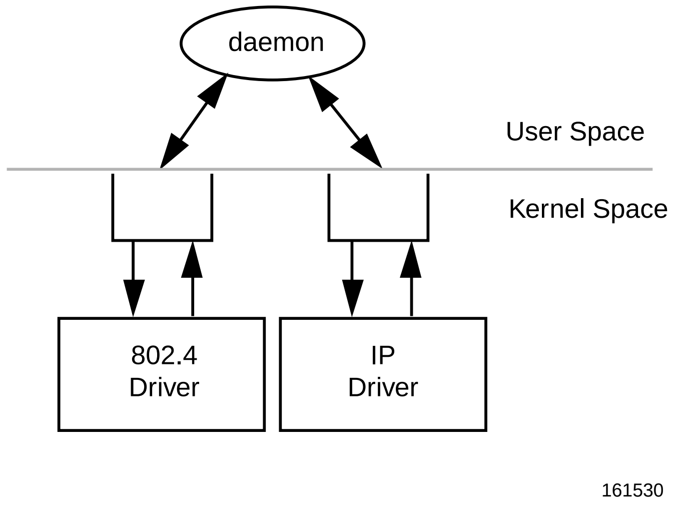


Figure 9-2. Before Link

The sequence of instructions to this point is

```
if ((fd_802_4 = open(“/dev/802_4”, O_RDWR)) < 0) {
    perror(“open of /dev/802_4 failed”);
    exit(1);
}

if ((fd_ip = open(“/dev/ip”, O_RDWR)) < 0) {
    perror(“open of /dev/ip failed”);
    exit(2);
}

/* now link 802.4 to underside of IP */

if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
    perror(“I_LINK ioctl failed”);
    exit(3);
}
```

`I_LINK` takes two file descriptors as arguments. The first file descriptor, `fd_ip`, must reference the Stream connected to the multiplexing driver, and the second file descriptor, `fd_802_4`, must reference the Stream to be connected below the multiplexor. Figure 9-3 shows the state of these Streams following the **`I_LINK`** call. The complete Stream to the 802.4 driver has been connected below the IP driver. The Stream head's queues of the 802.4 driver is used by the IP driver to manage the lower half of the multiplexor.

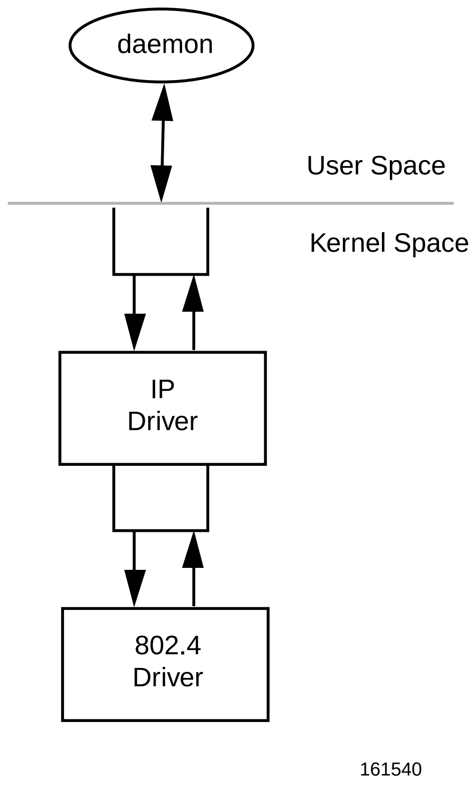


Figure 9-3. IP Multiplexor after First Link

`I_LINK` returns an integer value, called `muxid`, which is used by the multiplexing driver to identify the Stream just connected below it. This `muxid` is ignored in the example, but is useful for dismantling a multiplexor or routing data through the multiplexor. Its significance is discussed later.

The following sequence of system calls is used to continue building the internetworking protocol multiplexor (IP):

```
if ((fd_802_3 = open(“/dev/802_3”, O_RDWR)) < 0) {
     perror(“open of /dev/802_3 failed”);
     exit(4);
}

if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
     perror(“I_LINK ioctl failed”);
     exit(5);
}
```

All links below the IP driver have now been established, giving the configuration in Figure 9-5

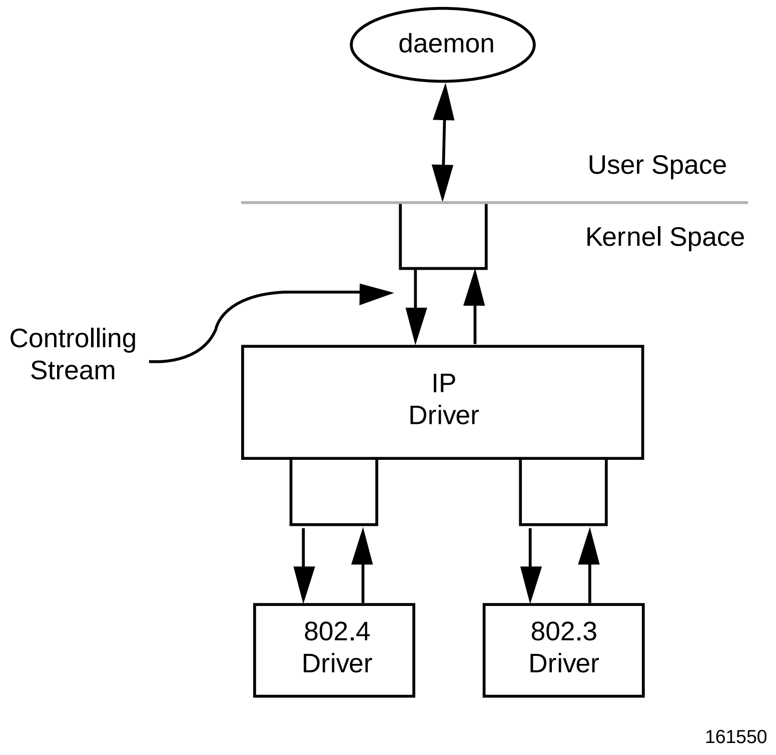


Figure 9-4. IP Multiplexor

The Stream above the multiplexing driver used to establish the lower connections is the controlling Stream and has special significance when dismantling the multiplexing configuration. This will be illustrated later in this section. The Stream referenced by `fd_ip` is the controlling Stream for the IP multiplexor.

NOTE

> The order in which the Streams in the multiplexing configuration are opened is unimportant. If it is necessary to have intermediate modules in the Stream between the IP driver and media drivers, these modules must be added to the Streams associated with the media drivers (using **`I_PUSH`**) before the media drivers are attached below the multiplexor.

The number of Streams that can be linked to a multiplexor is restricted by the design of the particular multiplexor. The manual page describing each driver describes such restrictions. See *Device Driver Reference.* However, only one **`I_LINK`** operation is allowed for each lower Stream; a single Stream cannot be linked below two multiplexors simultaneously.

Continuing with the example, the IP driver is now linked below the transport protocol (TP) multiplexing driver. As seen in Figure 9-4, only one link is supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", O_RDWR)) < 0) {
    perror("open of /dev/tp failed");
    exit(6);
}

if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
    perror("I_LINK ioctl failed");
    exit(7);
}
```

The multilevel multiplexing configuration shown in Figure 9-5 has now been created.

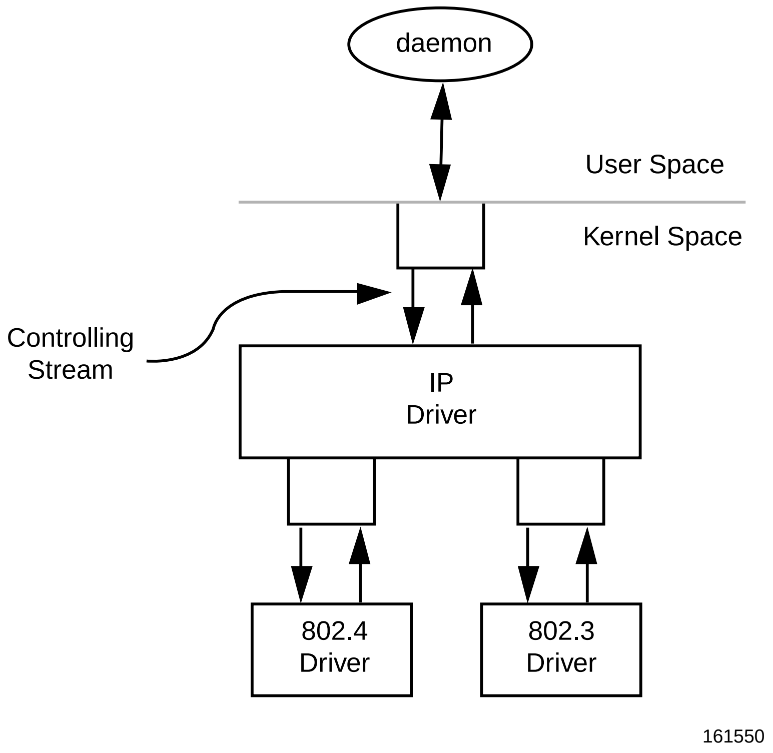


Figure 9-5. TP Multiplexor

Because the controlling Stream of the IP multiplexor has been linked below the TP multiplexor, the controlling Stream for the new multilevel multiplexor configuration is the Stream above the TP multiplexor.

At this point, the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexor. If these file descriptors are not closed, all later **`read`**, **`write`**, **`ioctl`**, **`poll`**, **`getmsg`** and **`putmsg`** (or **`getmsg(2)`** and **`putmsg(2))`** system calls issued to them will fail because **`I_LINK`** associates the Stream head of each linked Stream with the multiplexor, so the user may not access that Stream directly for the duration of the link.

The following sequence of system calls completes the daemon example:

```
    close(fd_802_4);
    close(fd_802_3);
    close(fd_ip);

    /* Hold multiplexor open forever */
    pause();
}
```

The transport driver supports several simultaneous Streams. These Streams are multiplexed over the single Stream connected to the IP multiplexor. The mechanism for establishing multiple Streams above the transport multiplexor is actually a by-product of the way in which Streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate Streams are connected to that driver. Of course, the driver must be designed with the intelligence to route data from the single lower Stream to the appropriate upper Stream.

The daemon process maintains the multiplexed Stream configuration through an open Stream (the controlling Stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new Streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and subnetworks that support the transport service.

Multilevel multiplexing configurations should be assembled from the bottom up because the passing of **`ioctl`**s through the multiplexor is determined by the multiplexing driver and cannot generally be relied on.

Dismantling a Multiplexor

Streams connected to a multiplexing driver from above with **`open`**, can be dismantled by closing each Stream with **`close`**. The mechanism for dismantling Streams that have been linked below a multiplexing driver is less obvious, and is described below.

The **`I_UNLINK`** **`ioctl`** call disconnects each multiplexor link below a multiplexing driver individually. This command has the form:

`ioctl``(`*fd*`, ` **`I_UNLINK`**`, ` *muxid*`);`

where *fd* is a file descriptor associated with a Stream connected to the multiplexing driver from above, and *muxid* is the identifier that was returned by **`I_LINK`** when a driver was linked below the multiplexor. Each lower driver may be disconnected individually in this way, or a special *muxid* value of `-1` may disconnect all drivers from the multiplexor simultaneously.

In the multiplexing daemon program shown in Figure 9-1, the multiplexor is never explicitly dismantled because all links associated with a multiplexing driver are automatically dismantled when the controlling Stream associated with that multiplexor is closed. Because the controlling Stream is open to a driver, only the final call of **`close`** for that Stream closes it. In this example, the daemon is the only process that opens the controlling Stream, so the multiplexing configuration is dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multilevel, multiplexed Stream configuration, the controlling Stream for each multiplexor at each level must be linked under the next higher level multiplexor. In the example, the controlling Stream for the IP driver was linked under the TP driver, which resulted in a single controlling Stream for the full, multilevel configuration. Because the multiplexing program relied on closing the controlling Stream to dismantle the multiplexed Stream configuration instead of using explicit **`I_UNLINK`** calls, the *muxid* values returned by **`I_LINK`** could be ignored.

An important side-effect of automatic dismantling on the close is that it is not possible for a process to build a multiplexing configuration with **`I_LINK`** and then exit. This is because **`exit(2)`** closes all files associated with the process, including the controlling Stream. To keep the configuration intact, the process must exist for the life of that multiplexor. That is the motivation for implementing the example as a daemon process.

However, if the process uses persistent links with the **`I_PLINK`** **`ioctl`** call, the multiplexor configuration remains intact after the process exits. Persistent links are described later in this chapter.

Routing Data through a Multiplexor

STREAMS provides a mechanism for building multiplexed Stream configurations. However, the criteria on which a multiplexor routes data is driver-dependent. For example, the protocol multiplexor shown before might use address information found in a protocol header to determine over which subnetwork data should be routed. It is the multiplexing driver's responsibility to define its routing criteria.

One routing option available to the multiplexor is to use the `muxid` value to determine to which Stream data should be routed (remember that each multiplexor link is associated with a *muxid*). **`I_LINK`** passes the *muxid* value to the driver and returns this value to the user. The driver can therefore specify that the *muxid* value must accompany data routed through it. For example, if a multiplexor routed data from a single upper Stream to one of several lower Streams (as did the IP driver), the multiplexor could require the user to insert the `muxid` of the desired lower Stream into the first four bytes of each message passed to it. The driver could then match the `muxid` in each message with the `muxid` of each lower Stream, and route the data accordingly.

Connecting/Disconnecting Lower Streams

Multiple Streams are created above a driver/multiplexor with the **`open`** system call on either different minor devices, or on a clonable device file. Note that any driver that handles more than one minor device is considered an upper multiplexor.

To connect Streams below a multiplexor requires additional software within the multiplexor. The main difference between STREAMS lower multiplexors and STREAMS device drivers are that multiplexors are pseudo-devices and that multiplexors have two additional `qinit` structures, pointed to by fields in the `streamtab` structure: the *lower half* read-side `qinit` and the *lower half* write-side `qinit`.

The multiplexor is conceptually divided into two parts: the lower half (bottom) and the upper half (top). The multiplexor `queue` structures that have been allocated when the multiplexor was opened, use the usual `qinit` entries from the multiplexor's `streamtab`. This is the same as any open of the STREAMS device. When a lower Stream is linked beneath the multiplexor, the `qinit` structures at the Stream head are substituted by the bottom half `qinit` structures of the multiplexors. Once the linkage is made, the multiplexor switches messages between upper and lower Streams. When messages reach the top of the lower Stream, they are handled by **`put`** and **`service`** routines specified in the bottom half of the multiplexor.

Connecting Lower Streams

A lower multiplexor is connected as follows: the initial **open** to a multiplexing driver creates a Stream, as in any other driver. **open** uses the first two `streamtab` structure entries to create the driver queues. At this point, the only distinguishing characteristic of this Stream are non-`NULL` entries in the `streamtab` `st_muxrinit` and `st_muxwinit` fields.

These fields are ignored by **`open`** (see the rightmost Stream in Figure 9-6). Any other Stream subsequently opened to this driver will have the same `streamtab` and thereby the same mux fields.

Next, another file is opened to create a (soon to be) lower Stream. The driver for the lower Stream is typically a device driver (see the leftmost Stream in Figure 9-6). This Stream has no distinguishing characteristics. It can include any driver compatible with the multiplexor. Any modules required on the lower Stream must be pushed onto it now.

Next, this lower Stream is connected below the multiplexing driver with an **`I_LINK`** **`ioctl`** call (see **`streamio(7)`**). The Stream head points to the Stream head routines as its procedures (known by its `queue`). An **`I_LINK`** to the upper Stream, referencing the lower Stream, causes STREAMS to modify the contents of the Stream head's queues in the lower Stream. The pointers to the Stream head routines, and other values, in the Stream head's queues are replaced with those contained in the mux fields of the multiplexing driver's `streamtab`. Changing the Stream head routines on the lower Stream means that all subsequent messages sent upstream by the lower Stream's driver, eventually, are passed to the **`put`** procedure designated in `st_muxrinit`, the multiplexing driver. The **`I_LINK`** also establishes this upper Stream as the control Stream for this lower Stream. STREAMS remembers the relationship between these two Streams until the upper Stream is closed, or the lower Stream is unlinked.

Finally, the Stream head sends an `M_IOCTL` message with `ioc_cmd` set to **`I_LINK`** to the multiplexing driver. The `M_DATA` part of the `M_IOCTL` contains a `linkblk` structure. The multiplexing driver stores information from the `linkblk` structure in private storage and returns an `M_IOCACK` message (acknowledgment). `l_index` is returned to the process requesting the **`I_LINK`**. This value can be used later by the process to disconnect this Stream.

An **`I_LINK`** is required for each lower Stream connected to the driver. Additional upper Streams can be connected to the multiplexing driver by **`open`** calls. Any message type can be sent from a lower Stream to user processes along any of the upper Streams. The upper Streams provide the only interface between the user processes and the multiplexor.

Note that no direct data structure linkage is established for the linked Streams. The read queue's `q_next` is `NULL` and the write queue's `q_next` points to the first entity on the lower Stream. Messages flowing upstream from a lower driver (a device driver or another multiplexor) enters the multiplexing driver **`put`** procedure with `l_qbot` as the `queue` value. The multiplexing driver has to route the messages to the appropriate upper (or lower) Stream. Similarly, a message coming downstream from user space on any upper Stream has to be processed and routed, if required, by the driver.

Also note that the lower Stream (see the headers and file descriptors) is no longer accessible from user space. This causes all system calls to the lower Stream to return `EINVAL`, except for **`close`**. This is why all modules have to be in place before the lower Stream is linked to the multiplexing driver.

Finally, note that the absence of direct linkage between the upper and lower Streams means that STREAMS flow control has to be handled by special code in the multiplexing driver. The flow control mechanism cannot see across the driver.

In general, multiplexing drivers should be implemented so that new Streams can be dynamically connected to (and existing Streams disconnected from) the driver without interfering with its ongoing operation. The number of Streams that can be connected to a multiplexor is developer-dependent.

Disconnecting Lower Streams

Dismantling a lower multiplexor is done by disconnecting (unlinking) the lower Streams. Unlinking can be initiated in three ways:

- An **`I_UNLINK ioctl`** references a specific Stream
- An **`I_UNLINK`** references all lower Streams
- The last **`close`** of the control Stream performs the unlinking

As in the link, an unlink sends a `linkblk` structure to the driver in an `M_IOCTL` message. In the first bullet item, **`I_UNLINK`** uses the `l_index` value returned in the **`I_LINK`** to specify the lower Stream to be unlinked. In the second and third bullet items, the calls must designate a file corresponding to a control Stream which causes all the lower Streams that were previously linked by this control Stream to be unlinked. The driver sees a series of individual unlinks.

If no open references exist for a lower Stream, a subsequent unlink automatically closes the Stream. Otherwise, the lower Stream must be closed by **`close`** following the unlink. STREAMS automatically dismantles all cascaded multiplexors (below other multiplexing Streams) if their controlling Stream is closed. An **`I_UNLINK`** leaves lower, cascaded multiplexing Streams intact unless the Stream file descriptor was previously closed.

Multiplexor Construction Example

This section describes an example of multiplexor construction and usage. Figure 9-6 shows the Streams before their connection to create the multiplexing configuration of Figure 9-7. Multiple upper and lower Streams interface to the multiplexor driver. The user processes of Figure 9-5 are not shown in Figure 9-6.

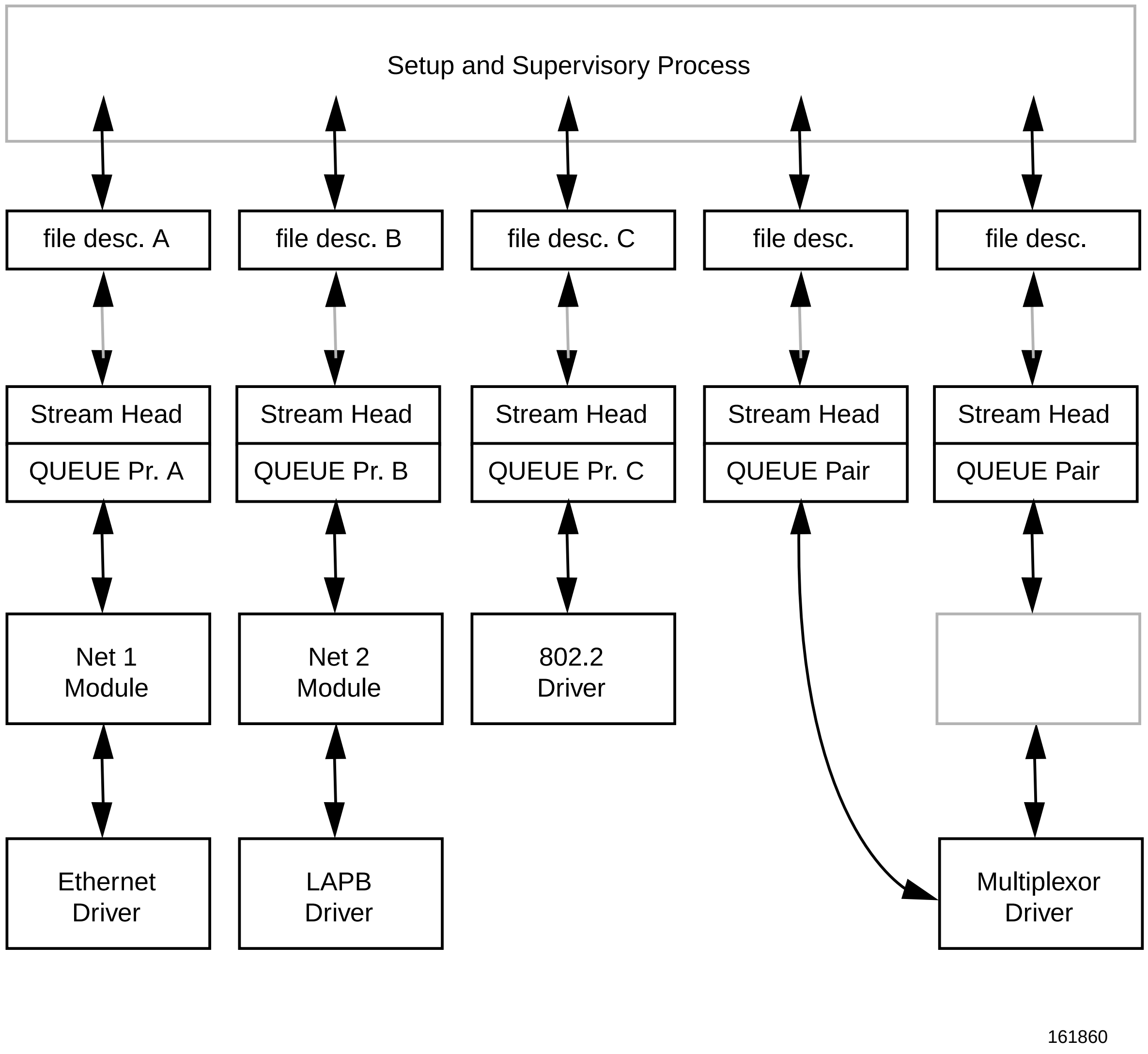


Figure 9-6. Internet Multiplexor before Connecting

The Ethernet^{TM}, LAPB, and IEEE 802.2 device drivers terminate links to other nodes. The multiplexor driver is an Internet Protocol (IP) multiplexor that switches data among the various nodes or sends data upstream to a user(s) in the system. The Net modules typically provide a convergence function, which matches the multiplexor driver and device driver interface.

Figure 9-6 depicts only a portion of the full, larger Stream. In the dotted rectangle above the IP multiplexor, there generally is an upper transport control protocol (TCP) multiplexor, additional modules and, possibly, additional multiplexors in the Stream. Multiplexors can also be cascaded below the IP driver if the device drivers are replaced by multiplexor drivers.

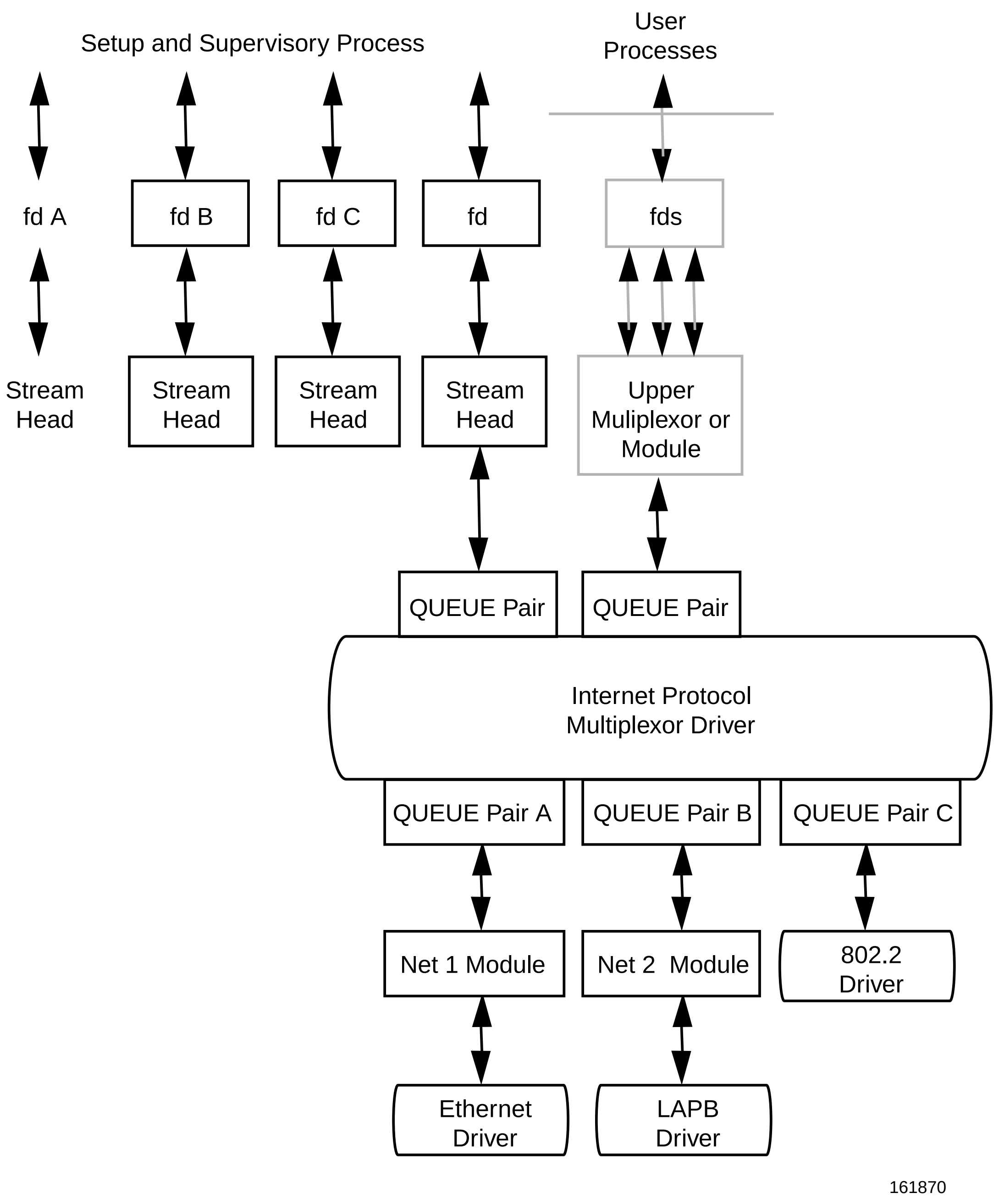


Figure 9-7. Internet Multiplexor after Connecting

Streams A, B, and C are opened by the process, and modules are pushed as needed. Two upper Streams are opened to the IP multiplexor. The rightmost Stream represents multiple Streams, each connected to a process using the network. The Stream second from the right provides a direct path to the multiplexor for supervisory functions. It is the control Stream, leading to a process that sets up and supervises this configuration. It is always directly connected to the IP driver. Although not shown, modules can be pushed on the control Stream.

After the Streams are opened, the supervisory process typically transfers routing information to the IP drivers (and any other multiplexors above the IP), and initializes the links. As each link becomes operational, its Stream is connected below the IP driver. If a more complex multiplexing configuration is required, the IP multiplexor Stream with all its connected links can be connected below another multiplexor driver.

Figure 9-7 shows that the file descriptors for the lower device driver Streams are left dangling. The primary purpose in creating these Streams is to provide parts for the multiplexor. Those not used for control and not required for error recovery (by reconnecting through an **`I_UNLINK ioctl`**) have no further function. These lower Streams can be closed to free the file descriptor without affecting the multiplexor.

Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This example resembles a typical device driver, with two differences: the device handling functions are performed by a separate driver, connected as a lower Stream, and the device information (that is, relevant user process) is contained in the input data rather than in an interrupt call.

Each upper Stream is created by **`open(2)`**. A single lower Stream is opened and then linked by the multiplexing facility. This lower Stream might connect to the tty driver. The implementation of this example is a foundation for an M-to-N multiplexor.

As in the loop-around driver (in “STREAMS Drivers”), flow control requires the use of standard and special code, since connectivity among the Streams is broken at the driver. Different approaches are used for flow control on the lower Stream, for messages coming upstream from the device driver, and on the upper Streams, for messages coming downstream from the user processes.

The multiplexor declarations are shown in Screen 9-2:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>

static int muxopen(queue_t *, dev_t *, int, int, cred_t *);
static int muxclose(queue_t *, int, cred_t *);
static int muxuwput(queue_t *, mblk_t *);
static int muxlwsrv(queue_t *);
static int muxlrput(queue_t *, mblk_t *);
static int muxuwsrv(queue_t *);

static struct module_info info = {0xaabb, “mux”, 0, INFPSZ, 512, 128};

static struct qinit urinit = {/* upper read */
    NULL, NULL, muxopen, muxclose, NULL, &info, NULL };

static struct qinit uwinit = {/* upper write */
    muxuwput, muxuwsrv, NULL, NULL, NULL, &info, NULL };

static struct qinit lrinit = {/* lower read */
    muxlrput, NULL, NULL, NULL, NULL, &info, NULL };

static struct qinit lwinit = {/* lower write */
    NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL };

struct streamtab muxinfo = {&urinit, &uwinit, &lrinit, &lwinit};

struct mux {
    queue_t  *qptr;   /* back pointer to read queue */
    lock_t   *lck;    /* lock to protect mux struct */
    int      flag;    /* used to coordinate muxlrput with muxclose*/
}

/* flag bits */
#define BUSY     0x1
#define CLOSING  0x2

extern struct    mux mux_mux[];
extern int       mux_cnt;

int muxdevflag = D_MP;

lkinfo_t     mux_lkinfo;
lock_t       *muxlck;
sv_t         *muxsv;
int          muxbot_ref;/* prevents unlinks while putnext in progress */
queue_t      *muxbot; /* linked lower queue */
int          muxerr;  /* set if error or hangup on lower stream */
```

Screen 9-2. Multiplexor Declarations

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write `qinit` structures. The multiplexing `qinit` structures replace those in each lower Stream head after the **`I_LINK`** has completed successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower queues. There must be an upper Stream write **`put`** procedure and lower Stream read **`put`** procedure. If the queue procedures of the opposite upper/lower queue are not needed, the queue can be skipped over, and the message put to the following queue.

In the example, the upper read-side procedures are not used. The lower Stream read queue **`put`** procedure transfers the message directly to the read queue upstream from the multiplexor. There is no lower write **put** procedure because the upper write **put** procedure directly feeds the lower write queue downstream from the multiplexor.

The driver uses a private data structure, `mux`. `mux_mux[dev]` points back to the opened upper read queue. This is used to route messages coming upstream from the driver to the appropriate upper queue. It is also used to find a free major/minor device for a `CLONE-OPEN` driver open case.

The upper queue open contains the canonical driver open code as shown in Screen 9-3:

```
void muxinit(void)
{
    register struct mux *mux;

    muxlck = LOCK_ALLOC(MUXHIER, plstr, &mux_lkinfo, KM_NOSLEEP);
    muxsv = SV_ALLOC(KM_NOSLEEP);
    for (mux = mux_mux; mux < &mux_mux[mux_cnt]; mux++)
            mux->lck = LOCK_ALLOC(MUXHIER, plstr, &mux_lkinfo, KM_NOSLEEP);
}

static int muxopen(queue_t *q, dev_t *devp, int flag, int sflag,
           cred_t *credp)
{
    struct mux *mux;
    dev_t device;
    pl_t pl;
    if (q->q_ptr)
        return(EBUSY);
    if (muxlck == NULL || muxsv == NULL)
        return(ENXIO);

    if (sflag == CLONEOPEN) {
        for (device = 0; device < mux_cnt; device++) {
    if (mux_mux[device].lck == NULL)
        continue;
            pl = LOCK(mux_mux[device].lck, plstr);
            if (mux_mux[device].qptr == NULL)
                break;
            /* Note that we break out of if statement */
            /* with the correct lock held */
            if (device >= mux_cnt)
            UNLOCK(&mux_mux[device].lck, pl);
                return(ENXIO);
        }
    }
    else {
        device = getminor(*devp);
        if (device < 0 || device >= mux_cnt)
            return(ENXIO);
    if (mux_mux[device].lck == NULL)
        return (EXNIO);
        pl = LOCK(mux_mux[device].lck, plstr);
        }
    }
    /*
     * Once we get here, the device is valid and we're holding its lock.
     */
    mux = &mux_mux[device];
    mux->qptr = q;
    mux->flag = 0;
    q->q_ptr = (char *) mux;
    WR(q)->q_ptr = (char *) mux;
    UNLOCK(mux->lck, pl);
    qprocson(q);
    return(0);
}
```

Screen 9-3. Canonical Driver Open Code

muxopen checks for a clone or ordinary open call. It initializes q_ptr to point at the mux_mux[] structure.

The core multiplexor processing is the following: downstream data written to an upper Stream is queued on the corresponding upper write message queue if the lower Stream is flow controlled. This allows flow control to propagate towards the Stream head for each upper Stream. A lower write **service** procedure, rather than a write **put** procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read-side, data coming up the lower Stream are passed to the lower read **put** procedure. The procedure routes the data to an upper Stream based on the first byte of the message. This byte holds the minor device number of an upper Stream. The **put** procedure handles flow control by testing the upper Stream at the first upper read queue beyond the driver. The **put** procedure treats the Stream component above the driver as the next queue.

Upper Write Put Procedure

muxuwput, the upper queue write **put** procedure, traps **ioctl**s, in particular **I_LINK** and **I_UNLINK**:

```
static int muxuwput(queue_t *q, mblk_t *mp)
{
    pl_t pl;
    struct mux *mux;

    mux = (struct mux *) q->q_ptr;
    switch (mp->b_datap->db_type) {
    case M_IOCTL: {
        struct iocblk *iocp;
        struct linkblk *linkp;

        /*
         * ioctl.  Only channel 0 can do ioctls.  Two calls are
         * recognized: LINK, and UNLINK
         */

        if (mux != mux_mux)
            goto iocnak;

        iocp = (struct iocblk *) mp->b_rptr;
        switch (iocp->ioc_cmd) {
        case I_LINK:
            /*
             * Link.  The data contains a linkblk structure
             * Remember the bottom queue in muxbot.
             */
            pl = LOCK(muxlck, plstr);
            if (muxbot != NULL) {
                UNLOCK(muxlck, pl);
                goto iocnak;
            }
            linkp = (struct lnkblk *) mp->b_cont->b_rptr;
            muxbot = linkp->l_qbot;
            muxerr = 0;
            muxbot_ref = 0;
            UNLOCK(muxlck, pl);
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
        case I_UNLINK:
            /*
             * Unlink.  The data contains a linkblk structure.
             * If muxbot is busy, fail unlink.
             */
            linkp = (struct linkblk *) mp->b_cont->b_rptr;
            pl = LOCK(muxlck, plstr);
            if (muxbot_ref) {
                mp->b_datap->db_type = M_IOCNAK;
                iocp->ioc_error = EAGAIN;
            } else {
                  muxbot = NULL;
```

Screen 9-4. Upper Write Put Procedure

```
                } mp->b_datap->db_type = M_IOCACK;
            UNLOCK(muxlck, pl);
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
        default:
        iocnak:
            /* fail ioctl */
            mp->b_datap->db_type = M_IOCNAK;
            qreply(q, mp);
        }
        break;
    }
    case M_FLUSH:
        if (*mp->b_rptr & FLUSHW)
            flushq(q, FLUSHDATA);
        if (*mp->b_rptr & FLUSHR) {
            *mp->b_rptr &= ~FLUSHW;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;
    case M_DATA:
        /*
         * Data.  If we have no bottom queue --> fail
         * Otherwise, queue the data and invoke the lower
         * service procedure.
         */
        pl = LOCK(muxlck, plstr);
        if (muxerr || muxbot == NULL) {
            UNLOCK(muxlck, pl);
            goto bad;
        }
        if (canputnext(muxbot)) {
            mblk_t *bp;
            if ((bp = allocb(1, BPRI_MED)) == NULL) {
                UNLOCK(muxlck, pl);
                putq(q, mp);
                bufcall(1, BPRI_MED, qenable, q);
                break;
            }
            muxbot_ref = 1;
            UNLOCK(muxlck, pl);
            *bp->b_wptr++ = (struct mux*) q->q_ptr - mux_mux;
            bp->b_cont = mp;
            putnext(muxbot, bp);
            pl = LOCK(muxlck, plstr);
            muxbot_ref = 0;
            UNLOCK(muxlck, pl);
        } else {
            UNLOCK(muxlck, pl);
            putq(q, mp);
        }
        break;
    default:
bad:
        /*
         * Send an error message upstream.
         */
        mp->b_datap->db_type = M_ERROR;
        mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
        *mp->b_wptr++ = EINVAL;
        qreply(q, mp);
    }
}
```

First, there is a check to enforce that the Stream associated with minor device 0 will be the single, controlling Stream. The **ioctl**s are only accepted on this Stream. As described previously, a controlling Stream is the one that issues the **I_LINK**. Having a single control Stream is a recommended practice. **I_LINK** and **I_UNLINK** include a `linkblk` structure containing:

`l_qtop`	The upper write queue from which the **ioctl** is coming. It should always equal `q`.
`l_qbot`	The new lower write queue. It is the former Stream head write queue and is important because it is where the multiplexor gets and puts its data.
`l_index`	A unique (system wide) identifier for the link. It can be used for routing or during selective unlinks. Since the example only supports a single link, `l_index` is not used.

For **I_LINK**, `l_qbot` is saved in `muxbot` and a positive acknowledgment is generated. From this point on, until an **I_UNLINK** occurs, data from upper queues will be routed through `muxbot`. Note that when an **I_LINK**, is received, the lower Stream has already been connected. This allows the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgment) in response to an **I_LINK** will cause the lower Stream to be disconnected.

The **I_UNLINK** handling code nulls out `muxbot` and generates a positive acknowledgment. A negative acknowledgment should not be returned to an **I_UNLINK**. The Stream head assures that the lower Stream is connected to a multiplexor before sending an **I_UNLINK** `M_IOCTL`.

muxuwput handles `M_FLUSH` messages as a normal driver would, except that there are no messages enqueued on the upper read queue, so there is no need to call **flushq** if `FLUSHR` is set.

`M_DATA` messages are not placed on the lower write message queue. They are queued on the upper write message queue. When flow control subsides on the lower Stream, the lower **service** procedure, **muxlwsrv**, is scheduled to start output. This is similar to starting output on a device driver.

Upper Write Service Procedure

Screen 9-5 shows the code for the upper multiplexor write **service** procedure:

```
static int muxuwsrv(queue_t *q)
{
    struct mux *muxp;
    mblk_t *mp;
    pl_t pl;
    muxp = (struct mux *) q->q_ptr;
    while (mp = getq(q)) {
        pl = LOCK(muxlck, plstr);
        if (!muxbot) {
            UNLOCK(muxlck, pl);
            flushq(q, FLUSHALL);
            return;
        }
        if (muxerr) {
            UNLOCK(muxlck, pl);
            flushq(q, FLUSHALL);
            return;
        }
        if (canputnext(muxbot)) {
            muxbot_ref = 1;
            UNLOCK(muxlck, pl);
            putnext(muxbot, mp);
            pl = LOCK(muxlck, plstr);
            muxbot_ref = 0;
            UNLOCK(muxlck, pl);
        } else {
            UNLOCK(muxlck, pl);
            putbq(q, mp);
            return(0);
        }
    }
}
```

Screen 9-5. Upper Write Service Procedure

As long as there is a Stream still linked under the multiplexor and there are no errors, the **`service`** procedure takes a message off the queue and sends it downstream, if flow control allows.

Lower Write Service Procedure

`muxlwsrv`, the lower (linked) queue write **`service`** procedure is scheduled as a result of flow control subsiding downstream (it is back-enabled).

```
static int muxlwsrv(queue_t *q)
{
    register int i;
    pl_t pl;

    for (i = 0; i < mux_cnt; i++) {
        pl = LOCK(mux_mux[i].lck, plstr);
        if (mux_mux[i].qptr)
            qenable(mux_mux[i].qptr);
        UNLOCK(mux_mux[i].lck, pl);
    }
}
```

Screen 9-6. Lower Write Service Procedure

`muxlwsrv` steps through all possible upper queues. If a queue is active and there are messages on the queue, then the upper write **`service`** procedure is enabled by **`qenable`**.

Lower Read Put Procedure

The lower (linked) queue read **`put`** procedure is shown in Screen 9-7:

```
static int muxlrput(queue_t *q, mblk_t *mp)
{
    queue_t *uq;
    mblk_t *b_cont;
    int device;
    register struct mux *mux;
    pl_t pl;
    pl = LOCK(muxlck, plstr);
    if (muxerr) {
        freemsg(mp);
        UNLOCK(muxlck, pl);
        return(0);
    }
    UNLOCK(muxlck, pl);
    switch (mp->b_datap->db_type) {
    case M_FLUSH:
        /*
         * Flush queues.  NOTE: sense of tests is reversed since
         * we are acting like a “stream head”
         */
        if (*mp->b_rptr & FLUSHW) {
            *mp->b_rptr &= ~FLUSHR;
            qreply(q, mp);
        } else
            freemsg(mp);
        break;
    case M_ERROR:
    case M_HANGUP:
        pl = LOCK(muxlck, plstr);
        muxerr = 1;
        UNLOCK(muxlck, pl);
        freemsg(mp);
        break;

    case M_DATA:
        /*
         * Route message.  First byte indicates device to send to.
         * No flow control.
         *
         * Extract and delete device number.  If the leading block is
         * now empty and more blocks follow, strip the leading block.
         */
        device = *mp->b_rptr++;

        /* Sanity check.  Device must be in range */
        if (device < 0 || device >= mux_cnt) {
            freemsg(mp);
            break;
        }

        /*
         * If upper streams is open and not backed up, send the
```

Screen 9-7. Lower Read Put Procedure

```
         * message there, otherwise discard it.
         */
        mux = &mux_mux[device];
        pl = LOCK(mux->lck, plstr);
        uq = mux->qptr;
        if (uq != NULL && canputnext(uq)) {
            mux->flag |= BUSY;
            UNLOCK(mux->lck, pl);
            putnext(uq, mp);
            pl = LOCK(mux->lck, plstr);
            mux->flag &= ~BUSY;
            if (mux->flag & CLOSING)
                SV_SIGNAL(muxsv, 0);
        } else
            freemsg(mp);
        UNLOCK(mux->lck, pl);
        break;
    default:
        freemsg(mp);
    }
}
```

`muxlrput` receives messages from the Stream linked below the multiplexor. Here, it needs to act as the Stream head of the lower stream. This means that during `M_FLUSH` handling, the sense of the tests are reversed. If `FLUSHW` is set, then `FLUSHR` is turned off and the message is sent back downstream. Otherwise, the message is freed. No flushing is necessary in this example because no messages are enqueued on the lower queues of the multiplexor.

`muxlrput` also handles `M_ERROR` and `M_HANGUP` messages. If one is received, it locks up the upper Streams by setting `muxerr`.

`M_DATA` messages are routed by looking at the first data byte of the message. This byte contains the minor device of the upper Stream. Several sanity checks are made to see if the device is in range and the upper Stream is open and not full.

This multiplexor does not support flow control on the read-side. It is merely a router. If everything checks out, the message is put to the proper upper queue. Otherwise, the message is discarded.

The upper Stream **`close`** routine simply clears the mux entry so this queue will no longer be found.

```
/*
 * Upper queue close
 */
static int muxclose(queue_t *q, int flag, cred_t *credp)
{
    register struct mux *mux;
    pl_t pl;

    mux = (struct mux *) q->q_ptr;
    qprocsoff(q);
    pl = LOCK(mux->lck, plstr);
    /*
     * coordinate with muxlwrput.  Use a global sync. variable since this
     * case is unlikely and not worth the overhead of having 1 per
     * minor.
     */
    while (mux->flag & BUSY) {
        mux->flag |= CLOSING;
        /* don't allow signals - this should be a short wait */
        SV_WAIT(muxsv, primed, mux->lck);
        pl = LOCK(mux->lck, plstr);
        mux->flag &= ~CLOSING;
    }
    mux->qptr = NULL;
    UNLOCK(mux->lck);
    q->q_ptr = NULL;
    WR(q)->q_ptr = NULL;
    return(0);
}
```

Screen 9-8. Clean Upper queue

Persistent Links

With **I_LINK** and **I_UNLINK ioctl**s, the file descriptor associated with the Stream above the multiplexor used to set up the lower multiplexor connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling Stream dismantles the whole multiplexing configuration. Some applications may not want to keep a process running merely to hold the multiplexor configuration together. Therefore, “free-standing” links below a multiplexor are needed. A persistent link is such a link. It is similar to a STREAMS multiplexor link, except that a process is not needed to hold the links together. After the multiplexor has been set up, the process may close all file descriptors and exit, and the multiplexor remains intact.

Two **ioctl**s, **I_PLINK** and **I_PUNLINK**, are used to create and remove persistent links that are associated with the Stream above the multiplexor. **close(2)** and **I_UNLINK** are not able to disconnect the persistent links.

The format of **I_PLINK** is

ioctl(*fd0*, **I_PLINK**, *fd1*)

The first file descriptor, *fd0*, must reference the Stream connected to the multiplexing driver and the second file descriptor, *fd1*, must reference the Stream to be connected below the multiplexor. The persistent link can be created in the following way:

```
upper_stream_fd = open(“/dev/mux”, O_RDWR);
lower_stream_fd = open(“/dev/driver”, O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in a file
 */
exit(0);
```

Figure 9-8 shows how **`open(2)`** establishes a Stream between the device and the Stream head.

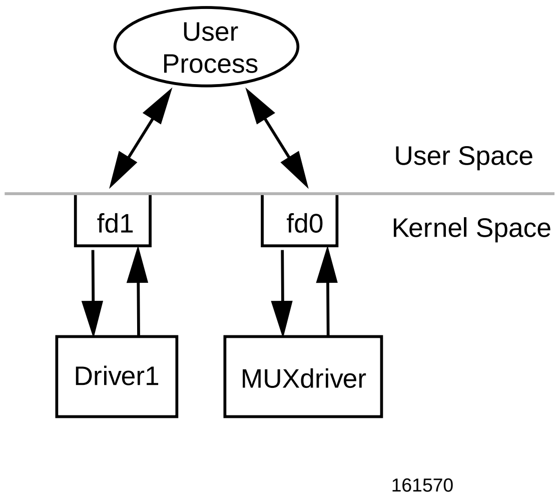


Figure 9-8. open() of MUXdriver and Driver1

The persistent link can still exist even if the file descriptor associated with the upper Stream to the multiplexing driver is closed. The **`I_PLINK`** **`ioctl`** returns an integer value, `muxid`, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may pass the `muxid` value to some other process to dismantle the link, if the dismantling is desired, or it can leave the `muxid` value in a file so that other processes may find it later. Figure 9-9 shows a multiplexor after **`I_PLINK`**.

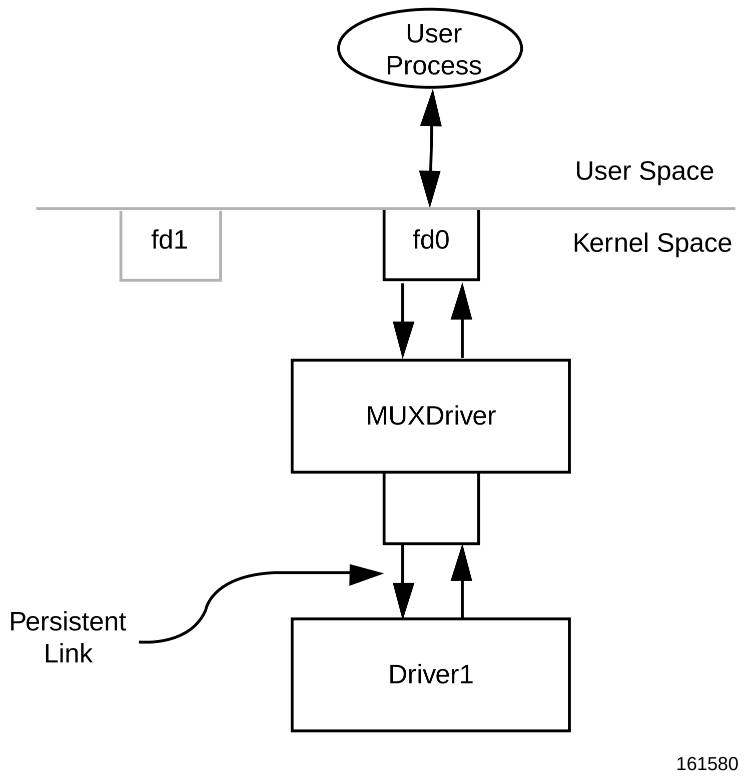


Figure 9-9. Multiplexor after I_PLINK

Several users can open the MUXdriver and send data to Driver1 since the persistent link to Driver1 remains intact, as shown in Figure 9-10.

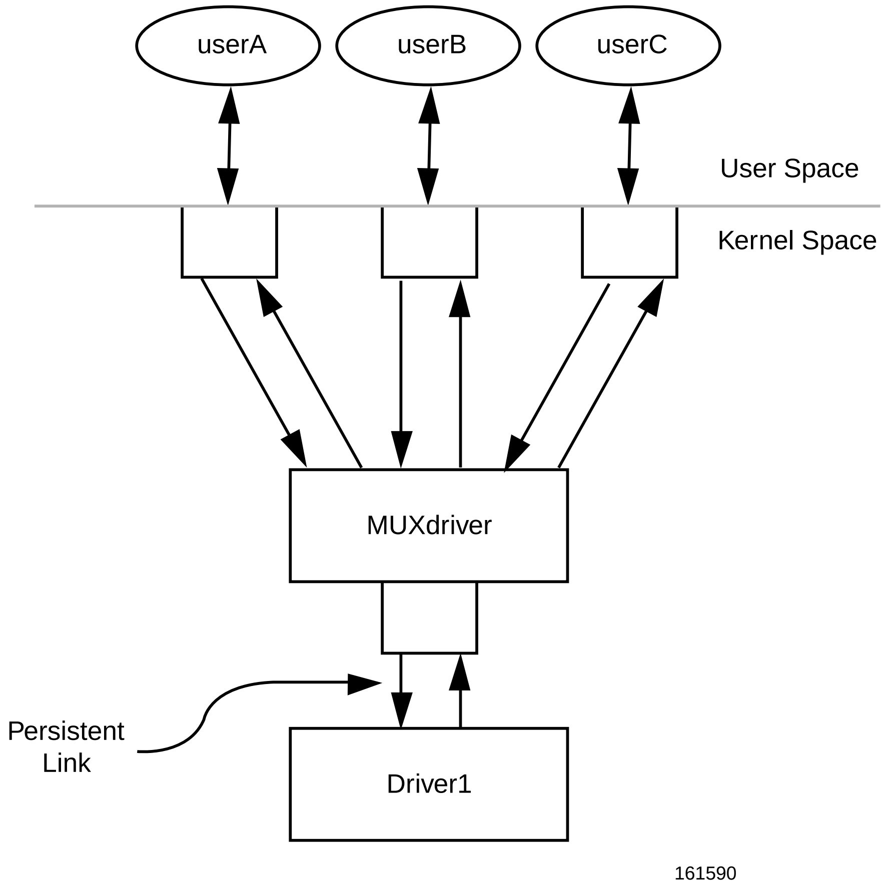


Figure 9-10. Other Users Opening a MUXdriver

The **`I_PUNLINK`** **`ioctl`** is used for dismantling the persistent link. Its format is

> **`ioctl`**`(`*fd0*`,` **`I_PUNLINK`**`,` *muxid*`)`

where the *fd0* is the file descriptor associated with Stream connected to the multiplexing driver from above. The *muxid* is returned by the **`I_PLINK ioctl`** for the Stream that was connected below the multiplexor. The **`I_PUNLINK`** removes the persistent link between the multiplexor referenced by the *fd0* and the Stream to the driver designated by the *muxid*. Each of the bottom persistent links can be disconnected individually. An **`I_PUNLINK ioctl`** with the *muxid* value of `MUXID_ALL` removes all persistent links below the multiplexing driver referenced by *fd0*.

Screen 9-9 dismantles the previously given configuration:

```
fd = open("/dev/mux", O_RDWR);
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

Screen 9-9. Retrieving the MUX ID from the File

The use of the **ioctl**s **I_PLINK** and **I_PUNLINK** should not be intermixed with **I_LINK** and **I_UNLINK**. Any attempt to unlink a regular link with **I_PUNLINK** or to unlink a persistent link with **I_UNLINK ioctl** causes the errno value of EINVAL to be returned.

Because multilevel multiplexing configurations are allowed in STREAMS, it is possible to have a situation where persistent links exist below a multiplexor whose Stream is connected to the above multiplexor by regular links. Closing the file descriptor associated with the controlling Stream removes the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexor whose Stream is connected to the above multiplexor with persistent links. In this example, the regular links are removed if the persistent link above is removed and no other references to the lower Streams exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by creating a persistent link of multiplexor 2 below multiplexor 1 and then closing the controlling file descriptor associated with the multiplexor 2 and reopening it again and then linking the multiplexor 1 below the multiplexor 2, but this is not allowed. The operating system prevents a multiplexor configuration from containing a cycle to ensure that messages cannot be routed infinitely, thus creating an infinite loop or overflowing the kernel stack.

Design Guidelines

The following lists general multiplexor design guidelines:

- The upper half of the multiplexor acts like the end of the upper Stream.
- The lower half of the multiplexor acts like the head of the lower Stream.
- *Service* procedures are used for flow control.
- Message routing is based on multiplexor specific criteria.
- When one Stream is being fed by many Streams, flow control may have to take place. Then all feeding Streams on the other end of the multiplexor have to be enabled when the flow control is relieved.

- When one Stream is feeding many Streams, flow control may also have to take place. Be careful not to starve other Streams when one becomes flow-controlled.
- Messages received on the lower half of a multiplexor that are not understood should be freed.
- Messages that should close a multiplexor are driver dependent.

10
Transport Provider Interface

10
Transport Provider Interface

Introduction

This chapter describes the STREAMS-based Transport Provider Interface (TPI). TPI is a service interface that maps to strategic levels of the Open Systems Interconnection (OSI) Reference Model. TPI supports the services of the Transport Layer for connection-mode and connectionless-mode services. One advantage to using TPI is its ability to hide implementation details of a particular service from the consumer of the service. This enables system programmers to develop software independent of the particular protocol that provides a specific service. This chapter focuses on TPI as it is defined within the STREAMS environment.

How TPI Works

TPI defines a message interface to a transport provider implemented under STREAMS. A user communicates to a transport provider via a full duplex path known as a stream. See Figure 10-1. This stream provides a mechanism in which messages may be passed to the transport provider from the transport user and vice versa.

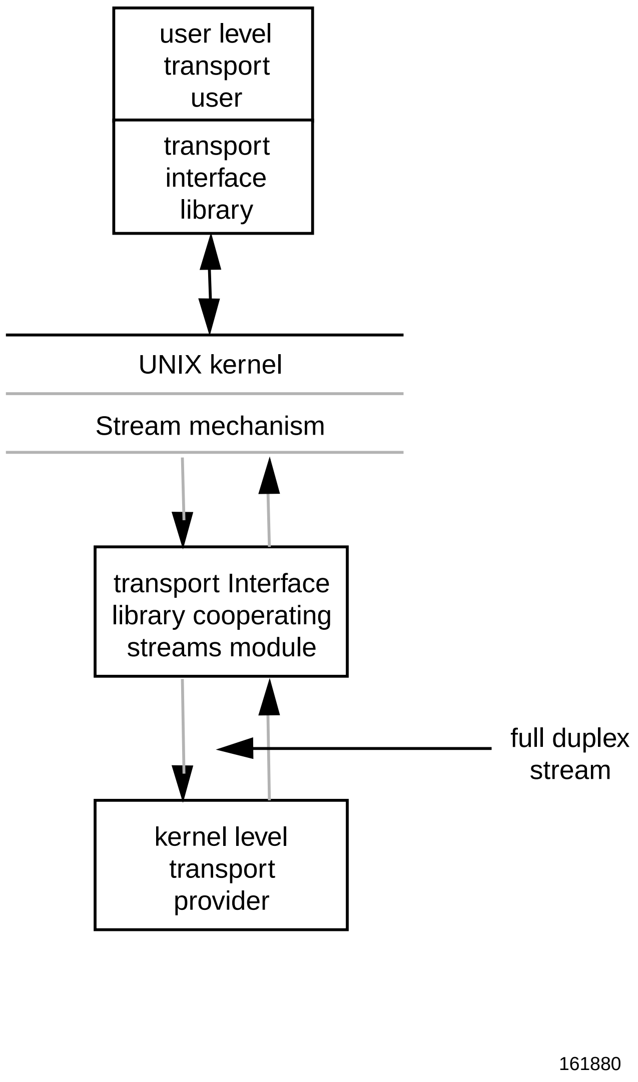

user level transport user
transport interface library
UNIX kernel
Stream mechanism
transport Interface library cooperating streams module
full duplex stream
kernel level transport provider
161880

Figure 10-1. Example of a Stream from a User to a Transport Provider

The STREAMS messages that are used to communicate transport service primitives between the transport user and the transport provider may have one of the following formats:

- An `M_PROTO` message block followed by zero or more `M_DATA` message blocks. The `M_PROTO` message block contains the type of transport service primitive and all the relevant arguments associated with the primitive. The `M_DATA` blocks contain transport user data associated with the transport service primitive.
- One `M_PCPROTO` message block containing the type of transport service primitive and all the relevant arguments associated with the primitive.

- One or more M_DATA message blocks containing transport user data.

Section 7 on-line manual pages describe the transport primitives which define both a connection-mode and connectionless-mode transport service. They include primitives that pertain to both transport modes.

For each type of transport service, two types of primitives exist:

- Primitives which originate from the transport user.

  The primitives which originate from the transport user make requests to the transport provider or respond to an event of the transport provider.

- Primitives which originate from the transport provider.

  The primitives which originate from the transport provider are either confirmations of a request or are indications to the transport user that an event has occurred.

"Mapping Of Transport Primitives to OSI" lists the primitive types along with the mapping of those primitives to the STREAMS message types and the transport primitives of the ISO IS 8072 and IS 8072/DAD transport service definitions. The format of these primitives and the rules governing the use of them are described in "Allowable Sequence of TPI Primitives."

Overview of Error Handling Capabilities

There are two error handling facilities available to the transport user: one to handle non-fatal errors and one to handle fatal errors.

Non-Fatal Errors

The non-fatal errors are those that a transport user can correct, and are reported in the form of an error acknowledgment to the appropriate primitive in error. Only those primitives which require acknowledgments may generate a non-fatal error acknowledgment. These acknowledgments always report a syntactical error in the specified primitive when the transport provider receives the primitive. The primitive descriptions above define those primitives and rules regarding the acknowledgment of them. These errors are reported to the transport user via the T_ERROR_ACK primitive, and give the transport user the option of reissuing the transport service primitive that caused the error. The T_ERROR_ACK primitive also indicates to the transport user that no action was taken by the transport provider on receipt of the primitive which caused the error.

These errors do not change the state of the transport service interface as seen by the transport user. The state of the interface after the issuance of a T_ERROR_ACK primitive should be the same as it was before the transport provider received the interface primitive that was in error.

The allowable errors that can be reported on the receipt of a transport initiated primitive are presented in the description of the appropriate primitives.

Fatal Errors

Fatal errors are those which can not be corrected by the transport user, or those errors which result in an uncorrectable error in the interface or in the transport provider.

The most common of these errors are listed under the appropriate primitives. The transport provider should issue fatal errors only if the transport user can not correct the condition which caused the error or if the transport provider has no means of reporting a transport user correctable error. If the transport provider detects an uncorrectable non-protocol error internal to the transport provider, the provider should issue a fatal error to the user.

Fatal errors are indicated to the transport user via the STREAMS message type `M_ERROR` with the PowerMAX OS system error `EPROTO`. This is the only type of error that the transport provider should use to indicate a fatal protocol error to the transport user. The message `M_ERROR` will result in the failure of all the operating system service routines on the stream. The only way for a user to recover from a fatal error is to ensure that all processes close the file associated with the stream. At that point, the user may reopen the file associated with the stream.

Transport Service Interface Sequence of Primitives

The allowable sequence of primitives are described in the state diagrams and tables in "Allowable Sequence of TPI Primitives" for both the connection-mode and connectionless-mode transport services. The following are rules regarding the maintenance of the state of the interface:

- It is the responsibility of the transport provider to keep record of the state of the interface as viewed by the transport user.
- The transport provider must never issue a primitive that places the interface out of state.
- The uninitialized state of a stream is the initial and final state, and it must be bound (see `T_BIND_REQ` primitive) before the transport provider may view it as an active stream.
- If the transport provider sends a `M_ERROR` upstream, it should also drop any further messages received on its write side of the stream.

The following rules apply only to the connection-mode transport services.

- A transport connection release procedure can be initiated at any time during the transport connection establishment or data transfer phase.
- The state tables for the connection-mode transport service providers include the management of the sequence numbering when a transport provider sends multiple `T_CONN_IND` requests without waiting for the response of the previously sent indication. It is the responsibility of the transport providers not to change state until all the indications have been responded to, therefore the provider should remain in the `T_WRES_CIND` state while there are any outstanding connect indications pending response. The provider should change state appropriately when all the connect indications have been responded to.

- The state of a transport service interface of a stream may only be transferred to another stream when it is indicated in a `T_CONN_RES` primitive. The following rules then apply to the cooperating streams:
  - The stream which is to accept the current state of the interface must be bound to an appropriate protocol address and must be in the idle state.
  - The user transferring the current state of a stream must have correct permissions for the use of the protocol address bound to the accepting stream.
  - The stream which transfers the state of the transport interface must be placed into an appropriate state after the completion of the transfer.

Precedence of TPI Primitives on a Stream

The following rules apply to the precedence of transport interface primitives with respect to their position on a stream:

NOTE

The stream queue which contains the transport user initiated primitives is referred to as the stream write queue. The stream queue which contains the transport provider initiated primitives is referred to as the stream read queue.

- The transport provider has responsibility for determining precedence on its stream write queue, as described in the rules in “Transport Primitive Precedence.” This section specifies the rules for precedence for both the connection-mode and connectionless-mode transport services.
- The transport user has responsibility for determining precedence on its stream read queue, as described in the rules in “Transport Primitive Precedence.” All primitives on the stream are assumed to be placed on the queue in the correct sequence as defined above.

The following rules apply only to the connection-mode transport services.

- There is no guarantee of delivery of user data once a `T_DISCON_REQ` primitive has been issued.

Rules for Flushing Queues

The following rules pertain to flushing the stream queues. No other flushes should be needed to keep the queues in the proper condition.

- The transport providers must be aware that they will receive `M_FLUSH` messages from upstream. These flush requests are issued to ensure that the providers receive certain messages and primitives. It is the responsibility of the providers to act appropriately as deemed necessary by the providers.
- The transport provider must send up a `M_FLUSH` message to flush both the read and write queues after receiving a successful `T_UNBIND_REQ` message and before issuing the `T_OK_ACK` primitive.

The following rules pertain only to the connection-mode transport providers.

- If the interface is in the `T_DATA_XFER`, `T_WIND_ORDREL` or `T_WACK_ORDREL` state, the transport provider must send up a `M_FLUSH` message to flush both the read and write queues before sending up a `T_DISCON_IND`.
- If the interface is in the `T_DATA_XFER`, `T_WIND_ORDREL` or `T_WACK_ORDREL` state, the transport provider must send up a `M_FLUSH` message to flush both the read and write queues after receiving a successful `T_DISCON_REQ` message and before issuing the `T_OK_ACK` primitive.

Mapping Of Transport Primitives to OSI

The following table maps those transport primitives as seen by the transport provider to the STREAMS message types used to realize the primitives and to the ISO IS 8072 and IS 8072/DAD1 transport service definition primitives.

Table 10-1. Mapping ISO IS 8072 and IS 8072/DAD1 to Transport Primitives

Transport Primitives	Stream Message Types	IS 8072 Transport Primitives
T_CONN_REQ	M_PROTO	T-CONNECT request
T_CONN_IND	M_PROTO	T-CONNECT indication
T_CONN_RES	M_PROTO	T-CONNECT response
T_CONN_CON	M_PROTO	T-CONNECT confirm
T_DATA_REQ	M_PROTO	T-DATA request
T_DATA_IND	M_PROTO	T-DATA indication
T_EXDATA_REQ	M_PROTO	T-EXPEDITED-DATA request
T_EXDATA_IND	M_PROTO	T-EXPEDITED-DATA indication
T_DISCON_REQ	M_PROTO	T-DISCONNECT request
T_DISCON_IND	M_PROTO	T-DISCONNECT indication
T_UNITDATA_REQ	M_PROTO	T-UNITDATA request
T_UNITDATA_IND	M_PROTO	T-UNITDATA indication
T_ORDREL_REQ	M_PROTO	not defined in ISO
T_ORDREL_IND	M_PROTO	not defined in ISO
T_BIND_REQ	M_PROTO	not defined in ISO
T_BIND_ACK	M_PCPROTO	not defined in ISO
T_UNBIND_REQ	M_PROTO	not defined in ISO
T_OK_ACK	M_PCPROTO	not defined in ISO
T_ERROR_ACK	M_PCPROTO	not defined in ISO
T_INFO_REQ	M_PCPROTO	not defined in ISO
T_INFO_ACK	M_PCPROTO	not defined in ISO
T_UDERR_IND	M_PROTO	not defined in ISO
T_OPTMGMT_REQ	M_PROTO	not defined in ISO
T_OPTMGMT_ACK	M_PCPROTO	not defined in ISO

Allowable Sequence of TPI Primitives

The following tables describe the possible events that may occur on the interface and the possible states as viewed by the transport user that the interface may enter due to an event.

The events map directly to the transport service interface primitives as described in "Introduction."

Table 10-2. Kernel Level Transport Interface States

Possible States			
State	Abbreviation	Description	Service Type
`sta_0`	`unbnd`	unbound	`T_COTS, T_COTS_ORD, T_CLTS`
`sta_1`	`w_ack b_req`	awaiting acknowledgment of `T_BIND_REQ`	`T_COTS, T_COTS_ORD, T_CLTS`
`sta_2`	`w_ack u_req`	awaiting acknowledgment of `T_UNBIND_REQ`	`T_COTS, T_COTS_ORD, T_CLTS`
`sta_3`	`idle`	idle - no connection	`T_COTS, T_COTS_ORD, T_CLTS`
`sta_4`	`w_ack op_req`	awaiting acknowledgment of `T_OPTMGMT_REQ`	`T_COTS, T_COTS_ORD, T_CLTS`
`sta_5`	`w_ack c_req`	awaiting acknowledgment of `T_CONN_REQ`	`T_COTS, T_COTS_ORD`
`sta_6`	`w_con c_req`	awaiting confirmation of `T_CONN_REQ`	`T_COTS, T_COTS_ORD`
`sta_7`	`w_res c_ind`	awaiting response of `T_CONN_IND`	`T_COTS, T_COTS_ORD`
`sta_8`	`w_ack c_res`	awaiting acknowledgment of `T_CONN_RES`	`T_COTS, T_COTS_ORD`
`sta_9`	`data_t`	data transfer	`T_COTS, T_COTS_ORD`
`sta_10`	`w_ind or_rel`	awaiting `T_ORDREL_IND`	`T_COTS_ORD`
`sta_11`	`w_req or_rel`	awaiting `T_ORDREL_REQ`	`T_COTS_ORD`
`sta_12`	`w_ack dreq6`	awaiting acknowledgment of `T_DISCON_REQ`	`T_COTS, T_COTS_ORD`
`sta_13`	`w_ack dreq7`	awaiting acknowledgment of `T_DISCON_REQ`	`T_COTS, T_COTS_ORD`
`sta_14`	`w_ack dreq9`	awaiting acknowledgment of `T_DISCON_REQ`	`T_COTS, T_COTS_ORD`
`sta_15`	`w_ack dreq10`	awaiting acknowledgment of `T_DISCON_REQ`	`T_COTS, T_COTS_ORD`
`sta_16`	`w_ack dreq11`	awaiting acknowledgment of `T_DISCON_REQ`	`T_COTS, T_COTS_ORD`

Variables and Outputs

The following describes the variables and outputs used in the state tables.

Table 10-3. State Table Variables

Variable	Description
`q`	queue pair pointer of current stream
`rq`	queue pair pointer of responding stream as described in the `T_CONN_RES` primitive
`outcnt`	counter for the number of outstanding connection indications not responded to by the transport user

Figure 10-2. State Table Outputs

Output	Description
[1]	`outcnt` = 0
[2]	`outcnt = outcnt + 1`
[3]	`outcnt = outcnt - 1`
[4]	pass connection to queue as indicated in the `T_CONN_RES` primitive

Outgoing Events

The following outgoing events are those which are initiated from the transport service user. They either make requests of the transport provider or respond to an event of the transport provider.

Table 10-4. Kernel Level Transport Interface Outgoing Events

Event	Description	Service Type
`bind_req`	bind request	`T_COTS, T_COTS_ORD, T_CLTS`
`unbind_req`	unbind request	`T_COTS, T_COTS_ORD, T_CLTS`
`optmgmt_req`	options mgmt request	`T_COTS, T_COTS_ORD, T_CLTS`

Table 10-4. Kernel Level Transport Interface Outgoing Events

Event	Description	Service Type
`conn_req`	connection request	`T_COTS`, `T_COTS_ORD`
`conn_res`	connection response	`T_COTS`, `T_COTS_ORD`
`discon_req`	disconnect request	`T_COTS`, `T_COTS_ORD`
`data_req`	data request	`T_COTS`, `T_COTS_ORD`
`exdata_req`	expedited data request	`T_COTS`, `T_COTS_ORD`
`ordrel_req`	orderly release request	`T_COTS_ORD`
`unitdata_req`	unitdata request	`T_CLTS`

Incoming Events

The following incoming events are those which are initiated from the transport provider. They are either confirmations of a request or are indications to the transport user that an event has occurred.

Table 10-5. Kernel Level Transport Interface Incoming Events

Event	Description	Service Type
`bind_ack`	bind acknowledgment	`T_COTS`, `T_COTS_ORD`, `T_CLTS`
`optmgmt_ack`	options management acknowledgment	`T_COTS`, `T_COTS_ORD`, `T_CLTS`
`error_ack`	error acknowledgment	`T_COTS`, `T_COTS_ORD`, `T_CLTS`

Table 10-5. Kernel Level Transport Interface Incoming Events

Event	Description	Service Type

`ok_ack1`	ok acknowledgment
	`outcnt == 0`
		`T_COTS,`
		`T_COTS_ORD,`
		`T_CLTS`

`ok_ack2`	ok acknowledgment
	`outcnt == 1,`
	`q == rq`
		`T_COTS,`
		`T_COTS_ORD,`

`ok_ack3`	ok acknowledgment
	`outcnt == 1,`
	`q != rq`
		`T_COTS,`
		`T_COTS_ORD,`

`ok_ack4`	ok acknowledgment
	`outcnt > 1`
		`T_COTS,`
		`T_COTS_ORD,`

`conn_ind`	connection indication	`T_COTS,`
		`T_COTS_ORD`

`conn_con`	connection confirmation	`T_COTS,`
		`T_COTS_ORD`

`data_ind`	data indication	`T_COTS,`
		`T_COTS_ORD`

`exdata_ind`	expedited data indication	`T_COTS,`
		`T_COTS_ORD`

`ordrel_ind`	orderly release indication	`T_COTS_ORD`

`discon_ind1`	disconnect indication
	`outcnt == 0`
		`T_COTS,`
		`T_COTS_ORD`

`discon_ind2`	disconnect indication
	`outcnt == 1`
		`T_COTS,`
		`T_COTS_ORD`

`discon_ind3`	disconnect indication
	`outcnt > 1`
		`T_COTS,`
		`T_COTS_ORD`

`pass_conn`	pass connection	`T_COTS,`
		`T_COTS_ORD`

`unitdata_ind`	unitdata indication	`T_CLTS`
`uderror_ind`	unitdata error indication	`T_CLTS`

Transport Service State Tables

The tables shown in Figure 10-3, Figure 10-4 and Figure 10-5 describe the possible next states the interface may enter given a current state and event.

The contents of each box represent the next state given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action. The transport provider must take the specific actions in the order specified in the state table.

state / event	sta_0 unbnd	sta_1 w_ack b_req	sta_2 w_ack u_req	sta_3 idle	sta_4 w_ack op_req
bind_req	sta_1				
unbind_req				sta_2	
optmgmt_req				sta_4	
bind_ack		sta_3 [1]			
optmgmt_ack					sta_3
error_ack		sta_0	sta_3		sta_3
ok_ack1			sta_0		

161890

Figure 10-3. Initialization State Table

state event	sta_3 idle	sta_5 w_ack c_req	sta_6 w_con c_req	sta_7 w_res c_ind	sta_8 w_ack c_res	sta_9 data_t	sta_10 w_ind or rel **	sta_11 w_req or rel **	sta_12 w_ack dreq6	sta_13 w_ack dreq7	sta_14 w_ack dreq9	sta_15 w_ack dreq10	sta_16 w_ack dreq11
conn_req	sta_5												
conn_res				sta_8									
discon_req			sta_12	sta_13		sta_14	sta_15	sta_16					
data_req						sta_9		sta_11					
exdata_req						sta_9		sta_11					
**ordrel_req						sta_10		sta_3					
conn_ind	sta_7 [2]			sta_7 [2]									
conn_con			sta_9										
data_ind						sta_9	sta_10						
exdata_ind						sta_9	sta_10						
**ordrel_ind						sta_11	sta_3						
discon_ind1			sta_3			sta_3	sta_3	sta_3					
discon_ind2				sta_3 [3]									
discon_ind3				sta_7 [3]									
error_ack		sta_3			sta_7				sta_6	sta_7	sta_9	sta_10	sta_11
ok_ack1		sta_6							sta_3		sta_3	sta_3	sta_3
ok_ack2					sta_9 [3]					sta_3 [3]			
ok_ack3					sta_3 [3][4]					sta_3 [3]			
ok_ack4					sta_7 [3][4]					sta_7 [3]			
pass_conn	sta_9												

**Only supported if service is type T_COTS_ORD

161900

Figure 10-4. Data-Transfer State Table for Connection Oriented Service

state / event	sta_3 idle
unitdata_req	sta_3
unitdata_ind	sta_3
uderror_ind	sta_3

161910

Figure 10-5. Data-Transfer State Table for Connectionless Service

Transport Primitive Precedence

The stream queue which contains the transport user initiated primitives is referred to as the stream write queue. The stream queue which contains the transport provider initiated primitives is referred to as the stream read queue. Figure 10-6 shows the stream write queue precedence table. Figure 10-7 shows the steam read queue precedence table.

X

Y

	t_conn_req	t_conn_res	t_discon_req	t_data_req	t_exdata_req	t_bind_req	t_unbind_req	t_info_req	t_unitdata_req	t_optmgmt_req	t_ordrel_req
t_conn_req			4								
t_conn_res			3								
t_discon_req											
t_data_req			5	1	2						1
t_exdata_req			5	1	1						1
t_bind_req											
t_unbind_req											
t_info_req											
t_unitdata_req									1		
t_optmgmt_req											
t_ordrel_req			5								

161920

Key

blank: not applicable / queue should be empty

1 : X has no precedence over Y
2 : X has precedence over Y
3 : X has precedence over Y
and Y must be removed
4 : X has precedence over Y
and both X and Y must be removed
5 : X may have precedence over Y
(choice of user) and if X does, then
it is the same as 3

Figure 10-6. Stream Write Queue Precedence Table

X

Y

	t_conn_ind	t_conn_con	t_discon_ind	t_data_ind	t_exdata_ind	t_info_ack	t_bind_ack	t_error_ack	t_ok_ack	t_unitdata_ind	t_uderror_ind	t_optmgmt_ack	t_ordrel_ack
t_conn_ind			4										
t_conn_con			3	1	1								
t_discon_ind	1							2	2				
t_data_ind			5	1	2				1				1
t_exdata_ind			5	1	1				1				1
t_info_ack													
t_bind_ack	1												
t_error_ack	1	1	1	1	1								
t_ok_ack	1	1	1	1	1								
t_unitdata_ind								2		1	2	2	
t_uderror_ind								1		1	1	1	
t_optmgmt_ack	1									1	1		
t_ordrel_ack	1		5					2	2				

161930

Key

blank: not applicable / queue should be empty

1 : X has no precedence over Y

2 : X has precedence over Y

3 : X has precedence over Y
and Y must be removed

4 : X has precedence over Y
and both X and Y must be removed

5 : X may have precedence over Y
(choice of user) and if X does, then
it is the same as 3

Figure 10-7. Stream Read Queue Precedence Table

11
Data Link Provider Interface

11
Data Link Provider Interface

Introduction

This chapter presents some guidelines for the development of network device drivers which conform to the Data Link Provider Interface (DLPI). It contains the following:

- An overview of the DLPI model
- A description of the different protocols used in a LAN together with the requirements for network management
- A description of the PowerMAX OS driver network environment and the capabilities a driver should provide in order to operate there
- Details of the framework for the design of these drivers
- A model of the OSI Data Link Layer
- A listing of DLPI services and primitives

DLPI specifies a Streams based interface between the data link layer (data link service provider) and the network layer (Data Link Service user) of the OSI reference model. It enables a Data Link Service (DLS) user to access any DLPI conformance provider without special knowledge of the provider's protocol.

NOTE

A DLS user is the user-level application or user-level or kernel-level protocol that accesses the services of the data link layer.

This implies that the DLPI conformance providers can be freely substituted with minimal changes to the implementation of the DLS user.

However, there may be many different DLS users such as CLNS, IP and IPX which may use different framing formats and have other unique requirements. These formats divide the driver into hardware dependent and hardware independent sections. The hardware independent code provides the generic part that does not need to change from one driver to the next and deals mostly with the specifics of the DLPI. Additionally it provides support for a number of different potential DLS users, specifically TCP/IP, Netware and OSI. The hardware independent code is available to developers of drivers and allows them to concern themselves only with the hardware specifics of their particular driver.

How DLPI Works

DLPI is a STREAMS-based implementation of the service specification of the IEEE ISO DIS 8886 and ISO 8802 Logical Link Control (802.2) standard. The IEEE 802 standards divide the data link layer of the OSI reference model into two sub-layers:

- A media independent upper portion called the Logical Link Control (LLC) layer, described in standard 802.2
- A media dependent lower layer called the Media Access Control (MAC) layer, described in standards 802.3 for Carrier Source Multiple Access with Collision Detection (CSMA/CD), 802.4 for token bus and 802.5 for Token Ring protocols.

Figure 11-1 shows the IEEE 802 model.

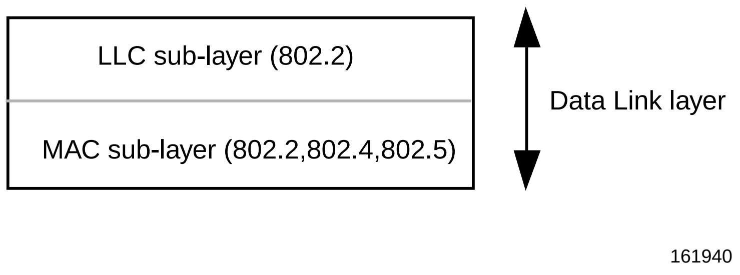


Figure 11-1. The IEEE 802 Model

The major components of the DLPI model are shown in Figure 11-2. The driver is referred to as the DLS provider and a protocol module which is layered on top is referred to as a DLS user.

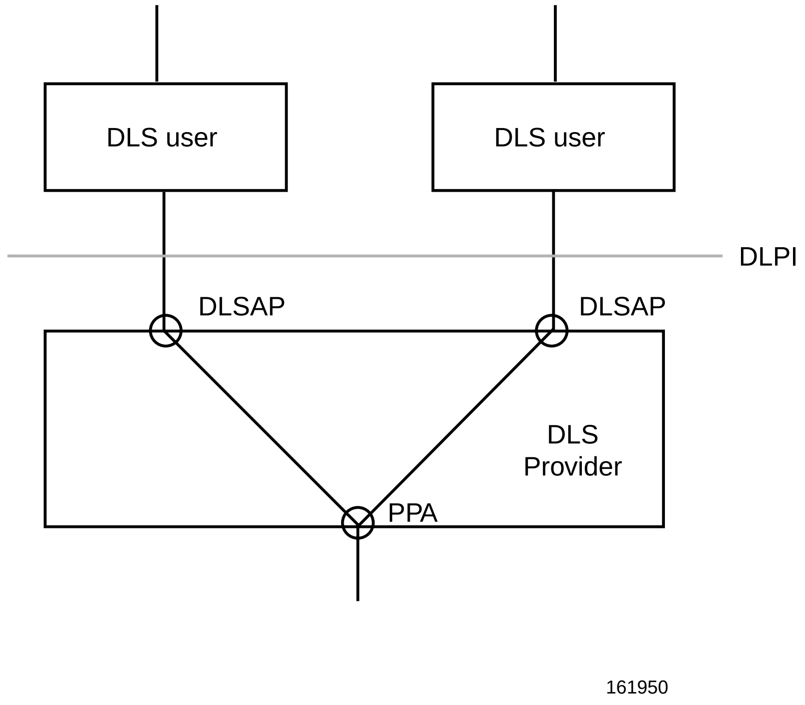


Figure 11-2. The DLPI Model

A DLS user accesses the services of a provider at a Service Access Point (SAP) using DLPI primitives in the form of STREAMS messages. It can be seen that a DLS provider must potentially route data from a single physical medium to multiple DLS users, for example IP and IPX. Individual DLS users identify themselves to the DLS provider using a SAP address which is conveyed to the provider using a primitive operation which binds a DLSAP with a STREAM.

DLPI supports three modes of communication to deal with the wide variety of data link providers and upper layer requirements:

- connection-oriented
- unacknowledged connectionless
- acknowledged connectionless

The framework described in this chapter is for the unacknowledged connectionless mode since this is the form used by most LAN protocols.

Additionally, DLPI permits two *styles* to distinguish between Physical Points of Attachment (PPAs). Style one providers assigns PPAs based on the major and minor number of the device opened. Typically, there will be one major number per board and DLS users will be assigned a minor number when opening the stream using the STREAMS **clone-open** feature. Style two providers enable the DLS user to specify the particular PPA after an open by using an attach primitive. The framework described in this chapter uses a style one provider.

Hardware/Software Environment

Developers of Streams drivers should refer to the “STREAMS Modules” and “STREAMS Drivers” chapters in this guide, and the *Device Driver Reference.* The *Device Driver Reference* also contains information on dynamic loadable kernel modules. Packaging and installation guidelines can be found in the *Device Driver Programming.*

The LAN Environment

This section gives an overview of the different LAN environments, including media, protocol suites and network management. It describes the numerous standards documents relating to networking and highlights the more important information. The references themselves should be consulted for more detailed information. For further background reading see *Computer Networks*, Tannenbaum, Andrew S., 2nd ed. 1988 Prentice-Hall.

Media Access Methods

The two media access methods are CSMA/CD and Token Ring.

CSMA/CD

CSMA/CD LANs are governed by two standards, the Ethernet 2.0 specification and the IEEE 802.3 standard. See the *Ethernet: Data Link Layer and Physical Layer Specifications, Digital, Intel and Xerox, 1982* and the *ANSI/IEEE Std 802.3 ISO 8802/2 CSMA/CD Access Method* The 802.3 standard covers a whole range of speeds from 1 to 20 Mbps on a variety of media. However, the two most common configurations are *thick* Ethernet and *thin* Ethernet cables using 10 Mbps baseband transmission.

The 802.3 standard is derived from the Ethernet specification; however it does differ slightly in the framing used.

Figure 11-3 shows the Ethernet frame format. Figure 11-4 shows the 802.3 frame format.

Preamble	Destination address	Source address	Packet type	Data	CRC
8	6	6	2	0 - 1500	4

161960

Figure 11-3. Ethernet Frame Format

Preamble	Start of frame	Destination address	Source address	Length	Data	CRC
7	1	2 or 6	2 or 6	2	0 - 1500	4

161970

Figure 11-4. 802.3 Frame Format

The only difference between an Ethernet and 802.3 frame that impacts a DLPI driver is that the packet type indicator in an Ethernet frame is used as a length field in 802.3 frames. Packet types are used to indicate the protocol using the Ethernet frame. Fortunately, most protocol type values are greater than 1500, the maximum length of data, which enables a driver to distinguish between the two types of framing. The exceptions to this rule are the Xerox PUP protocols which have type fields of `0x200` and `0x201`.

Token Ring

The two main Token Ring protocols are IEEE 802.5 and FDDI. See the *ANSI/IEEE Std 802.5 ISO 8802/5 Token Ring Access Method*, the *ISO FDDI Physical Layer Protocol, ISO 9314-1, 1989*, the *ISO FDDI Media Access Control, ISO 9314-2, 1987* and the *ISO FDDI Physical Layer Medium Dependant, ISO 9314-3, 1989* for more information regarding Token Ring protocols.

The 802.5 frame format is shown in Figure 11-5.

Frame control	Access control	Delim	Source address	Dest. address	Data	CRC	Delim	Frame status
1	1	1	2 or 6	2 or 6	unlimited	4	1	1

161980

Figure 11-5. 802.5 Frame Format

There is no explicit maximum length for a frame, however there is an implicit maximum since the entire frame must be transmitted within the token holding time.

Logical Link Layer Protocols

Messages for the IEEE 802.2 LLC protocol have two formats depending on whether the SNAP extension is used. These formats are shown in Figure 11-6.

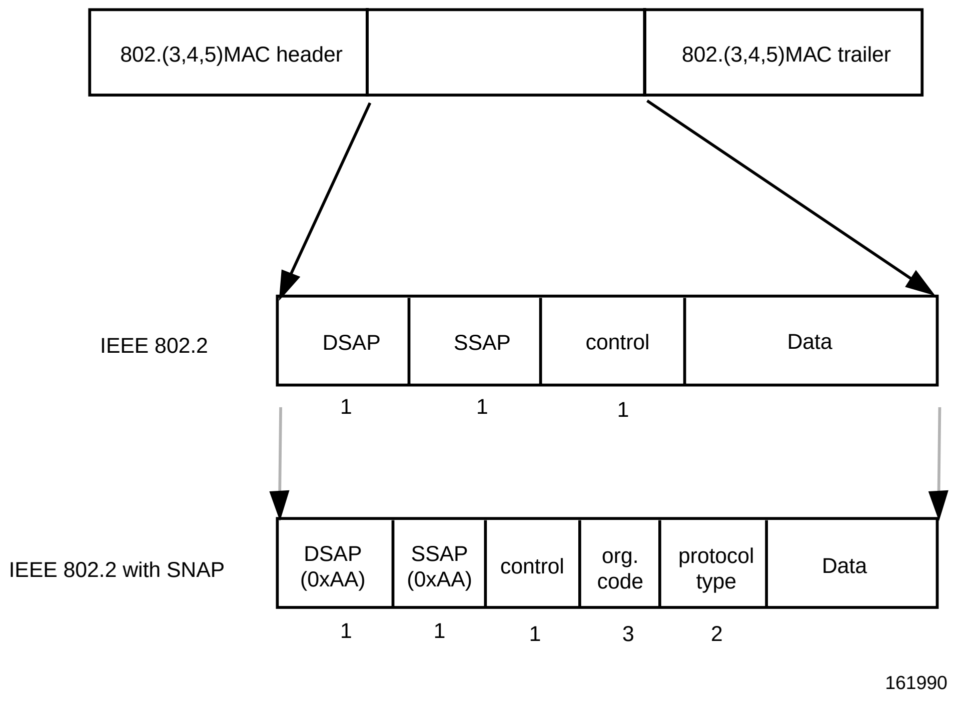


Figure 11-6. 802.2 Message Format

Protocol Suites

The framework described in this document provides support for TCP/IP, NetWare and OSI protocols and also allows for the possibility of them all being used at the same time on the same host. It must therefore multiplex all incoming packets and route them to the appropriate protocol stack. This is done by examining the frame format and LLC headers and matching them with information provided by the upper layer protocols at bind time.

OSI

The OSI protocols require a network driver to provide support for the IEEE 802.2 LLC protocol along with an appropriate IEEE MAC layer protocol such as 802.3 for CSMA/CD and 802.5 for Token Ring. See *ANSI/IEEE Std 802.2 ISO 8802/2 Logical Link Control.*

TCP/IP over CSMA/CD

The TCP/IP protocol suite can use a CSMA/CD LAN in one of two ways. The first is using Ethernet V2.0 frames, as described in *RFC-894, A Standard for the Transmission of IP Datagrams over Ethernet.*

In this example the type field of the Ethernet frame is set to indicate the protocol using the frame. For example, for IP it would be set to `0x0800`. The second method is using 802.3 format frames. The latter is achieved by using the SNAP LSAP in the 802.2 header. This is described in *RFC-1042, A Standard for the Transmission of IP Datagrams over IEEE 802 Networks.*

In this example the DSAP and SSAP are set to 0xAA to indicate that the SNAP format is being used and the last two bytes of the SNAP header are set to the same protocol type identifier as used in Ethernet frames.

TCP/IP over Token Ring

The SNAP frame format will be used to transmit IP datagrams over a token ring network. While the Org-code will be 0 for both IP and ARP packets, the EtherType fields will be 0x800 and 0x806 respectively.

NetWare over CSMA/CD

NetWare client and Portable NetWare stacks can communicate in any of four different frame formats:

- Ethernet V-2.0 with the protocol ID field set to `0x8137`
- IEEE-802.3 with no LLC header. This is the predominant frame format among NetWare LANs. Any received frame whose length/type field has a value of less than `0x600` is automatically handed over to the IPX Transport/Network driver. The user data part of the frame is not encapsulated in a IEEE-802.2 frame.
- IEEE 802.3 frame containing a IEEE-802.2 LPDU with SAP values of 0xE0
- SNAP frames with `0xAA` SAPs in the LLC header and the type set to `0x8137`

NetWare over Token Ring

IEEE-802.2 or SNAP frame formats can be used to send IPX packets. The actual frame format will depend on the address that will be used with the bind request primitive. An address values of `0xFE` will result in all IPX packets to be transmitted as IEEE-802.2 frames. However, an address of `0x8137` will result in SNAP frames as the frame type of choice.

Network Management Support

A network interface driver will probably need to provide some support for network management especially in the form of statistics. There are two standards governing network management:

- The SNMP MIB as defined in *RFC-1213, Management Information Base for Network Management of TCP/IP-based Internet: MIB-II,*
- The OSI network management framework, as defined in *ISO/IEC DIS 10165, Information Processing Systems - Open Systems Interconnection - Structure of Management Information.*

Additionally, there are the `ifstats` statistics which are required by some TCP/IP implementations.

Broadcast and Multicast Support

Broadcast and multicast support are required by various protocols. For example OSI CLNS uses certain multicast addresses to identify all end systems and all intermediate systems for routing purposes.

Promiscuous Mode

A promiscuous mode SAP is one which receives all packets on the network whether addressed to it or not. This is typically used by applications which monitor network traffic. This must not interfere with the normal running of other protocols on the host. Therefore, in the framework described in “The DLPI Network Driver Framework,” packets intended for the local host are duplicated and sent to the promiscuous SAP as well as the real destination SAP. Users can query and set the state of promiscuous mode by using the **`DLIOCGPROMISC`** and **`DLIOCSPROMISC`** streams messages respectively. These streams messages are documented in **`ioctl(3d)`**

The DLPI Network Driver Framework

The driver software is logically divided into two halves:

- The hardware independent layer handles the DLPI primitives from the DLS user.
- The hardware dependent layer interfaces with the hardware controller.

Figure 11-7 shows the structure of the driver. Data is passed between the Hardware Independent Layer and the DLS user in the form of messages. A message is a set of data structures used to pass data, status and control information.

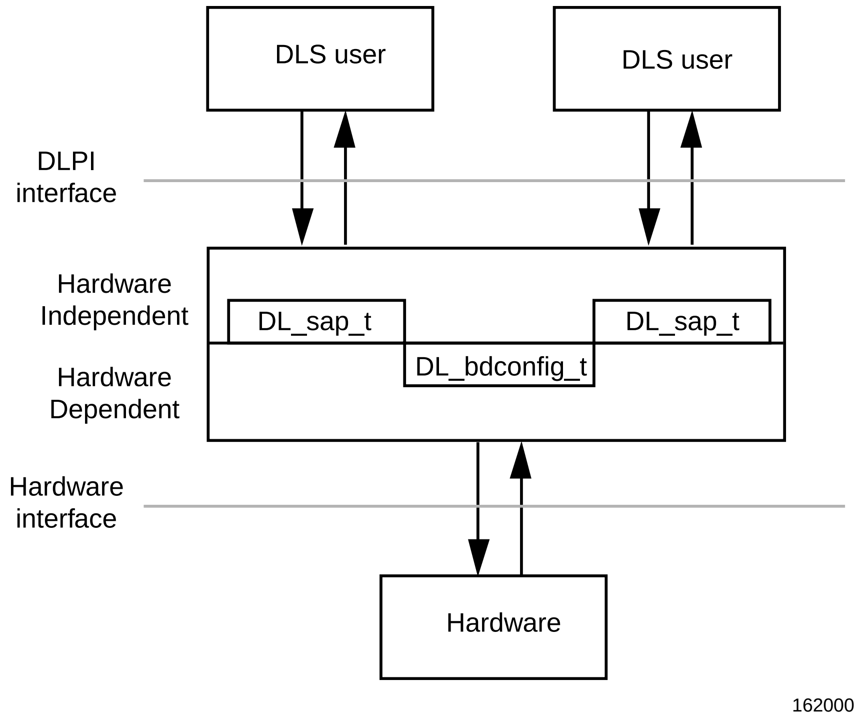


Figure 11-7. Structure of the Driver

The driver described here provides only the unacknowledged connectionless services. The DLPI primitives for connection mode data transfer are not supported by this provider. In addition to data transfer, the DLS user communicates to the Hardware Independent Layer using an **`ioctl`** interface for getting status messages and controlling the card functionality. See “Hardware/Software Environment.”

The Hardware Independent Layer invokes appropriate Hardware Dependent Layer functions during initialization and frame transmission. The Hardware Dependent Layer communicates with the Hardware Independent Layer by inserting received frames directly into the read queue of the Hardware Independent Layer. In addition, the two layers share some variables for synchronization and flow control purposes.

The Hardware Dependent Layer is responsible for card specific initialization functions, frame transmission and reception. It also keeps track of error information and translates some **`ioctl`**s to controller commands. See “The Hardware Dependent Layer” for more information.

The DLS provider is configured as a STREAMS driver. A DLS user accesses the provider using **`open(2)`** to establish a Stream to the driver. Thereafter, the user and the provider communicate by using the DLPI primitives.

After an open, the process must identify itself to the provider by binding a SAP to the Stream with a `DL_BIND_REQ` primitive. This allows the provider to determine the destination of received frames. A privileged process (that is, one with `uid 0`) may set its SAP to be “promiscuous” so that it can receive all incoming frames.

Major Data Structures

The DLPI data structures and the associated defines are present in **`/usr/include/sys/dlpi.h`**. Other major data structures needed in the driver are those for SAPs, board configuration structure, Ethernet statistics and the MIB. These are described here. Hardware specific data structures are also needed in the driver, but these are outside the scope of this chapter.

An Ethernet driver associates every installed board/adapter with an instance of a configuration structure `DL_bdconfig_t`. A configuration structure describes the characteristics of the board, contains information needed to operate the board and also holds adapter specific statistics maintained by the driver. Furthermore, each instance of a `DL_bdconfig_t` structure is shared by the hardware dependent and independent parts of the driver and used to pass information between them. A configuration structure is composed of a number of standard fields often used by all the drivers. In addition, fields are available for optional use by the individual drivers.

`major`	The major number of the device that identifies a particular board.
`io_start`	The start of I/O base address.
`xio_start`	The start of extended I/O base address.
`max_saps`	The number of service access points associated with the board
`bd_dltype`	Device type (Ethernet, FDDI, and so on)
`bd_number`	The board number in a multiboard setup.
`flags`	A bitmask to identify the operational status of the board.
`tx_next`	The next SAP to be serviced.
`timer_id`	Watchdog timer id.
`timer_val`	Watchdog timer value.
`eaddr`	The physical address of the board.
`ttl_valid_sap`	Total number of valid SAPs.
`sap_ptr`	Pointer to a table of service access points associated with the board.
`promisc_cnt`	The number of promiscuous SAPs currently associated with the board.
`multicast_cnt`	The total number of multicast address currently being used.
`valid_sap`	A pointer to a link list of valid SAPs that are in use.
`mib`	A list of statistics that are currently being maintained by the board.
`ifstats`	Pointer to the interface statistics structure.

All fields in the board structure may be initialized by the driver-specific **`init`** routines. For some drivers, the `major`, `bd_number` and `max_saps` fields are initialized in the **`space.c`** files that are a part of every driver package. The `sap_ptr` is initialized to NULL by the **`init`** routines. The `promisc_cnt` fields are initialized by the **`init`** routines and updated by the **`DLpromisc_on`** and **`DLpromisc_off`** routines. The `multicast_cnt` fields are initialized by the **`init`** routines and updated by the **`DLadd_multicast`** and **`DLdel_multicast`** routines.

The `flags` element can take the following values:

`BOARD_PRESENT`	The board specific `init` routines will turn on the `BOARD_PRESENT` bit after a successful initialization and reset sequence.
`BOARD_DISABLED`	An unsuccessful initialization or a reset sequence will result in the hardware dependent part of the driver turning on the `BOARD_DISABLED` bit. In addition, this bit could be turned on if the driver recognizes a malfunctioning board.
`TX_BUSY`	The bit indicates a temporary lack of resources (e.g., buffers needed to transmit a packet).
`TX_QUEUED`	This bit indicates packets waiting to be transmitted over the network.

Each SAP is identified by a set of standard parameters that describe both the type of the SAP and its operational characteristics. Each instance of a SAP is associated with a sap structure (`DL_sap_t`).

`state`	Identifies the current state of the SAP as defined by DLPI. Must be initialized to `DL_UNBOUND` by the driver specific `init` routines.
`sap_addr`	An unique identifier for the SAP.
`flags`	Defines the operational characteristics of the SAP.
`read_q`	The read side STREAMS queue associated with the SAP.
`write_q`	The write side STREAMS queue associated with the SAP.
`mac_type`	One of `DL_ETHER`, `DL_CSMACD,` or `DL_FDDI` depending on the type of the SAP.
`service_mode`	`DL_CLDLS`, Connection Less Data Link Service.
`provider_style`	`DL_STYLE1`.
`bd`	Back pointer that points to the controlling board.
`next_sap`	Points to next SAP in the list of valid SAPs.
`max_spdu`	Maximum amount of user data that can be transmitted in every frame and this is a function of the type of the SAP.

`min_spdu`	Minimum amount of user data that can be transmitted in every frame and this is a function of the type of the SAP.
`sap_sv`	SAP synchronization variable

The `flags` field of the SAP structure can assume the following values:

`RAWCSMACD`	A SAP through which only 802.3 frames are sent and received.
`SNAPCSMACD`	A SAP that sends and receives SNAP format frames.
`PROMISCUOUS`	A SAP that receives a copy of all the inbound frames irrespective of the destination SAP.
`SEND_LOCAL_TO_NET`	Indicates that a copy of all the loopback frames should also be sent over the network.
`PRIVILEGED`	Need super-user permission to operate the SAP.

A number of statistical counters are maintained as a part of the configuration structure. Counters are updated both by the hardware independent and dependent parts of the driver. A brief description of each counter is provided. A user can retrieve the current values of the counters using the appropriate **`ioctl`**s.

`ifInOctets`	The total number of bytes received from a given board.
`ifOutOctets`	The total number of bytes sent from a given board.
`ifOutUcastPkts`	number of unicastpackets sent out.
`ifOutNUcastPkts`	number of broadcast and multicast packets sent out.
`ifInDiscards`	number of valid packets received but dropped.
`ifInUcastPkts`	number of unicast packets received.
`ifInNUcastPkts`	number of broadcast and multicast packets received.
`ifInErrors`	number of packets received with errors.
`ifUnknownProtos`	number of packets received and dropped because of an invalid destination SAP.
`ifOutQlen`	number of packets queued up for transmission.
`ifOutErrors`	number of packets transmitted with errors.
`etherAlignErrors`	number of frames alignment errors.
`etherCRCerrors`	number of frames with CRC errors.
`etherMissedPkts`	number of missed packets.
`etherOverrunErrors`	number of DMA Overrun errors.
`etherUnderrunErrors`	number of DMA Underrun errors.

`etherCollisions`	number of collisions.
`etherAbortErrors`	number of Transmits aborted at interface.
`etherReadqFull`	Number of times read queues were flow controlled.
`etherRcvResources`	Number of resource allocation failures (e.g.: Buffers).

The Hardware Independent Layer

The Hardware Independent Layer processes all calls made by the DLS user and the hardware dependent layer. It has the following routines:

`DLopen()`	open routine
`DLclose()`	close routine
`DLwput()`	put routine for the write queue
`DLrsrv()`	service routine for the read queue
`DLrecv()`	process a completely formed incoming packet

Note that all of these functions have a common prefix of `DL`. This is to make the functions hardware independent. An `#include` file in the Hardware Dependent Layer converts these functions to be hardware specific. See “Function Names and File Organization.”

Further details on these and related functions can be found in the on-line manual pages. “DLPI Primitives” contains information on the meaning of error codes used in the procedure list.

The Hardware Dependent Layer

The Hardware Dependent Layer provides the services of an external I/O device (Ethernet controller). It handles data transfer between the kernel and the device and is not involved in DLPI interface processing other than conversion between data structures used by the STREAMS mechanism and data structures that the device understands. Additionally, it updates all the fields (other than the ones mentioned above) in the MIB structure.

Further details on functions for the Hardware Dependent Layer can be found on the manual pages in the on-line manual pages.

Watchdog Routines

All implementations of Ethernet drivers contain a watchdog routine. A watchdog routine monitors adapter activity and informs the user of any malfunctions. The **`timeout()`** routine that is available as a part of the operating system forms the basis for watchdog activity. The algorithm is as follows:

- The `timer_val` field of the `DL_bdconfig_t` structure is set to an appropriate value and a call to **`timeout()`** (with the watchdog routine as one of its arguments) is made after each packet is transmitted (**`DLxmit_packet`**`()`) routine. The `timer_id` field of `DL_bdconfig_t` is updated to reflect the return value from **`timeout()`**.
- Every call to the watchdog routine results in the following:
  - The `timer_val` is decremented.
  - If the `timer_val` is zero, a warning message is printed out on the console.
  - A non zero of value of `timer_val` results in another call to `time-out` with the original set of arguments.
- As a part of the interrupt processing associated with successful transmission of packets, a call to **`untimeout`** is issued to cancel the watchdog activity. The `timer_id` field of `DL_bdconfig_t` is used as the argument to **`untimeout`**.

Function Names and File Organization

All function names used (but not necessarily defined) in the Hardware Independent Layer have the common prefix of DL. Function names in the Hardware Dependent Layer have the common prefix used with the STREAMS initialization. An include file maps the general names of the Hardware Independent Layer to driver specific names of the Hardware Dependent Layer. This allows us to use the same Hardware Independent Layer routines without any modification with different device-specific Hardware Dependent Layer routines to get different drivers. For example, the function **`DLxmit_packet()`** in the Hardware Independent Layer is defined as **`eglxmit_packet()`** in the `egl` driver and as **`hpexmit_packet()`** in the `hpe` driver.

Model of the Data Link Layer

The data link layer (layer 2 in the OSI Reference Model) is responsible for the transmission and error-free delivery of bits of information over a physical communications medium.

The model of the data link layer is presented here to describe concepts that are used throughout DLPI. It is described in terms of an interface architecture, as well as addressing concepts needed to identify different components of that architecture. The description of the model assumes familiarity with the OSI Reference Model.

Model of the Service Interface

Each layer of the OSI Reference Model has two standards:

- one that defines the services provided by the layer, and
- one that defines the protocol through which layer services are provided.

DLPI is an implementation of the first type of standard. It specifies an interface to the services of the data link layer. Figure 11-8 depicts the abstract view of DLPI.

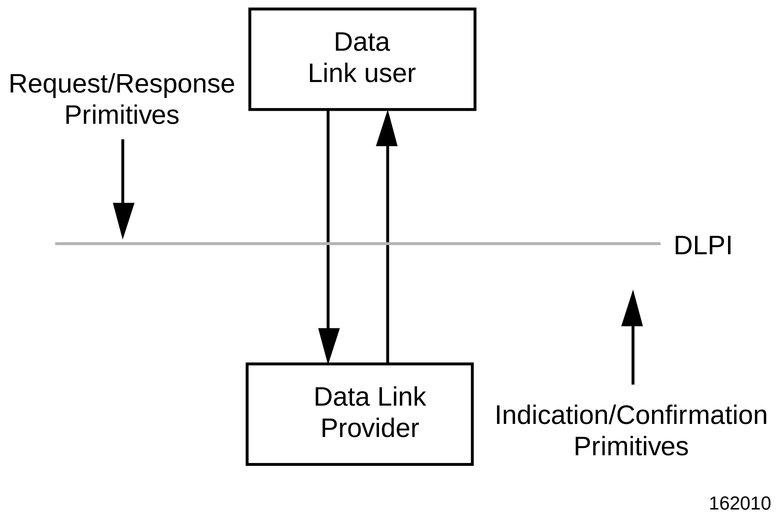

Data Link user
Request/Response Primitives
DLPI
Data Link Provider
Indication/Confirmation Primitives
162010

Figure 11-8. Abstract View of DLPI

The data link interface is the boundary between the network and data link layers of the OSI Reference Model. The network layer entity is the user of the services of the data link interface (DLS user), and the data link layer entity is the provider of those services (DLS provider). This interface consists of a set of primitives that provide access to the data link layer services, plus the rules for using those primitives (state transition rules). A data link interface service primitive might request a particular service or indicate a pending event.

To provide uniformity among the various PowerMAX OS system networking products, an effort is underway to develop service interfaces that map to the OSI Reference Model. A set of kernel-level interfaces, based on the STREAMS development environment, constitute a major portion of this effort. The service primitives that make up these interfaces are defined as STREAMS messages that are transferred between the user and provider of the service. DLPI is one such kernel-level interface, and is targeted for STREAMS protocol modules that either use or provide data link services. In addition, user programs that need to access a STREAMS-based data link provider directly may do so using the **`putmsg(2)`** and **`getmsg(2)`** system calls.

In Figure 11-8, the DLS provider is configured as a STREAMS driver, and the DLS user accesses the provider using **`open(2)`** to establish a stream to the DLS provider. The stream acts as a communication endpoint between a DLS user and the DLS provider. After the stream is created, the DLS user and DLS provider communicate via messages.

DLPI is intended to free data link users from specific knowledge of the characteristics of the data link provider. Specifically, the definition of DLPI hopes to achieve the goal of allowing a DLS user to be implemented independent of a specific communications medium. Any data link provider (supporting any communications medium) that conforms to DLPI may be substituted beneath the DLS user to provide the data link services. Support of a new DLS provider should not require any changes to the implementation of the DLS user.

Modes of Communication

The data link provider interface supports two modes of communication: connection and connectionless. The connection mode is circuit-oriented and enables data to be transferred over a pre-established connection in a sequenced manner. Data may be lost or corrupted in this service mode, however, due to provider-initiated re-synchronization or connection aborts.

The connectionless mode is message-oriented and supports data transfer in self-contained units with no logical relationship required between units. Because there is no acknowledgment of each data unit transmission, this service mode can be unreliable in the most general case. However, a specific DLS provider can provide assurance that messages will not be lost, duplicated, or reordered.

The acknowledged connectionless mode provides the means by which a data link user can send data and request the return of data at the same time. Although the exchange service is connectionless, in-sequence delivery is guaranteed for data sent by the initiating station. The data unit transfer is point-to-point.

Connection-Mode Service

The connection-mode service is characterized by four phases of communication:

Local Management	This phase enables a DLS user to initialize a stream for use in communication and establish an identity with the DLS provider.
Connection Establishment	This phase enables two DLS users to establish a data link connection between them to exchange data. One user (the calling DLS user) initiates the connection establishment procedures, while another user (the called DLS user) waits for incoming connect requests. The called DLS user is identified by an address associated with its stream. For both the calling and called DLS users, only one connection may be established per stream. Thus, the stream is the communication endpoint for a data link connection. The called DLS user may choose to accept a connection on the stream where it received the connect request, or it may open a new stream to the DLS provider and accept the connection on this new, responding stream. By accepting the connection on a separate stream, the initial stream can be designated as a listening stream through which all connect requests will be processed. As each request arrives, a new stream (communication endpoint) can be opened to

	handle the connection, enabling subsequent requests to be queued on a single stream until they can be processed.
Data Transfer	In this phase, the DLS users are considered peers and may exchange data simultaneously in both directions over an established data link connection. Either DLS user may send data to its peer DLS user at any time. Data sent by a DLS user is guaranteed to be delivered to the remote user in the order in which it was sent.
Connection Release	This phase enables either DLS user, or the DLS provider, to break an established connection. The release procedure is considered abortive, so any data that has not reached the destination user when the connection is released may be discarded by the DLS provider.

Connectionless-Mode Service

The connectionless mode service does not use the connection establishment and release phases of the connection-mode service. The local management phase is still required to initialize a stream. Once initialized, however, the connectionless data transfer phase is immediately entered. Because there is no established connection, however, the connectionless data transfer phase requires the DLS user to identify the destination of each data unit to be transferred. The destination DLS user is identified by the address associated with its user.

Connectionless data transfer does not guarantee that data units will be delivered to the destination user in the order in which they were sent. Furthermore, it does not guarantee that a given data unit will reach the destination DLS user, although a given DLS provider may provide assurance that data will not be lost.

DLPI Addressing

Each user of DLPI must establish an identity to communicate with other data link users. This identity consists of two pieces. First, the DLS user must somehow identify the physical medium over which it will communicate. This is particularly evident on systems that are attached to multiple physical media. Second, the DLS user must register itself with the DLS provider so that the provider can deliver protocol data units destined for that user. Figure 11-9 illustrates the components of this identification approach.

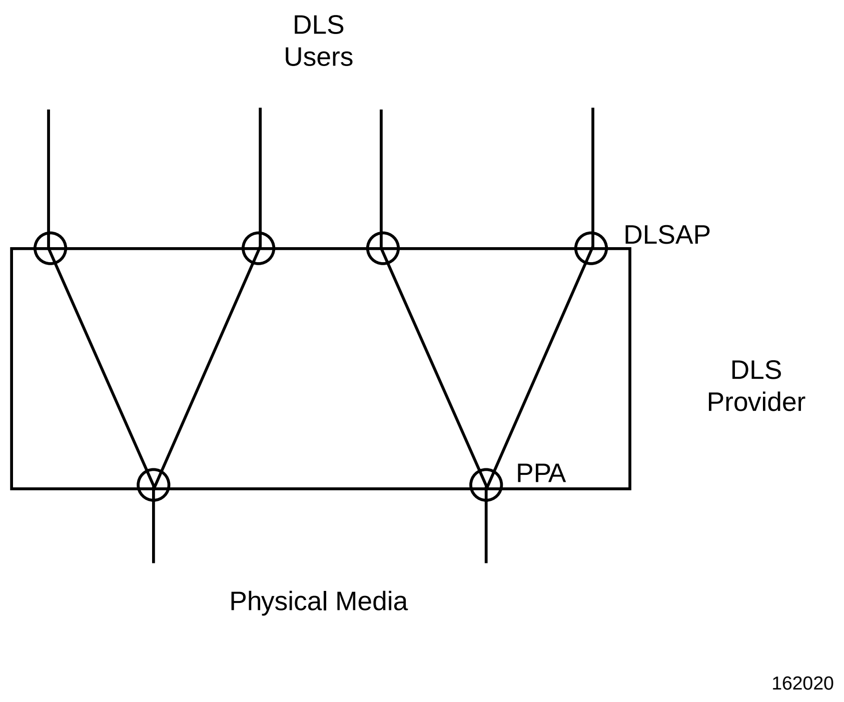


Figure 11-9. Data Link Addressing Components

Physical Attachment Identification

The *physical point of attachment* (PPA in Figure 11-9) is the point at which a system attaches itself to a physical communications medium. All communication on that physical medium funnels through the PPA. On systems where a DLS provider supports more than one physical medium, the DLS user must identify which medium it will communicate through. A PPA is identified by a unique PPA identifier. For media that support physical layer multiplexing of multiple channels over a single physical medium (such as the B and D channels of ISDN), the PPA identifier must identify the specific channel over which communication will occur.

Two styles of DLS provider are defined by DLPI, distinguished by the way they enable a DLS user to choose a particular PPA. The *style* 1 provider assigns a PPA based on the major/minor device the DLS user opened. One possible implementation of a *style* 1 driver would reserve a major device for each PPA the data link driver would support. This would allow the STREAMS **`cloneopen`** feature to be used for each PPA configured. This style of provider is appropriate when few PPAs will be supported.

If the number of PPAs a DLS provider will support is large, a *style* 2 provider implementation is more suitable. The *style* 2 provider requires a DLS user to explicitly identify the desired PPA using a special **`attach`** service primitive. For a *style* 2 driver, the **`open(2)`** creates a stream between the DLS user and DLS provider, and the **`attach`** primitive then associates a particular PPA with that stream. The format of the PPA identifier is specific to the DLS provider, and should be described in the provider-specific addendum documentation.

Data Link User Identification

A data link user's identity is established by associating it with a data link service access point (DLSAP), which is the point through which the user will communicate with the data link provider. A DLSAP is identified by a DLSAP address.

The DLSAP address identifies a particular data link service access point that is associated with a stream (communication endpoint). A **`bind`** service primitive enables a DLS user to either choose a specific DLSAP by specifying its DLSAP address, or to determine the DLSAP associated with a stream by retrieving the bound DLSAP address. This DLSAP address can then be used by other DLS users to access a specific DLS user. The format of the DLSAP address is specific to the DLS provider, and should be described in the provider-specific addendum documentation. However, DLPI provides a mechanism for decomposing the DLSAP address into component pieces. The `DL_INFO_ACK` primitive returns the length of the SAP component of the DLSAP address, along with the total length of the DLSAP address.

Certain DLS Providers require the capability of binding on multiple DLSAP addresses. This can be achieved through subsequent binding of DLSAP addresses. DLPI supports *peer* and *hierarchical* binding of DLSAPs. When the User requests peer addressing, the DLSAP specified in a subsequent bind may be used in lieu of the DLSAP bound in the `DL_BIND_REQ`. This will allow for a choice to be made between a number of DLSAPs on a stream when determining traffic based on DLSAP values. An example of this would be to specify various `ether_type` values as DLSAPs. The `DL_BIND_REQ`, for example, could be issued with `ether_type` value of IP, and a subsequent bind could be issued with ether type value of ARP. The Provider may now multiplex off of the `ether_type` field and allow for either IP or ARP traffic to be sent up this stream.

When the DLS User requests hierarchical binding, the subsequent bind will specify a DLSAP that will be used in addition to the DLSAP bound using a `DL_BIND_REQ`. This will allow additional information to be specified, that will be used in a header or used for demultiplexing. An example of this would be to use hierarchical bind to specify the OUI (Organizationally Unique Identifier) to be used by SNAP.

If a DLS Provider supports peer subsequent bind operations, the first SAP that is bound is used as the source SAP when there is ambiguity.

DLPI supports the ability to associate several streams with a single DLSAP, where each stream may be a unique data link connection endpoint. However, not all DLS providers can support such configurations because some DLS providers may have no mechanism beyond the DLSAP address for distinguishing multiple connections. In such cases, the provider will restrict the DLS user to one stream per DLSAP.

The Connection Management Stream

The earlier description of the connection-mode service assumed that a DLS user bound a DLSAP to the stream it would use to receive connect requests. In some instances, however, it is expected that a given service may be accessed through any one of several DLSAPs. To handle this scenario, a separate stream would be required for each possible destination DLSAP, regardless of whether any DLS user actually requested a connection to that DLSAP. Obvious resource problems can result in this scenario.

To obviate the need for tying up system resources for all possible destination DLSAPs, a *connection* management stream utility is defined in DLPI. A connection management stream is one that receives any connect requests that are not destined for currently bound DLSAPs capable of receiving connect indications. With this mechanism, a special listener can handle incoming connect requests intended for a set of DLSAPs by opening a connection management stream to the DLS provider that will retrieve all connect requests arriving through a particular PPA. In the model, then, there may be a connection management stream per PPA.

DLPI Services

The various features of the DLPI interface are defined in terms of the services provided by the DLS provider, and the individual primitives that may flow between the DLS user and DLS provider.

The data link provider interface supports three modes of service:

Connection	The connection mode is circuit-oriented and enables data to be transferred over an established connection in a sequenced manner.
Connectionless	The connectionless mode is message-oriented and supports data transfer in self-contained units with no logical relationship required between units.
Acknowledged Connectionless	The *acknowledged* connectionless mode is similar to connectionless mode, however, messages are acknowledged and in-sequence delivery is guaranteed for sender data.

The XID and TEST services that are supported by DLPI are listed in Table 11-1. The DLS user can issue an XID or TEST request to the DLS Provider. The Provider will transmit an XID or TEST frame to the peer DLS Provider. On receiving a response, the DLS Provider sends a confirmation primitive to the DLS user. On receiving an XID or TEST frame from the peer DLS Provider, the local DLS Provider sends up an XID or TEST indication primitive to the DLS user. The user must respond with an XID or TEST response frame to the Provider

The services are shown in Table 11-1 and described more fully in the remainder of this section.

Table 11-1. DLS Services and Primitives

Phase	Service	Primitives
Local Management	Information Reporting	`DL_INFO_REQ, DL_INFO_ACK, DL_ERROR_ACK`
	Attach	`DL_ATTACH_REQ, DL_DETACH_REQ, DL_OK_ACK, DL_ERROR_ACK`
	Bind	`DL_BIND_REQ, DL_BIND_ACK, DL_SUBS_BIND_REQ, DL_SUBS_BIND_ACK, DL_UNBIND_REQ, DL_SUBS_UNBIND_REQ, DL_OK_ACK, DL_ERROR_ACK`
	Other	`DL_ENABMULTI_REQ, DL_DISABMULTI_REQ, DL_OK_ACK, DL_ERROR_ACK`
	Optional	`DL_GET_STATISTICS_ACK, DL_GET_STATISTICS_REQ, DL_PHYS_ADDR_ACK, DL_PHYS_ADDR_REQ, DL_SET_PHYS_ADDR_REQ`
Connection Establishment	Connection Establishment	`DL_CONNECT_REQ, DL_CONNECT_IND, DL_CONNECT_RES, DL_CONNECT_CON, DL_DISCONNECT_REQ, DL_DISCONNECT_IND, DL_TOKEN_REQ, DL_TOKEN_ACK, DL_OK_ACK, DL_ERROR_ACK`
Connection-mode Data Transfer	Data Transfer	`DL_DATA_REQ, DL_DATA_IND`
	Reset	`DL_RESET_REQ, DL_RESET_IND, DL_RESET_RES, DL_RESET_CON, DL_OK_ACK, DL_ERROR_ACK`
Connection Release	Connection Release	`DL_DISCONNECT_REQ, DL_DISCONNECT_IND, DL_OK_ACK, DL_ERROR_ACK`
Connectionless-mode Data Transfer	Data Transfer	`DL_UNITDATA_REQ, DL_UNITDATA_IND`

Table 11-1. DLS Services and Primitives (Cont.)

Phase	Service	Primitives
	QOS Management	DL_UDQOS_REQ, DL_OK_ACK, DL_ERROR_ACK
	Error Reporting	DL_UDERROR_IND
XID and TEST services	XID	DL_XID_REQ, DL_XID_IND, DL_XID_RES, DL_XID_CON
	TEST	DL_TEST_REQ, DL_TEST_IND, DL_TEST_RES, DL_TEST_CON
Acknowledged Connectionless-mode Data Transfer	Data Transfer	DL_DATA_ACK_REQ, DL_DATA_ACK_IND, DL_DATA_ACK_STATUS_IN, DL_REPLY_REQ, DL_REPLY_IND, DL_REPLY_STATUS_IND, DL_REPLY_UPDATE_REQ, DL_REPLY_UPDATE_STATUS_IND
	QOS Management	DL_UDQOS_REQ, DL_OK_ACK, DL_ERROR_ACK
	Error Reporting	DL_UDERROR_IND

Local Management Services

The local management services apply to both the connection and connectionless modes of transmission. These services, which fall outside the scope of standards specifications, define the method for initializing a stream that is connected to a DLS provider. DLS provider information reporting services are also supported by the local management facilities

Information Reporting

This service provides information about the DLPI stream to the DLS user. The message `DL_INFO_REQ` requests the DLS provider to return operating information about the stream. The DLS provider returns the information in a `DL_INFO_ACK` message. Figure 11-10 shows the normal message sequence.

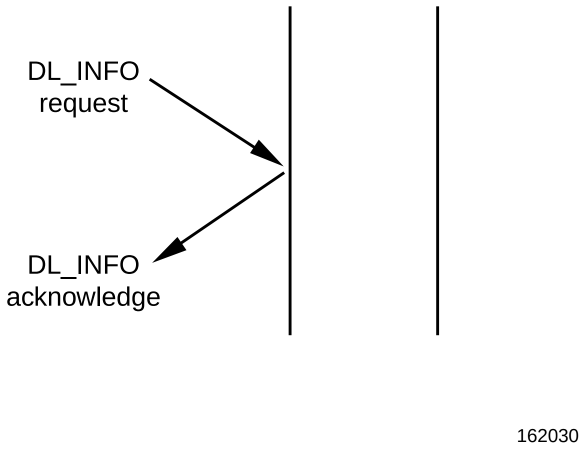


Figure 11-10. Information Reporting

Attach Service

The attach service assigns a physical point of attachment (PPA) to a stream. This service is required for *style* 2 DLS providers to specify the physical medium over which communication will occur. The DLS provider indicates success with a `DL_OK_ACK`, and indicates failure with a `DL_ERROR_ACK`. The normal message sequence is illustrated in Figure 11-11.

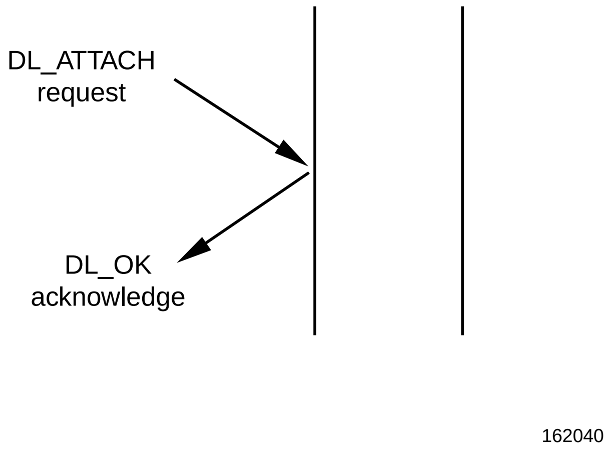


Figure 11-11. Attaching a Stream to a Physical Line

A PPA may be disassociated with a stream using the `DL_DETACH_REQ`. The normal message sequence is illustrated in Figure 11-12.

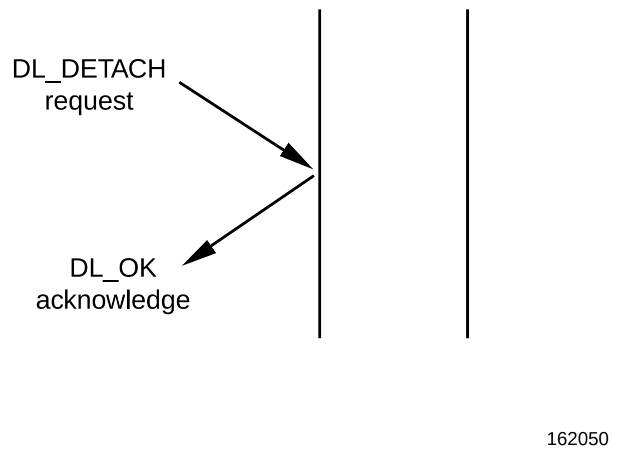


Figure 11-12. Detaching a Stream to a Physical Line

Bind Service

The bind service associates a data link service access point (DLSAP) with a stream. The DLSAP is identified by a DLSAP address. Each stream open to a DLS provider can have at most one DLSAP associated with it.

`DL_BIND_REQ` requests that the DLS provider bind a DLSAP to a stream. It also notifies the DLS provider to make the stream active with respect to the DLSAP for processing connectionless data transfer and connection establishment requests. Protocol-specific actions taken during activation should be described in DLS provider-specific addenda.

The DLS provider indicates success with a `DL_BIND_ACK`, and indicates failure with a `DL_ERROR_ACK`.

Certain DLS providers require the capability of binding on multiple DLSAP addresses. `DL_SUBS_BIND_REQ` provides that added capability. The DLS provider indicates success with a `DL_SUBS_BIND_ACK`, and indicates failure with a `DL_ERROR_ACK`.

The normal flow of messages is illustrated in Figure 11-13.

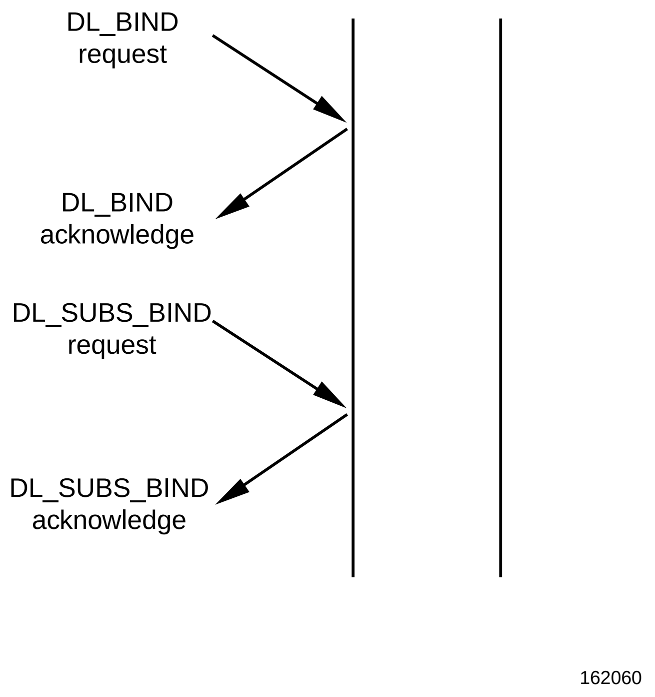


Figure 11-13. Binding a Stream to a DLSAP

`DL_UNBIND_REQ` requests the DLS provider to unbind a DLSAP from a stream. The DLS provider indicates success with a `DL_OK_ACK` and indicates failure with a `DL_ERROR_ACK`. The normal message sequence is shown in Figure 11-14.

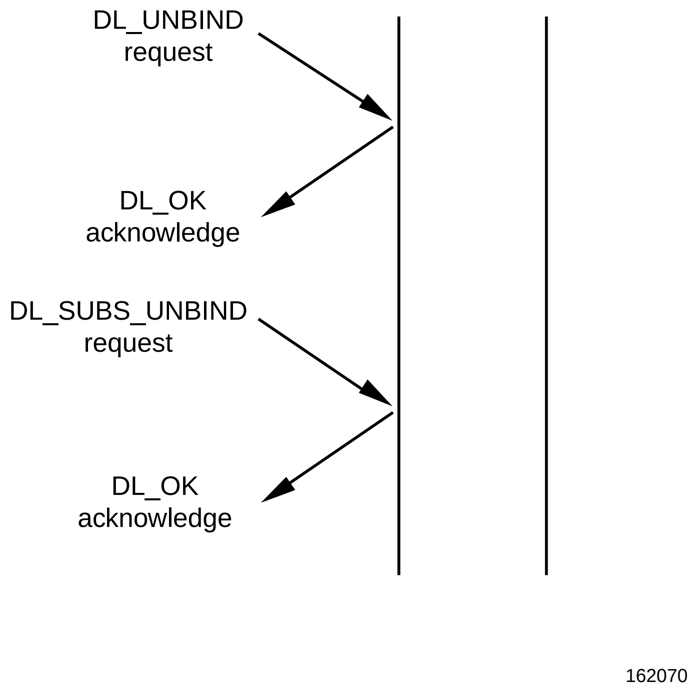


Figure 11-14. Unbinding a Stream to a DLSAP

`DL_ENABMULTI_REQ` requests that the DLS Provider enable specific multicast addresses on a per stream basis. The Provider indicates success with a `DL_OK_ACK`, and indicates failure with a `DL_ERROR_ACK` as shown in Figure 11-15.

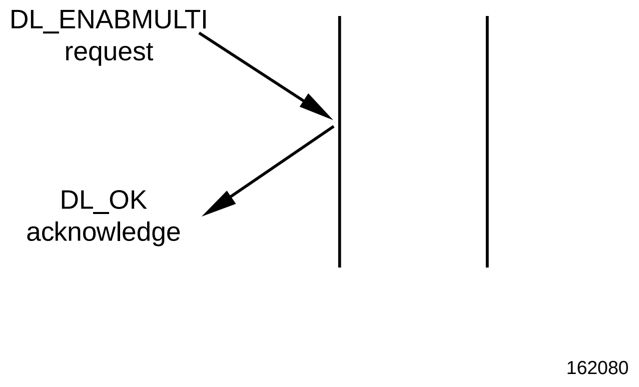


Figure 11-15. Enabling a Specific Multicast Address on a Stream

`DL_DISABMULTI_REQ` requests that the DLS Provider disables specific multicast addresses on a per Stream basis. The Provider indicates success with a `DL_OK_ACK`, and indicates failure with a `DL_ERROR_ACK` as shown in Figure 11-16.

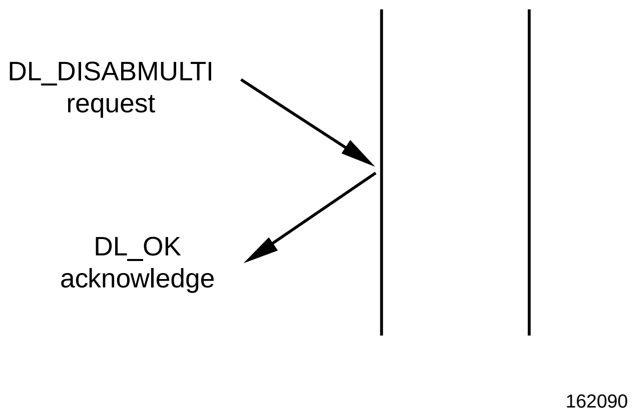


Figure 11-16. Disabling a Specific Multicast Address on a Stream

Connection-mode Services

The connection-mode services enable a DLS user to establish a data link connection, transfer data over that connection, reset the link, and release the connection when the conversation has terminated.

Connection Establishment Service

The connection establishment service establishes a data link connection between a local DLS user and a remote DLS user for the purpose of sending data. Only one data link connection is allowed on each stream.

Normal Connection Establishment

In the connection establishment model, the calling DLS user initiates connection establishment, while the called DLS user waits for incoming requests. `DL_CONNECT_REQ` requests that the DLS provider establish a connection. `DL_CONNECT_IND` informs the called DLS user of the request, which may be accepted using `DL_CONNECT_RES`. `DL_CONNECT_CON` informs the calling DLS user that the connection has been established.

The normal sequence of messages is illustrated in Figure 11-17.

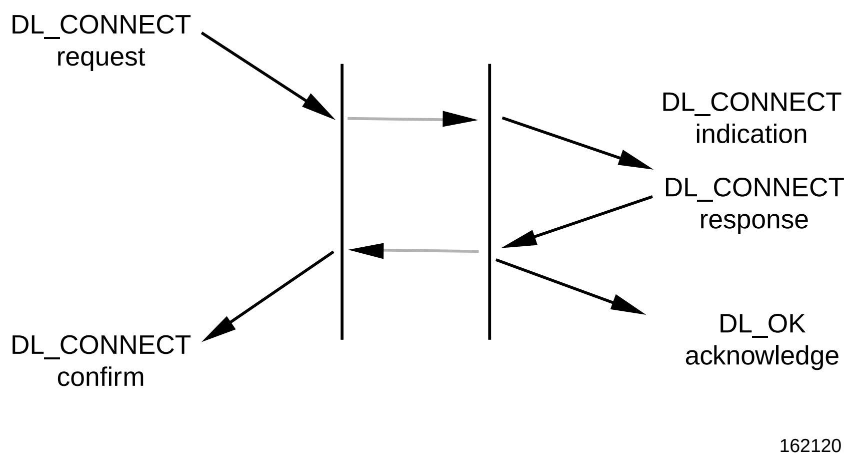


Figure 11-17. Successful Connection Establishment

Once the connection is established, the DLS users may exchange user data using `DL_DATA_REQ` and `DL_DATA_IND`.

The DLS user may accept an incoming connect request on either the stream where the connect indication arrived or an alternate, responding stream. The responding stream is indicated by a token in the `DL_CONNECT_RES`. This token is a value associated with the responding stream, and is obtained by issuing a `DL_TOKEN_REQ` on that stream. The DLS provider responds to this request by generating a token for the stream and returning it to the DLS user in a `DL_TOKEN_ACK`. The normal sequence of messages for obtaining a token is illustrated in Figure 11-18.

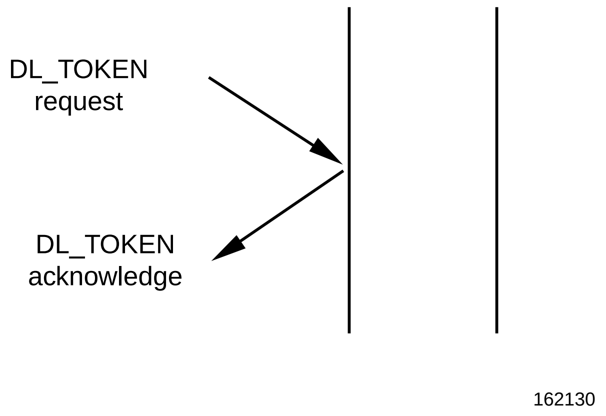


Figure 11-18. Token Retrieval

In the typical connection establishment scenario, the called DLS user processes one connect indication at a time, accepting the connection on another stream. Once the user responds to the current connect indication, the next connect indication (if any) can be processed. DLPI also enables the called DLS user to multi-thread incoming connect indications. The user can receive multiple connect indications before responding to any of them. This enables the DLS user to establish priority schemes on incoming connect requests.

Connection Establishment Rejections

In certain situations, the connection establishment request cannot be completed. The following paragraphs describe the occasions under which `DL_DISCONNECT_REQ` and `DL_DISCONNECT_IND` primitives will flow during connection establishment, causing the connect request to be aborted.

Figure 11-19 shows an example where the called DLS user chooses to reject the connect request by issuing `DL_DISCONNECT_REQ` instead of `DL_CONNECT_RES`.

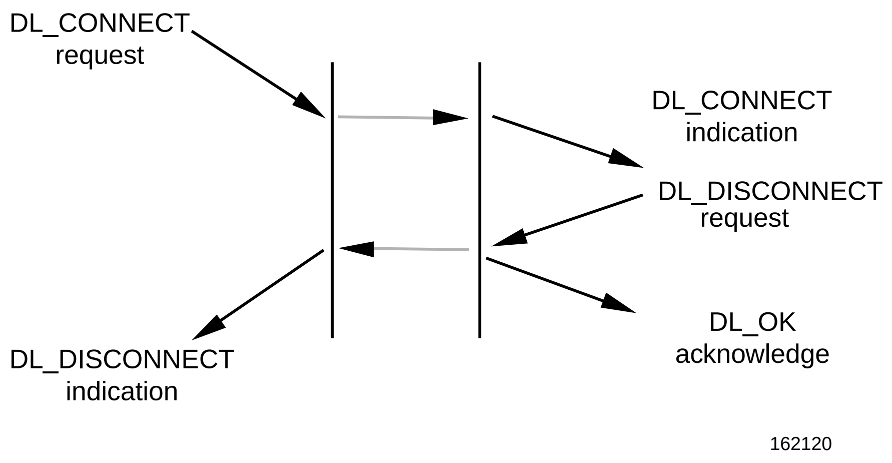


Figure 11-19. DLS User Rejection of Connection Establishment Attempt

Figure 11-20 shows an example where the DLS provider rejects a connect request for lack of resources or other reason. The DLS provider sends `DL_DISCONNECT_IND` in response to `DL_CONNECT_REQ`.

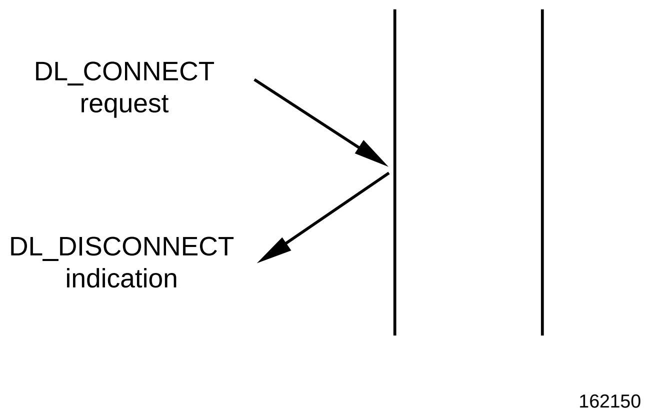


Figure 11-20. DLS Provider Rejection of Connection Establishment Attempt

Figure 11-21 shows an example where the calling DLS user chooses to abort a previous connection attempt. The DLS user issues `DL_DISCONNECT_REQ` at some point following a `DL_CONNECT_REQ`. The resulting sequence of primitives depends on the relative timing of the primitives involved, as defined in the following time sequence diagrams.

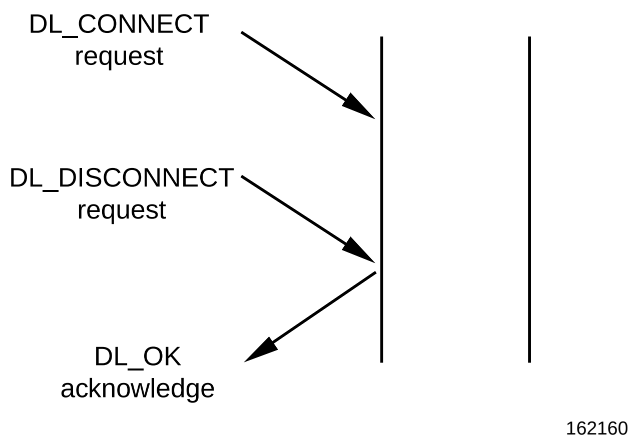


Figure 11-21. Both Primitives Are Destroyed by Provider

Data Transfer Service

The connection-mode data transfer service provides for the exchange of user data in either direction or in both directions simultaneously between DLS users. Data is transmitted in logical groups called data link service data units (DLSDUs). The DLS provider preserves both the sequence and boundaries of DLSDUs as they are transmitted.

Normal data transfer is neither acknowledged nor confirmed. It is up to the DLS users, if they so choose, to implement a confirmation protocol.

Each DL_DATA_REQ primitive conveys a DLSDU from the local DLS user to the DLS provider. Similarly, each DL_DATA_IND primitive conveys a DLSDU from the DLS provider to the remote DLS user. The normal flow of messages is illustrated in Figure 11-22.

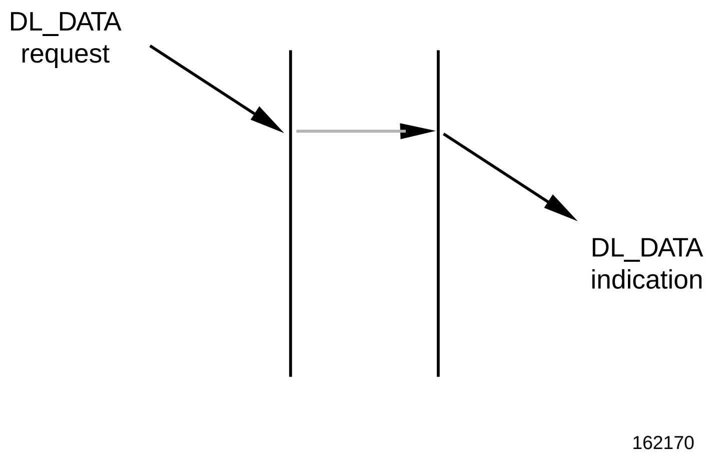


Figure 11-22. Normal Flow

Connection Release Service

The connection release service provides for the DLS users or the DLS provider to initiate the connection release. Connection release is an abortive operation, and any data in transit (has not been delivered to the DLS user) may be discarded.

DL_DISCONNECT_REQ requests that a connection be released. DL_DISCONNECT_IND informs the DLS user that a connection has been released. Normally, one DLS user requests disconnection and the DLS provider issues an indication of the ensuing release to the other DLS user, as illustrated by the message flow in Figure 11-23.

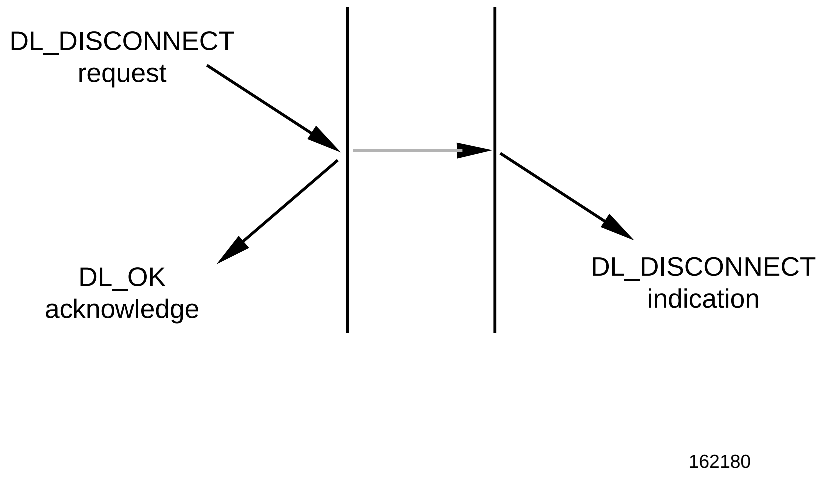


Figure 11-23. DLS User-Invoked Connection Release

Figure 11-24 illustrates that when two DLS users independently invoke the connection release service, neither receives a `DL_DISCONNECT_IND`.

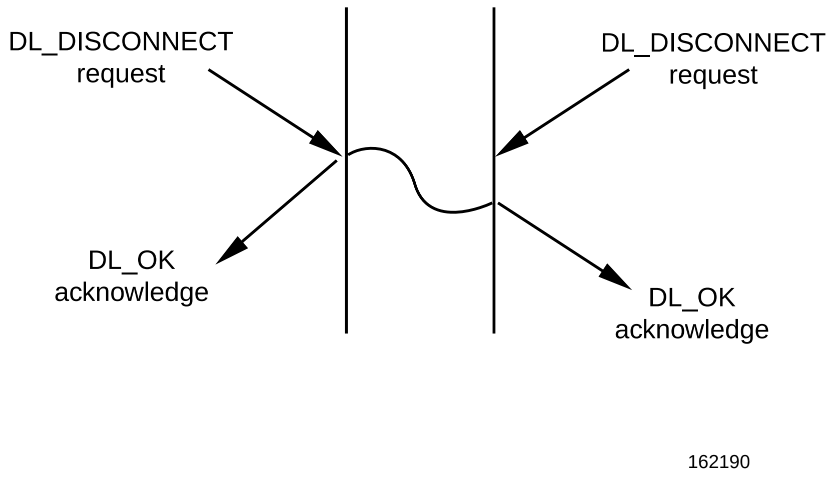


Figure 11-24. Simultaneous DLS User-Invoked Connection Release

Figure 11-25 shows that when a DLS provider initiates the connection release service, each DLS user receives a `DL_DISCONNECT_IND`.

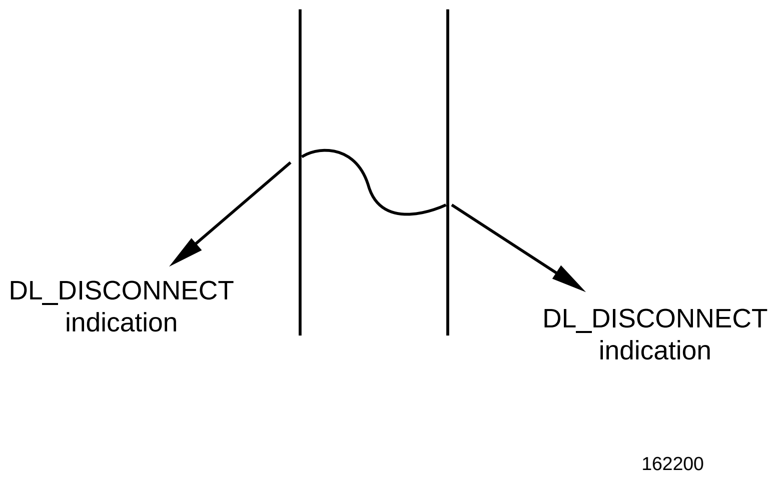


Figure 11-25. DLS Provider Invoked Connection Release

Figure 11-26 illustrates that when the DLS provider and the local DLS user simultaneously invoke the connection release service, the remote DLS user receives a `DL_DISCONNECT_IND`.

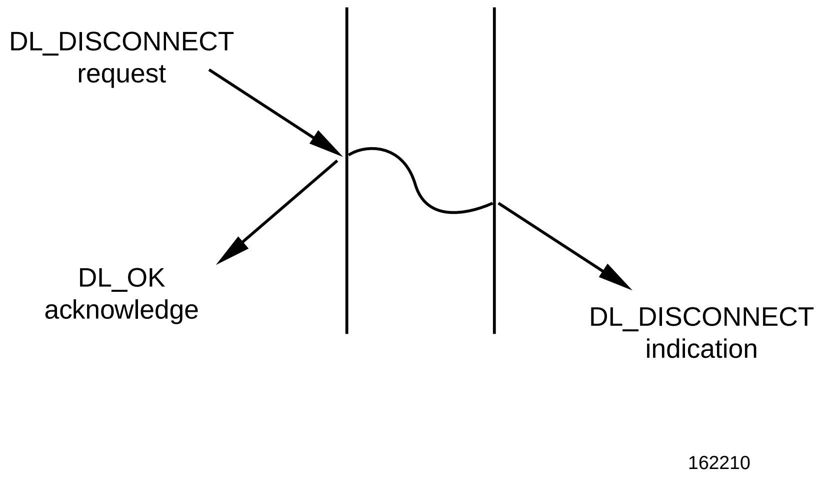


Figure 11-26. Simultaneous DLS User and Provider Connection Release

Reset Service

The reset service may be used by the DLS user to resynchronize the use of a data link connection, or by the DLS provider to report detected loss of data unrecoverable within the data link service.

Invocation of the reset service will unblock the flow of DLSDUs if the data link connection is congested. DLSDUs may be discarded by the DLS provider. The DLS user or users that did not invoke the reset will be notified that a reset has occurred. A reset may require a recovery procedure to be performed by the DLS users.

The interaction between each DLS user and the DLS provider will be one of the following:

- a `DL_RESET_REQ` from the DLS user, followed by a `DL_RESET_CON` from the DLS provider;
- a `DL_RESET_IND` from the DLS provider, followed by a `DL_RESET_RES` from the DLS user.

The `DL_RESET_REQ` acts as a synchronization mark in the stream of DLSDUs that are transmitted by the issuing DLS user. The `DL_RESET_IND` acts as a synchronization mark in the stream of DLSDUs that are received by the peer DLS user. Similarly, the `DL_RESET_RES` acts as a synchronization mark in the stream of DLSDUs that are transmitted by the responding DLS user. The `DL_RESET_CON` acts as a synchronization mark in the stream of DLSDUs that are received by the DLS user which originally issued the reset.

The resynchronizing properties of the reset service are that:

- No DLSDU transmitted by the DLS user before the synchronization mark in that transmitted stream will be delivered to the other DLS user after the synchronization mark in that received stream.
- The DLS provider will discard all DLSDUs submitted before the issuing of the `DL_RESET_REQ` that have not been delivered to the peer DLS user when the DLS provider issues the `DL_RESET_IND`.
- The DLS provider will discard all DLSDUs and EDLSDUs submitted before the issuing of the `DL_RESET_RES` that have not been delivered to the initiator of the `DL_RESET_REQ` when the DLS provider issues the `DL_RESET_CON`.
- No DLSDU or EDLSDU transmitted by a DLS user after the synchronization mark in that transmitted stream will be delivered to the other DLS user before the synchronization mark in that received stream.

The complete message flow depends on the origin of the reset, which may be the DLS provider or either DLS user. Figure 11-27 illustrates the message flow for a reset invoked by one DLS user.

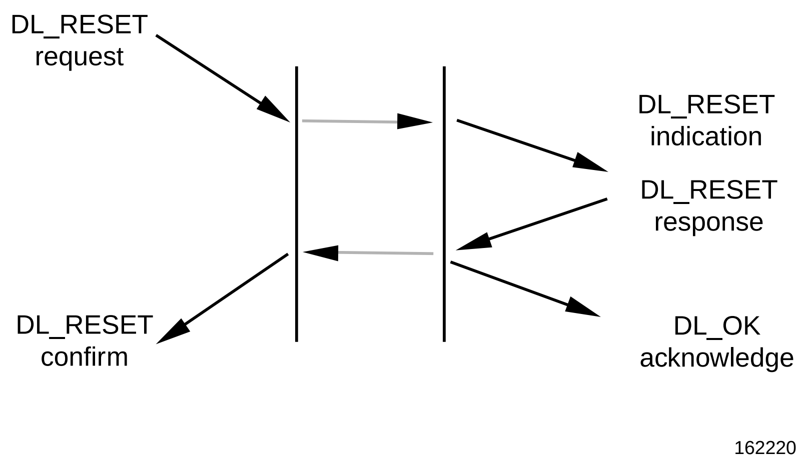


Figure 11-27. DLS User-Invoked Connection Reset

Figure 11-28 illustrates the message flow for a reset invoked by both DLS users simultaneously.

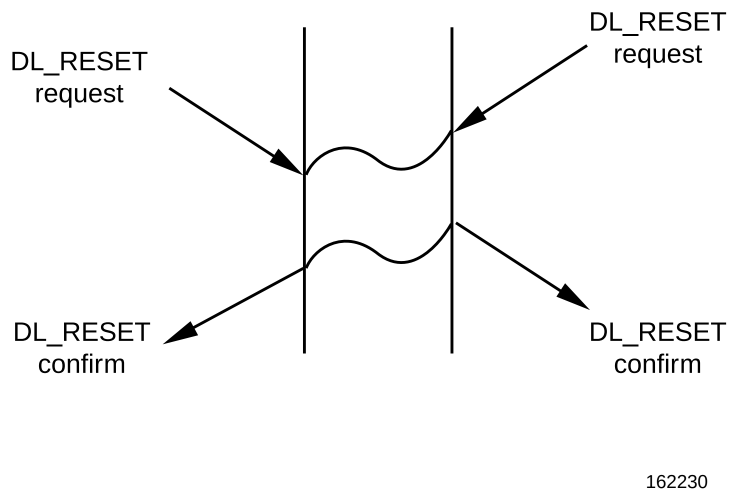


Figure 11-28. Simultaneous DLS User-Invoked Connection Reset

Figure 11-29 illustrates the message flow for a reset invoked by the DLS provider.

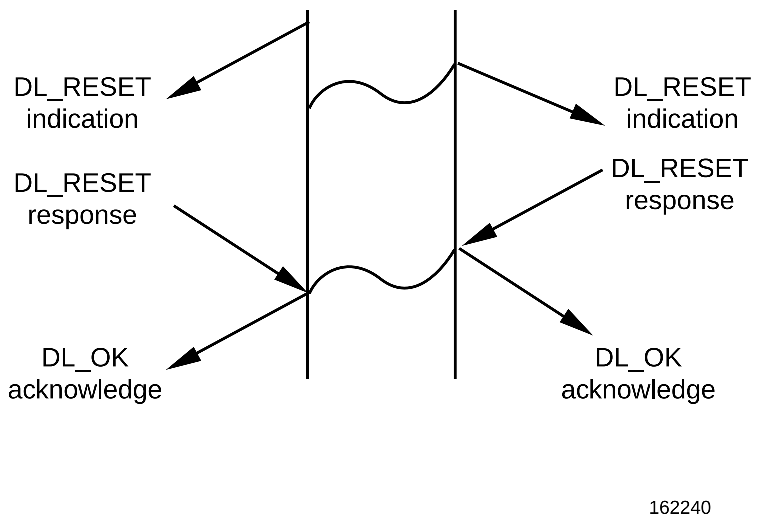


Figure 11-29. DLS Provider-Invoked Connection Reset

Figure 11-30 illustrates the message flow for a reset invoked simultaneously by one DLS user and the DLS provider.

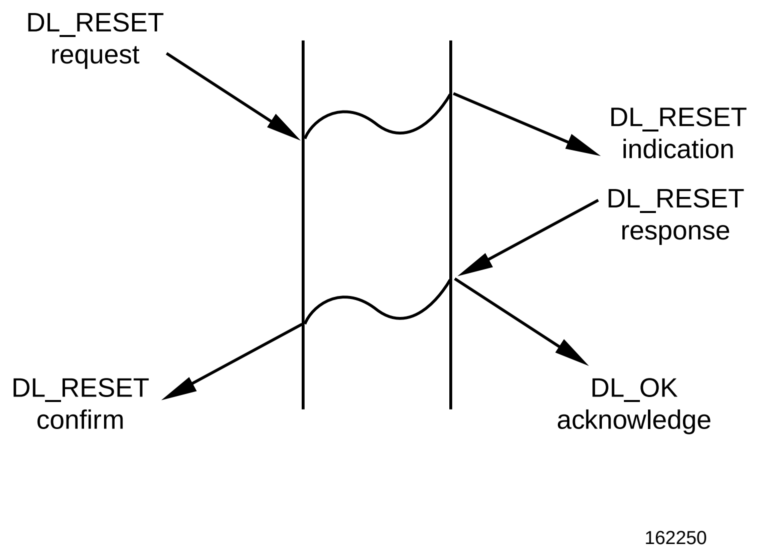


Figure 11-30. Simultaneous DLS User/Provider-Invoked Connection Reset

Connectionless-Mode Services

The connectionless-mode services enable a DLS user to transfer units of data to peer DLS users without incurring the overhead of establishing and releasing a connection. The connectionless service does not, however, guarantee reliable delivery of data units between peer DLS users. For example, a lack of flow control may cause buffer resource shortages that result in data being discarded.

Once a stream has been initialized via the local management services, it may be used to send and receive connectionless data units.

Connectionless Data Transfer Service

The connectionless data transfer service provides for the exchange of user data (DLSDUs) in either direction or in both directions simultaneously without having to establish a data link connection. Data transfer is neither acknowledged nor confirmed, and there is no end-to-end flow control provided. As such, the connectionless data transfer service cannot guarantee reliable delivery of data. However, a specific DLS provider can provide assurance that messages will not be lost, duplicated, or reordered.

`DL_UNITDATA_REQ` conveys one DLSDU to the DLS provider. `DL_UNITDATA_IND` conveys one DLSDU to the DLS user. The normal flow of messages is illustrated in Figure 11-31.

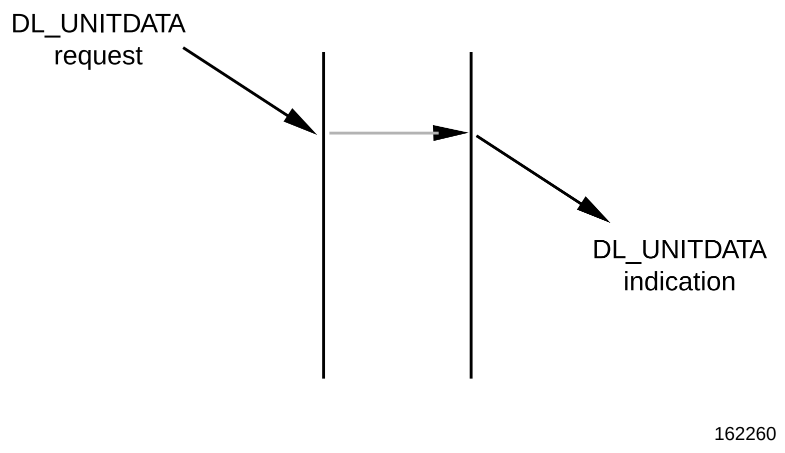


Figure 11-31. Connectionless Data Transfer

QOS Management Service

The QOS (Quality of Service) management service enables a DLS user to specify the quality of service it can expect for each invocation of the connectionless data transfer service. The `DL_UDQOS_REQ` directs the DLS provider to set the QOS parameters to the specified values. The normal flow of messages is illustrated in Figure 11-32.

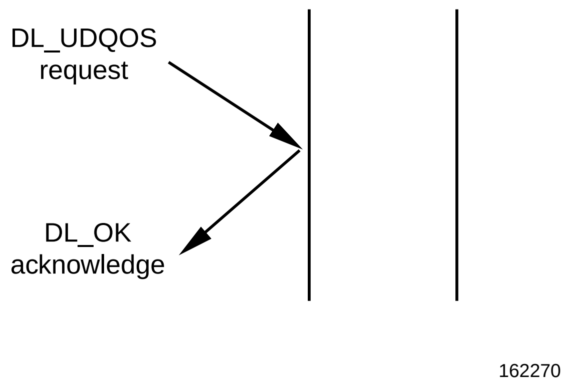


Figure 11-32. Connectionless Data Transfer (QOS)

Error Reporting Service

The connectionless-mode error reporting service may be used to notify a DLS user that a previously sent data unit either produced an error or could not be delivered. This service does not, however, guarantee that an error indication will be issued for every undeliverable data unit.

XID and TEST Service

The XID and TEST service enables the DLS User to issue an XID or TEST request to the DLS Provider. On receiving a response for the XID or TEST frame transmitted to the peer DLS Provider, the DLS Provider sends up an XID or TEST confirmation primitive to the DLS User. On receiving an XID or TEST frame from the peer DLS Provider, the local DLS Provider sends up an XID or TEST indication respectively to the DLS User. The DLS User must respond with an XID or TEST response primitive.

If the DLS User requested automatic handling of the XID or TEST response, at bind time, the DLS Provider will send up an error acknowledgment on receiving an XID or TEST request. Also, no indications will be generated to the DLS User on receiving XID or TEST frames from the remote side.

The normal flow of messages for the XID service is illustrated in Figure 11-33.

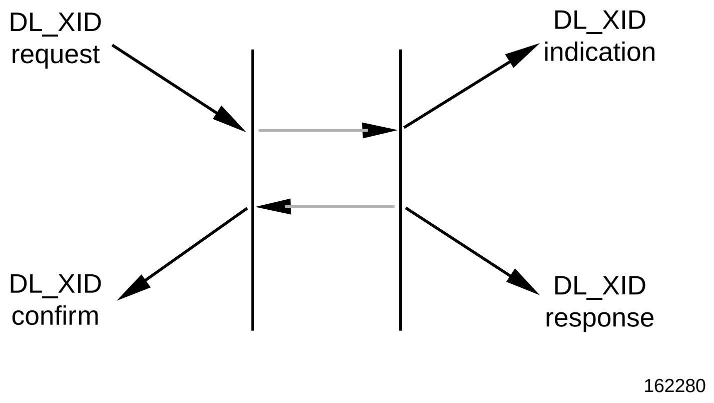


Figure 11-33. Message Flow: XID Service

The normal flow of messages for the TEST service is illustrated in Figure 11-34.

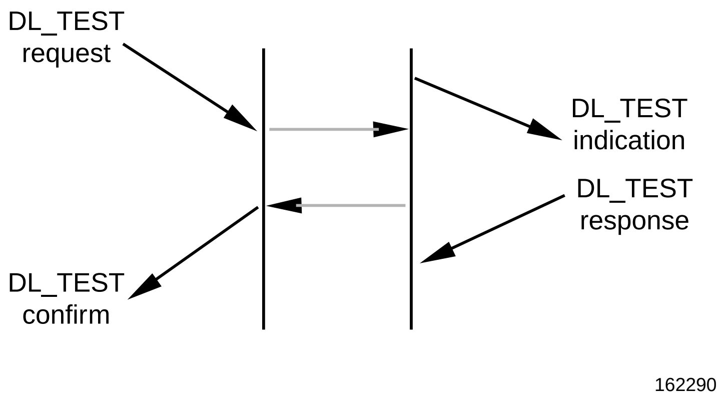


Figure 11-34. Message Flow: TEST Service

Acknowledged Connectionless-Mode Services

The acknowledged connectionless-mode services are designed for general use for the reliable transfer of information between peer DLS Users. These services are intended for applications that require acknowledgment of cross-LAN data unit transfer, but need to avoid the complexity that is viewed as being associated with the connection-mode services. Although the exchange service is connectionless, in-sequence delivery is guaranteed for data sent by the initiating station.

Acknowledged Connectionless-Mode Data Transfer Services

The acknowledged connectionless-mode data transfer services provide the means by which the DLS Users can exchange DLSDUs which are acknowledged at the LLC sublayer, without the establishment of a Data Link connection. The services provide a means by which a local DLS User can send a data unit to the peer DLS User, request a previously prepared data unit, or exchange data units with the peer DLS User.

Figure 11-35 illustrates the acknowledged connectionless-mode data unit transmission service.

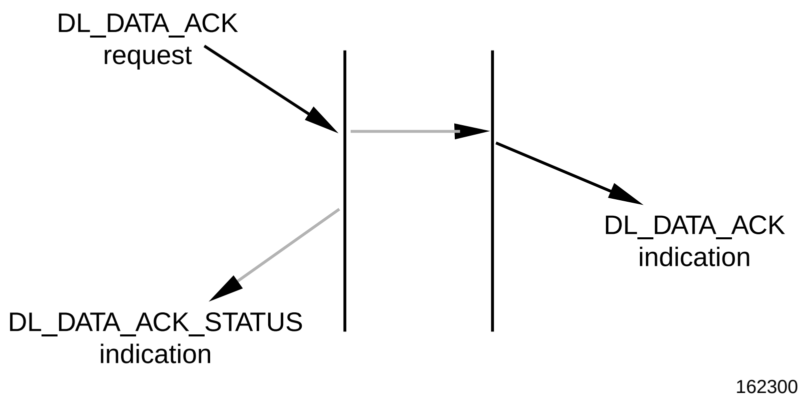


Figure 11-35. Acknowledged Connectionless-Mode Transmission Service

Figure 11-36 illustrates the acknowledged connectionless-mode data unit exchange service.

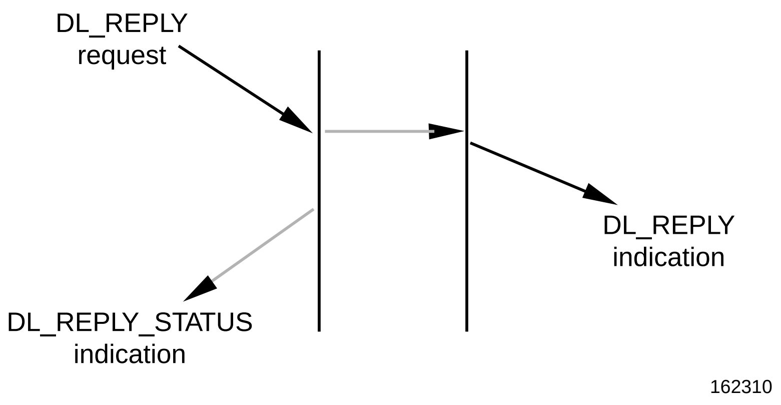


Figure 11-36. Acknowledged Connectionless-Mode Exchange Service

Figure 11-37 illustrates the Reply Data Unit Preparation service.

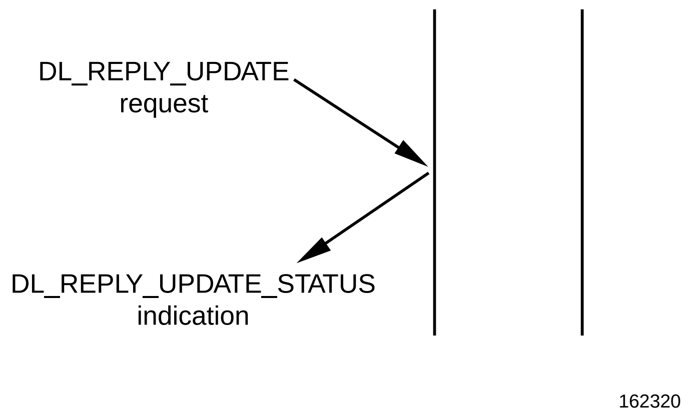


Figure 11-37. Reply Data Unit Preparation Service

Error Reporting Service

The acknowledged connectionless mode error reporting service is the same as the unacknowledged connectionless-mode error reporting service.

Connection-Mode Example

Figure 11-38 illustrates the primitives that flow during a complete, connection-mode sequence between stream **open** and stream **close**.

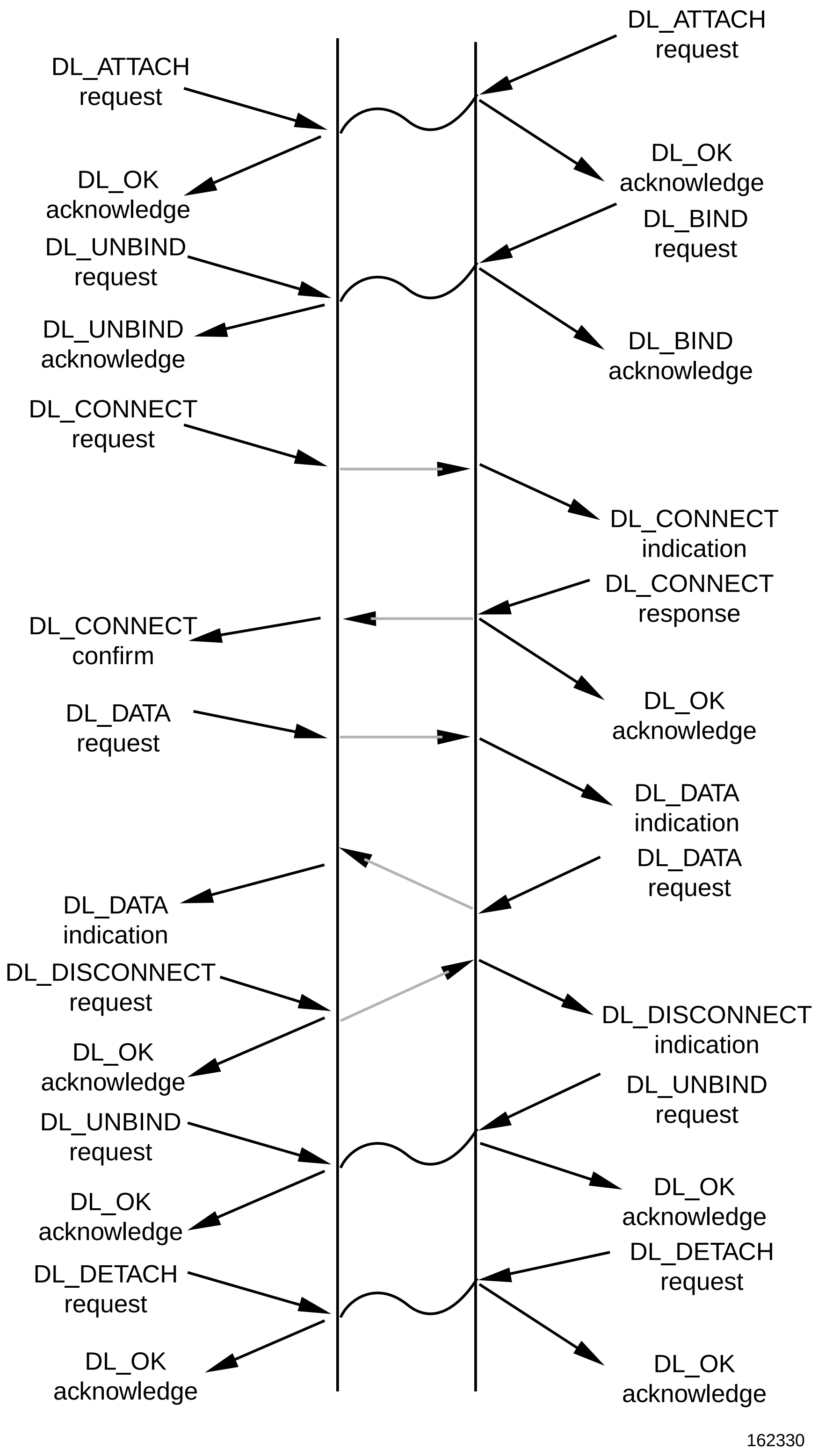


Figure 11-38. Connection Mode Example

DLPI Primitives

The kernel-level interface to the data link layer defines a Streams-based message interface between the provider of the data link service (DLS provider) and the consumer of the data link service (DLS user). Streams provides the mechanism in which DLPI primitives may be passed between the DLS user and DLS provider.

Before DLPI primitives can be passed between the DLS user and DLS provider, the DLS user must establish a stream to the DLS provider using **`open(2)`**. The DLS provider must therefore be configured as a STREAMS driver. When interactions between the DLS user and DLS provider have completed, the stream may be closed.

The Streams messages used to transport data link service primitives across the interface have one of the following formats:

- One `M_PROTO` message block followed by zero or more `M_DATA` blocks. The `M_PROTO` message block contains the data link layer service primitive type and all relevant parameters associated with the primitive. The `M_DATA` block(s) contain any DLS user data that might be associated with the service primitive.
- One `M_PCPROTO` message block containing the data link layer service primitive type and all relevant parameters associated with the service primitive.
- One or more `M_DATA` message blocks conveying user data.

The information contained in the `M_PROTO` or `M_PCPROTO` message blocks must begin on a byte boundary that is appropriate for structure alignment. STREAMS will allocate buffers that begin on such a boundary. However, these message blocks may contain information whose representation is described by a length and an offset within the block. An example is the DLSAP address (`dl_addr_length` and `dl_addr_offset`) in the `DL_BIND_ACK` primitive. The offset of such information within the message block is not guaranteed to be properly aligned for casting the appropriate data type (such as an `int` or a *structure*).

The following sections describe the format of the primitives that support the services described in the previous section. The primitives are grouped into four general categories:

- Local Management Service Primitives
- Connection-mode Service Primitives
- Connectionless-mode Service Primitives
- Acknowledged Connectionless-mode Service Primitives

Local Management Service Primitives

This section describes the local management service primitives that are common to both the connection and connectionless service modes. These primitives support the Information Reporting, Attach, and Bind services described earlier. Once a stream has been opened by a DLS user, these primitives initialize the stream, preparing it for use.

PPA Initialization / De-initialization

The PPA associated with each stream must be initialized before the DLS provider can transfer data over the medium. The initialization and de-initialization of the PPA is a network management issue, but DLPI must address the issue because of the impact such actions will have on a DLS user. More specifically, DLPI requires the DLS provider to initialize the PPA associated with a stream at some point before it completes processing of the `DL_BIND_REQ`. Guidelines for initialization and de-initialization of a PPA by a DLS provider are presented here.

A DLS provider may initialize a PPA using the following methods:

- Pre-initialized by some network management mechanism before the `DL_BIND_REQ` is received; or
- Automatic initialization on receipt of a `DL_BIND_REQ` or `DL_ATTACH_REQ`.

A specific DLS provider may support either of these methods, or possibly some combination of the two, but the method implemented has no impact on the DLS user. From the DLS user's viewpoint, the PPA is guaranteed to be initialized on receipt of a `DL_BIND_ACK`. For automatic initialization, this implies that the `DL_BIND_ACK` may not be issued until the initialization has completed.

If pre-initialization has not been performed and/or automatic initialization fails, the DLS provider will fail the `DL_BIND_REQ`. Two errors, `DL_INITFAILED` and `DL_NOTINIT`, may be returned in the `DL_ERROR_ACK` response to a DL_BIND_REQ if PPA initialization fails. `DL_INITFAILED` is returned when a DLS provider supports automatic PPA initialization, but the initialization attempt failed. `DL_NOTINIT` is returned when the DLS provider requires pre-initialization, but the PPA is not initialized before the `DL_BIND_REQ` is received.

A DLS provider may handle PPA de-initialization using the following methods:

- automatic de-initialization on receipt of the final `DL_DETACH_REQ` (for *style* 2 providers) or `DL_UNBIND_REQ` (for *style* 1 providers), or when closing of the last stream associated with the PPA;
- automatic de-initialization after expiration of a timer following the last `DL_DETACH_REQ`, `DL_UNBIND_REQ`, or close as appropriate; or
- no automatic de-initialization; administrative intervention is required to de-initialize the PPA at some point after it is no longer being accessed.

A specific DLS provider may support any of these methods, or possibly some combination of them, but the method implemented has no impact on the DLS user. From the DLS user's viewpoint, the PPA is guaranteed to be initialized and available for transmission until it closes or unbinds the stream associated with the PPA.

DLS provider-specific addendum documentation should describe the method chosen for PPA initialization and de-initialization.

Further details on the service primitives for local management can be found in the on-line manual pages.

Connection-Mode Service Primitives

This section describes the service primitives that support the connection-mode service of the data link layer. These primitives support the connection establishment, connection-mode data transfer, and connection release services described earlier.

Connection Establishment

In the connection establishment model, the calling DLS user initiates a request for a connection, and the called DLS user receives each request and either accepts or rejects it. In the simplest form (single-threaded), the called DLS user is passed a connect indication and the DLS provider holds any subsequent indications until a response for the current outstanding indication is received. At most one connect indication is outstanding at any time.

DLPI also enables a called DLS user to multi-thread connect indications and responses. This capability is desirable, for example, when imposing a priority scheme on all DLS users attempting to establish a connection. The DLS provider will pass all connect indications to the called DLS user (up to some pre-established limit as set by `DL_BIND_REQ` and `DL_BIND_ACK`). The called DLS user may then respond to the requests in any order.

To support multi-threading, a correlation value is needed to associate responses with the appropriate connect indication. A correlation value is contained in each `DL_CONNECT_IND`, and the DLS user must use this value in the `DL_CONNECT_RES` or `DL_DISCONNECT_REQ` primitive used to accept or reject the connect request. The DLS user can also receive a `DL_DISCONNECT_IND` with a correlation value when the calling DLS user or the DLS provider abort a connect request.

Once a connection has been accepted or rejected, the correlation value has no meaning to a DLS user. The DLS provider may reuse the correlation value in another `DL_CONNECT_IND`. Thus, the lifetime of a correlation value is the duration of the connection establishment phase, and as good programming practice it should not be used for any other purpose by the DLS provider.

The DLS provider assigns the correlation value for each connect indication. Correlation values must be unique among all outstanding connect indications on a given stream. The values may, but need not, be unique across all streams to the DLS provider. The correlation value must be a positive, non-zero value. There is no implied sequencing of connect indications using the correlation value; the values do not have to increase sequentially for each new connect indication.

Further details on the service primitives for connection-mode can be found in the on-line manual pages.

Connectionless-Mode Service Primitives

This section describes the primitives that support the connectionless-mode service of the data link layer. These primitives support the connectionless data transfer service described earlier.

Further details on the service primitives for connectionless-mode can be found in the on-line manual pages.

XID and TEST Operations Primitives

This section describes the service primitives that support the XID and TEST operations. The DLS User can issue these primitives to the DLS Provider requesting the provider to send an `XID` or a `TEST` frame. On receipt of an `XID` or `TEST` frame from the remote side, the DLS Provider can send the appropriate indication to the User.

Further details on the service primitives for XID and TEST operations can be found in the on-line manual pages.

Quality of Data Link Service

The quality of data link service is defined by the term *Quality of Service* (QOS), and describes certain characteristics of transmission between two DLS users. These characteristics are attributable solely to the DLS provider, but are observable by the DLS users. The visibility of QOS characteristics enables a DLS user to determine, and possibly negotiate, the characteristics of transmission needed to communicate with the remote DLS user. Quality of service characteristics apply to both the connection and connectionless modes of service. The semantics for each mode are discussed below.

Connection-Mode Service

Quality of Service (QOS) refers to certain characteristics of a data link connection as observed between the connection endpoints. QOS describes the specific aspects of a data link connection that are attributable to the DLS provider.

QOS is defined in terms of QOS parameters. The parameters give DLS users a means of specifying their needs. These parameters are divided into two groups, based on how their values are determined:

- QOS parameters that are negotiated on a per-connection basis during connection establishment; and
- QOS parameters that are not negotiated during connection establishment. The values are determined or known through other methods, usually administrative.

The QOS parameters that can be negotiated during connection establishment are: throughput, transit delay, priority, and protection. The QOS parameters for throughput and transit delay are negotiated end-to-end between the two DLS users and the DLS provider. The QOS parameters for priority and protection are negotiated locally by each DLS user with the DLS provider. The QOS parameters that cannot be negotiated are residual error rate and resilience. "Procedures for QOS Negotiation and Selection"describes the rules for QOS negotiation.

Once the connection is established, the agreed QOS values are not renegotiated at any point. There is no guarantee by any DLS provider that the original QOS values will be maintained, and the DLS users are not informed if QOS changes. The DLS provider also need only record those QOS values selected at connection establishment for return in response to the `DL_INFO_REQ` primitive.

QOS for Connectionless-Mode Services

The QOS for connectionless-mode and acknowledged connectionless-mode service refers to characteristics of the data link layer between two DLSAPs, attributable to the DLS provider. The QOS applied to each `DL_UNITDATA_REQ`/`DL_DATA_ACK_REQ` primitive may be independent of the QOS applied to preceding and following `DL_UNITDATA_REQ`/`DL_DATA_ACK_REQ` primitives. QOS cannot be negotiated between two DLS users as in the connection-mode service.

Every `DL_UNITDATA_REQ`/`DL_DATA_ACK_REQ` primitive may have certain QOS values associated with it. The supported range of QOS parameter values is made known to the DLS user in response to the `DL_INFO_REQ` primitive. The DLS user may select specific QOS parameter values to be associated with subsequent data unit transmissions using the `DL_UDQOS_REQ` primitive. This selection is a strictly local management function. If different QOS values are to be associated with each transmission, `DL_UDQOS_REQ` may be issued to alter those values before each `DL_UNITDATA_REQ`/`DL_DATA_ACK_REQ` is issued.

QOS Parameter Definitions

This section describes the quality of service parameters supported by DLPI for both connection-mode and connectionless-mode services. The following table summarizes the supported parameters. It indicates to which service mode (connection, connectionless, or both) the parameter applies. For those parameters supported by the connection-mode service, the table also indicates whether the parameter value is negotiated during connection establishment. If so, the table further indicates whether the QOS values are negotiated end-to-end among both DLS users and the DLS provider, or locally for each DLS user independently with the DLS provider.

Parameter	Service Mode	Negotiation
throughput	connection	end-to-end
transit delay	both	end-to-end
priority	both	local
protection	both	local
residual error rate	both	none
resilience	connection	none

Throughput

Throughput is a connection-mode QOS parameter that has end-to-end significance. It is defined as the total number of DLSDU bits successfully transferred by a `DL_DATA_REQ/DL_DATA_IND` primitive sequence divided by the input/output time, in seconds, for that sequence. Successful transfer of a DLSDU is defined to occur when the DLSDU is delivered to the intended user without error, in proper sequence, and before connection termination by the receiving DLS user.

The input/output time for a `DL_DATA_REQ/DL_DATA_IND` primitive sequence is the greater of:

- the time between the first and last `DL_DATA_REQ` in a sequence; and
- the time between the first and last `DL_DATA_IND` in the sequence.

Throughput is only meaningful for a sequence of complete DLSDUs.

Throughput is specified and negotiated for the transmit and receive directions independently at connection establishment. The throughput specification defines the target and minimum acceptable values for a connection. Each specification is an average rate.

The DLS user can delay the receipt or sending of DLSDUs. The delay caused by a DLS user is not included in calculating the average throughput values.

Parameter Format

```
typedef struct {
    long    dl_target_value;
    long    dl_accept_value;
} dl_through_t;
```

This typedef is used to negotiate the transmit and receive throughput values.

`dl_target_value`	specifies the desired throughput value for the connection in bits/second.
`dl_accept_value`	specifies the minimum acceptable throughput value for the connection in bits/second.

Transit Delay

Connection and connectionless modes can specify a transit delay, which indicates the elapsed time between a `DL_DATA_REQ` or `DL_UNITDATA_REQ` primitive and the corresponding `DL_DATA_IND` or `DL_UNITDATA_IND` primitive. The elapsed time is only computed for DLSDUs successfully transferred, as described previously for throughput.

In connection mode, transit delay is negotiated on an end-to-end basis during connection establishment. For each connection, transit delay is negotiated for the transmit and receive directions separately by specifying the target value and maximum acceptable value. For connectionless-mode service, a DLS user selects a particular value within the supported range using the `DL_UDQOS_REQ` primitive, and the value may be changed for each DLSDU submitted for connectionless transmission.

The transit delay for an individual DLSDU may be increased if the receiving DLS user flow controls the interface. The average and maximum transit delay values exclude any DLS user flow control of the interface. The values are specified in milliseconds, and assume a DLSDU size of 128 octets.

Parameter Format

```
typedef struct {
    long    dl_target_value;
    long    dl_accept_value;
} dl_transdelay_t;
```

This typedef is used to negotiate the transmit and receive transit delay values.

`dl_target_value`	specifies the desired transit delay value.
`dl_accept_value`	specifies the maximum acceptable transit delay value.

Priority

Priority is negotiated locally between each DLS user and the DLS provider in connection-mode service, and can also be specified for connectionless-mode service. The specification of priority is concerned with the relationship between connections or the relationship between connectionless data transfer requests. The parameter specifies the relative importance of a connection with respect to:

- the order in which connections are to have their QOS degraded, if necessary; and
- the order in which connections are to be released to recover resources, if necessary;

For connectionless-mode service, the parameter specifies the relative importance of unitdata objects with respect to gaining use of shared resources.

For connection-mode service, each DLS user negotiates a particular priority value with the DLS provider during connection establishment. The value is specified by a minimum and a maximum within a given range. For connectionless-mode service, a DLS user selects a particular priority value within the supported range using the `DL_UDQOS_REQ` primitive, and the value may be changed for each DLSDU submitted for connectionless transmission.

This parameter only has meaning in the context of some management entity or structure able to judge relative importance. The priority has local significance only, with a value of zero being the highest priority and `100` being the lowest priority.

Parameter Format

```
typedef struct {
    long    dl_min;
    long    dl_max;
} dl_priority_t;
```

`dl_min`	specifies the minimum acceptable priority.
`dl_max`	specifies the maximum desired priority.

Protection

Protection is negotiated locally between each DLS user and the DLS provider in connection-mode service, and can also be specified for connectionless-mode service. Protection is the extent to which a DLS provider attempts to prevent unauthorized monitoring or manipulation of DLS user-originated information. Protection is specified by a minimum and maximum protection option within the following range of possible protection options:

`DL_NONE`	DLS provider will not protect any DLS user data
`DL_MONITOR`	DLS provider will protect against passive monitoring
`DL_MAXIMUM`	DLS provider will protect against modification, replay, addition, or deletion of DLS user data

For connection-mode service, each DLS user negotiates a particular value with the DLS provider during connection establishment. The value is specified by a minimum and a maximum within a given range. For connectionless-mode service, a DLS user selects a particular value within the supported range using the `DL_UDQOS_REQ` primitive, and the value may be changed for each DLSDU submitted for connectionless transmission. Protection has local significance only.

Parameter Format

```
typedef struct {
    long    dl_min;
    long    dl_max;
} dl_protect_t;
```

`dl_min`	specifies the minimum acceptable protection.
`dl_max`	specifies the maximum desired protection.

Residual Error Rate

Residual error rate is the ratio of total incorrect, lost and duplicate DLSDUs to the total DLSDUs transferred between DLS users during a period of time. The relationship between these quantities is defined below:

where

$DLSDU_{tot}$	= total DLSDUs transferred, which is the total of $DLSDU_l$, $DLSDU_i$, $DLSDU_e$, and correctly received DLSDUs.
$DLSDU_e$	= DLSDUs received 2 or more times.
$DLSDU_i$	= incorrectly received DLSDUs.
$DLSDU_l$	= DLSDUs sent, but not received.

Parameter Format

```
long    dl_residual_error;
```

The residual error value is scaled by a factor of 1,000,000, since the parameter is stored as a long integer in the QOS data structures. Residual error rate is not a negotiated QOS parameter. Its value is determined by procedures outside the definition of DLPI. It is assumed to be set by an administrative mechanism, which is informed of the value by network management.

Resilience

Resilience is meaningful in connection mode only, and represents the probability of either DLS provider-initiated disconnects or DLS provider-initiated resets during a time interval of 10,000 seconds on a connection.

Resilience is not a negotiated QOS parameter. Its value is determined by procedures outside the definition of DLPI. It is assumed to be set by an administrative mechanism, which is informed of the value by network management.

Parameter Format

```
typedef struct {
    long    dl_disc_prob;
    long    dl_reset_prob;
} dl_resilience_t;
```

`dl_disc_prob`	specifies the probability of receiving a provider-initiated disconnect, scaled by 10000.
`dl_reset_prob`	specifies the probability of receiving a provider-initiated reset, scaled by 10000.

QOS Data Structures

To simplify the definition of the primitives containing QOS parameters and the discussion of QOS negotiation, the QOS parameters are organized into four structures. This section defines the structures and indicates which structures apply to which primitives.

Each structure is tagged with a type field contained in the first four bytes of the structure, similar to the tagging of primitives. The type field has been defined because of the current volatility of QOS parameter definition within the international standards bodies. If new QOS parameter sets are defined in the future for the data link layer, the type field will enable DLPI to accommodate these sets without breaking existing DLS user or provider implementations. If a DLS provider receives a structure type that it does not understand in a given primitive, the error `DL_BADQOSTYPE` should be returned to the DLS user in a `DL_ERROR_ACK` primitive.

Currently the following QOS structure types are defined:

`DL_QOS_RANGE1`	QOS range structure for connection-mode service for Issue 1 of DLPI
`DL_QOS_CO_SEL1`	QOS selection structure for connection-mode service for Issue 1 of DLPI
`DL_QOS_CL_RANGE1`	QOS range structure for connectionless-mode service for Issue 1 of DLPI
`DL_QOS_CL_SEL1`	QOS selection structure for connectionless-mode service for Issue 1 of DLPI

The syntax and semantics of each structure type is presented in the remainder of this section.

Further details on the structures used for Quality of Service can be found in the on-line manual pages.

Procedures for QOS Negotiation and Selection

This section describes the methods used for negotiating and/or selecting QOS parameter values. In the connection-mode service, some QOS parameter values may be negotiated during connection establishment. For connectionless-mode service, parameter values may be selected for subsequent data transmission.

Throughout this section, two special QOS values are referenced. These are defined for all the parameters used in QOS negotiation and selection. The values are:

`DL_UNKNOWN`	This value indicates that the DLS provider does not know the value for the field or does not support that parameter.
`DL_QOS_DONT_CARE`	This value indicates that the DLS user does not care to what value the QOS parameter is set.

These values are used to distinguish between DLS providers that support and negotiate QOS parameters and those that cannot. The following sections include the interpretation of these values during QOS negotiation and selection.

Connection-Mode QOS Negotiation

The current connection-mode QOS parameters can be divided into three types as follows:

- Those that are negotiated end-to-end between peer DLS users and the DLS provider during connection establishment (throughput and transit delay);
- those that are negotiated locally between each DLS user and the DLS provider during connection establishment (priority and protection); and
- those that cannot be negotiated (residual error rate and resilience).

The rules for processing these three types of parameters during connection establishment are described in this section.

The current definition of most existing data link protocols does not describe a mechanism for negotiating QOS parameters during connection establishment. As such, DLPI does not require every DLS provider implementation to support QOS negotiation. If a given DLS provider implementation cannot support QOS negotiation, two alternatives are available:

- The DLS provider may specify that any or all QOS parameters are unknown. This is indicated to the DLS user in the `DL_INFO_ACK`, where the values in the QOS range field (indicated by `dl_qos_range_length` and `dl_qos_range_offset`) and the current QOS field (indicated by `dl_qos_length` and `dl_qos_offset`) of this primitive are set to `DL_UNKNOWN`. This value will also be indicated on the `DL_CONNECT_IND` and `DL_CONNECT_CON` primitives. If the DLS provider does not support any QOS parameters, the QOS length field may be set to zero in each of these of these primitives.
- The DLS provider may interpret QOS parameters with strictly local significance, and their values in the `DL_CONNECT_IND` primitive will be set to `DL_UNKNOWN`.

A DLS user need not select a specific value for each QOS parameter. The special QOS parameter value, `DL_QOS_DONT_CARE`, is used if the DLS user does not care what quality of service is provided for a particular parameter. The negotiation procedures presented below explain the exact semantics of this value during connection establishment.

If QOS parameters are supported by the DLS provider, the provider will define a set of default QOS parameter values that are used whenever `DL_QOS_DONT_CARE` is specified for a QOS parameter value. These default values can be defined for all DLS users or can be defined on a per DLS user basis. The default parameter value set is returned in the QOS field (indicated by `dl_qos_length` and `dl_qos_offset`) of the `DL_INFO_ACK` before a DLS user negotiates QOS parameter values.

DLS provider addendum documentation must describe the known ranges of support for the QOS parameters and the default values, and also specify whether they are used in a local manner only.

The following procedures are used to negotiate QOS parameter values during connection establishment.

- The DL_CONNECT_REQ specifies the DLS user's desired range of QOS values in the dl_qos_co_range1_t structure. The target and least-acceptable values are specified for throughput and transit delay, as described in “Throughput,” and “Transit Delay.” The target value is the value desired by the calling DLS user for the QOS parameters. The least-acceptable value is the lowest value the calling user will accept. These values are specified separately for both the transmit and receive directions of the connection.

  If either value is set to DL_QOS_DONT_CARE the DLS provider will supply a default value, subject to the following consistency constraints:

  - If DL_QOS_DONT_CARE is specified for the target value, the value chosen by the DLS provider may not be less than the least-acceptable value.
  - If DL_QOS_DONT_CARE is specified for the least-acceptable value, the value set by the DLS provider cannot be greater than the target value.
  - If DL_QOS_DONT_CARE is specified for both the target and least-acceptable value, the DLS provider is free to select any value, without constraint, for the target and least-acceptable values.

For priority and protection, the DL_CONNECT_REQ specifies a minimum and maximum desired value as defined in “Priority,” and “Protection.” As with throughput and transit delay, the DLS user may specify a value of DL_QOS_DONT_CARE for either the minimum or maximum value. The DLS provider will interpret this value subject to the following consistency constraints:

- If DL_QOS_DONT_CARE is specified for the maximum value, the value chosen by the DLS provider may not be less than the minimum value.
- If DL_QOS_DONT_CARE is specified for the minimum value, the value set by the DLS provider cannot be greater than the maximum value.
- If DL_QOS_DONT_CARE is specified for both the minimum and maximum values, the DLS provider is free to select any value, without constraint, for the maximum and minimum values.

The values of the residual error rate and resilience parameters in the DL_CONNECT_REQ have no meaning and are ignored by the DLS provider.

If the value of dl_qos_length in the DL_CONNECT_REQ is set to zero by the DLS user, the DLS provider should treat all QOS parameter values as if they were set to DL_QOS_DONT_CARE, selecting any value in its supported range.

If the DLS provider cannot support throughput, transit delay, priority, and protection values within the ranges specified in the DL_CONNECT_REQ, a DL_DISCONNECT_IND should be sent to the calling DLS user.

If the requested ranges of values for throughput and transit delay in the DL_CONNECT_REQ are acceptable to the DLS provider, the QOS parameters will be adjusted to values the DLS provider will support. Only the target value may be adjusted, and it is set to a value the DLS provider is willing to provide (which may be of lower QOS than the target value). The least-acceptable value cannot be modified. The updated QOS range is then sent to the called DLS user in the dl_qos_co_range1_t structure of the DL_CONNECT_IND, where it is interpreted as the available range of service.

If the requested range of values for priority and protection in the DL_CONNECT_REQ is acceptable to the DLS provider, an appropriate value within the range is selected and saved for each parameter; these selected values will be returned to the DLS user in the corresponding DL_CONNECT_CON primitive. Because priority and protection are negotiated locally, the DL_CONNECT_IND will not contain values selected during negotiation with the calling DLS user. Instead, the DLS provider will offer a range of values in the DL_CONNECT_IND that will be supported locally for the called DLS user.

The DLS provider will also include the supported values for residual error rate and resilience in the DL_CONNECT_IND that is passed to the called DLS user.

If the DLS provider does not support negotiation of throughput, transit delay, priority, or protection, a value of DL_UNKNOWN should be set in the least-acceptable, target, minimum, and maximum value fields of the DL_CONNECT_IND. Also, if the DLS provider does not support any particular QOS parameter, DL_UNKNOWN should be specified in all value fields for that parameter. If the DLS provider does not support any QOS parameters, the value of dl_qos_length may be set to zero in the DL_CONNECT_IND.

After receiving the DL_CONNECT_IND, the called DLS user examines the QOS parameter values and selects a specific value from the proffered range of the throughput, transit delay, priority, and protection parameters. If the called DLS user does not agree on values in the given range, the connection should be refused with a DL_DISCONNECT_REQ primitive. Otherwise, the selected values are returned to the DLS provider in the dl_qos_co_sel1_t structure of the DL_CONNECT_RES primitive.

The values of residual error rate and resilience in the DL_CONNECT_RES are ignored by the DLS provider. These parameters may not be negotiated by the called DLS user. The selected values of throughput and transit delay are meaningful, however, and are adopted for the connection by the DLS provider. Similarly, the selected priority and protection values are adopted with local significance for the called DLS user.

If the user specifies DL_QOS_DONT_CARE for either throughput, transit delay, priority, or protection on the DL_CONNECT_RES, the DLS provider will select a value from the range specified for that parameter in the DL_CONNECT_IND primitive. Also, a value of zero in the dl_qos_length field of the DL_CONNECT_RES is equivalent to DL_QOS_DONT_CARE for all QOS parameters.

After completion of connection establishment, the values of throughput and transit delay as selected by the called DLS user are returned to the calling DLS user in the dl_qos_co_sel1_t structure of the DL_CONNECT_CON primitive. The values of priority and protection that were selected by the DLS provider from the range indicated in the DL_CONNECT_REQ will also be returned in the DL_CONNECT_CON. This primitive will also contain the values of residual error rate and resilience associated with the newly established connection. The DLS provider also saves the negotiated QOS parameter values for the connection, so that they may be returned in response to a DL_INFO_REQ primitive.

As with DL_CONNECT_IND, if the DLS provider does not support negotiation of throughput, transit delay, priority, or protection, a value of DL_UNKNOWN should be returned in the selected value fields. Furthermore, if the DLS provider does not support any particular

QOS parameter, DL_UNKNOWN should be specified in all value fields for that parameter, or the value of dl_qos_length may be set to zero in the DL_CONNECT_CON primitive.

Connectionless-Mode QOS Selection

This section describes the procedures for selecting QOS parameter values that will be associated with the transmission of connectionless data.

As with connection-mode protocols, the current definition of most existing connectionless data link protocols does not define a quality of service concept. As such, DLPI does not require every DLS provider implementation to support QOS parameter selection. The DLS provider may specify that any or all QOS parameters are unsupported. This is indicated to the DLS user in the DL_INFO_ACK, where the values in the supported range field (indicated by dl_qos_range_length and dl_qos_range_offset) and the current QOS field (indicated by dl_qos_length and dl_qos_offset) of this primitive are set to DL_UNKNOWN. If the DLS provider supports no QOS parameters, the QOS length fields in the DL_INFO_ACK may be set to zero.

If the DLS provider supports QOS parameter selection, the DL_INFO_ACK primitive will specify the supported range of parameter values for transit delay, priority, protection and residual error rate. Default values are also returned in the DL_INFO_ACK.

For each DL_UNITDATA_REQ, the DLS provider should apply the currently selected QOS parameter values to the transmission. If no values have been selected, the default values should be used.

At any point during data transfer, the DLS user may issue a DL_UDQOS_REQ primitive to select new values for the transit delay, priority, and protection parameters. These values are selected using the dl_qos_cl_sel1_t structure. The residual error rate parameter is ignored by this primitive and cannot be set by a DLS user.

In the DL_UDQOS_REQ, the DLS user need not require a specific value for every QOS parameter. DL_QOS_DONT_CARE may be specified if the DLS user does not care what quality of service is provided for a particular parameter. When specified, the DLS provider should retain the current (or default if no previous selection has occurred) value for that parameter.

Allowable Sequence of DLPI Primitives

This section presents the allowable sequence of DLPI primitives. The sequence is described using a state transition table that defines possible states as viewed by the DLS user. The state transition table describes transitions based on the current state of the interface and a given DLPI event. Each transition consists of a state change and possibly an interface action. The states, events, and related transition actions are described below, followed by the state transition table itself. Table 11-2 describes the states associated with DLPI. It presents the state name used in the state transition table, the corresponding DLPI state name, a brief description of the state, and an indication of whether the state is valid for connection-oriented data link service (DL_CODLS), connectionless data link service (DL_CLDLS), acknowledged connectionless data link service (DL_ACLDLS), or all.

Table 11-2. DLPI States

State	DLPI State	Description	Service Type
0) UNATTACHED	DL_UNATTACHED	Stream opened but PPA not attached	all
1) ATTACH PEND	DL_ATTACH_PENDING	The DLS user is waiting for an acknowledgement of a DL_ATTACH_REQ	all
2) DETACH PEND	DL_DETACH_PENDING	The DLS user is waiting for an acknowledgement of a DL_DETACH_REQ	all
3) UNBOUND	DL_UNBOUND	Stream is attached but not bound to a DLSAP	all
4) BIND PEND	DL_BIND_PENDING	The DLS user is waiting for an acknowledgement of a DL_BIND_REQ	all
5) UNBIND PEND	DL_UNBIND_PENDING	The DLS user is waiting for an acknowledgement of a DL_UNBIND_REQ	all
6) IDLE	DL_IDLE	The stream is bound and activated for use - connection establishment or connectionless data transfer may take place	all
7) UDQOS PEND	DL_UDQOS_PENDING	The DLS user is waiting for an acknowledgement of a DL_UDQOS_REQ	DL_CLDLS
8) OUTCON PEND	DL_OUTCON_PENDING	An outgoing connection is pending - the DLS user is waiting for a DL_CONNECT_CON	DL_CODLS
9) INCON PEND	DL_INCON_PENDING	An incoming connection is pending - the DLS provider is waiting for a DL_CONNECT_RES	DL_CODLS
10) CONN_RES PEND	DL_CONN_RES_PENDING	The DLS user is waiting for an acknowledgement of a DL_CONNECT_RES	DL_CODLS
11) DATAXFER	DL_DATAXFER	Connection-mode data transfer may take place	DL_CODLS
12) USER RESET PEND	DL_USER_RESET_PENDING	A user-initiated reset is pending - the DLS user is waiting for a DL_RESET_CON	DL_CODLS
13) PROV RESET PEND	DL_PROV_RESET_PENDING	A provider-initiated reset is pending - the DLS provider is waiting for a DL_RESET_RES	DL_CODLS
14) RESET_RES PEND	DL_RESET_RES_PENDING	The DLS user is waiting for an acknowledgement of a DL_RESET_RES	DL_CODLS
15) DISCON 8 PEND	DL_DISCON8_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_OUTCON_PENDING state	DL_CODLS
16) DISCON 9 PEND	DL_DISCON9_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_INCON_PENDING state	DL_CODLS

Table 11-2. DLPI States (Cont.)

State	DLPI State	Description	Service Type
17) DISCON 11 PEND	DL_DISCON11_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_DATAXFER state	DL_CODLS
18) DISCON 12 PEND	DL_DISCON12_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_USER_RESET_PENDING state	DL_CODLS
19) DISCON 13 PEND	DL_DISCON13_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_PROV_RESET_PENDING state	DL_CODLS
20) SUBS_BIND PEND	DL_SUBS_BIND_REQ	The DLS user is waiting for an acknowledgement of a DL_SUBS_BIND_REQ	all

Variables and Actions for State Transition Table

Table 11-3 and Table 11-4 describe variables and actions used to describe the DLPI state transitions.

The variables are used to distinguish various uses of the same DLPI primitive. For example, a `DL_CONNECT_RES` causes a different state transition depending on the current number of outstanding connect indications. To distinguish these different connect response events, a variable is used to track the number of outstanding connect indications.

Table 11-3. State Transition Table

Variable	Description
`token`	The token contained in a `DL_CONNECT_RES` that indicates on which stream the connection will be established. A value of zero indicates that the connection will be established on the stream where the `DL_CONNECT_IND` arrived. A non-zero value indicates the connection will be passed to another stream.
`outcnt`	Number of outstanding connect indications - those to which the DLS user has not responded. Actions in the state tables that manipulate this value may be disregarded when providing connectionless service.

DPLI State Transition Table Variables

The actions represent steps the DLS provider must take during certain state transitions to maintain the interface state. When an action is indicated in the state transition table, the DLS provider should change the state as indicated and perform the specified action.

Table 11-4. Variables

Action	Description
1	`outcnt = outcnt + 1;`
2	`outcnt = outcnt - 1;`
3	Pass connection to the stream indicated by the token in the `DL_CONNECT_RES` primitive

DLPI User-Originated Events

Table 11-5 describes events initiated by the DLS user that correspond to the various request and response primitives of DLPI. The table presents the event name used in the state transition table, a brief description of the event (including the corresponding DLPI primitive), and an indication of whether the event is valid for connection-oriented data link service (`DL_CODLS`), connectionless data link service (`DL_CLDLS`), acknowledged connectionless data link service (`DL_ACLDLS`), or all.

Table 11-5. Events

FSM Event	Description	Service Type
`ATTACH_REQ`	`DL_ATTACH_REQ` primitive	all
`DETACH_REQ`	`DL_DETACH_REQ` primitive	all
`BIND_REQ`	`DL_BIND_REQ` primitive	all
`SUBS_BIND_REQ`	`DL_SUBS_BIND_REQ` primitive	all
`UNBIND_REQ`	`DL_UNBIND_REQ` primitive	all
`UNITDATA_REQ`	`DL_UNITDATA_REQ` primitive	`DL_CLDLS`
`UDQOS_REQ`	`DL_UDQOS_REQ` primitive	`DL_CLDLS`
`CONNECT_REQ`	`DL_CONNECT_REQ` primitive	`DL_CODLS`
`CONNECT_RES`	`DL_CONNECT_RES` primitive	`DL_CODLS`
`PASS_CONN`	Received a passed connection from a `DL_CONNECT_RES` primitive	`DL_CODLS`
`DISCON_REQ`	`DL_DISCONNECT_REQ` primitive	`DL_CODLS`

Table 11-5. Events (Cont.)

FSM Event	Description	Service Type
DATA_REQ	DL_DATA_REQ primitive	DL_CODLS
RESET_REQ	DL_RESET_REQ primitive	DL_CODLS
RESET_RES	DL_RESET_RES primitive	DL_CODLS

DLPI Provider-Originated Events

Table 11-6 describes the events initiated by the DLS provider that correspond to the various indication, confirmation, and acknowledgment primitives of DLPI. The table presents the event name used in the state transition table, a brief description of the event (including the corresponding DLPI primitive), and an indication of whether the event is valid for connection-oriented data link service (DL_CODLS), connectionless data link service (DL_CLDLS) acknowledged connectionless data link service (DL_ACLDLS), or all.

Table 11-6. DLPI Provider Events

FSM Event	Description	Service Type
BIND_ACK	DL_BIND_ACK primitive	all
SUBS_BIND_ACK	DL_SUBS_BIND_ACK primitive	all
UNITDATA_IND	DL_UNITDATA_IND primitive	DL_CLDLS
UDERROR_IND	DL_UDERROR_IND primitive	DL_CLDLS
CONNECT_IND	DL_CONNECT_IND primitive	DL_CODLS
CONNECT_CON	DL_CONNECT_CON primitive	DL_CODLS
DISCON_IND1	DL_DISCONNECT_IND primitive when outcnt == 0	DL_CODLS
DISCON_IND2	DL_DISCONNECT_IND primitive when outcnt == 1	DL_CODLS
DISCON_IND3	DL_DISCONNECT_IND primitive when outcnt > 1	DL_CODLS
DATA_IND	DL_DATA_IND primitive	DL_CODLS
RESET_IND	DL_RESET_IND primitive	DL_CODLS
RESET_CON	DL_RESET_CON primitive	DL_CODLS
OK_ACK1	DL_OK_ACK primitive when outcnt == 0	all
OK_ACK2	DL_OK_ACK primitive when outcnt == 1 and token == 0	DL_CODLS

Table 11-6. DLPI Provider Events (Cont.)

FSM Event	Description	Service Type
OK_ACK3	DL_OK_ACK primitive when outcnt == 1 and token != 0	DL_CODLS
OK_ACK4	DL_OK_ACK primitive when outcnt > 1 and token != 0	DL_CODLS
ERROR_ACK	DL_ERROR_ACK	all

DLPI State Transition Table

Table 11-7 through Table 11-11 describe the DLPI state transitions. Each column represents a state of DLPI, as shown in Table 11-2, and each row represents a DLPI event, as shown in Table 11-4 and Table 11-5. The intersecting transition cell defines the resulting state transition, or next state, and associated actions, if any, that must be executed by the DLS provider to maintain the interface state. Each cell may contain the following:

-	This transition cannot occur.
n	The current input results in a transition to state "n."
n [*a*]	The list of actions "*a*" should be executed following the specified state transition "n" (see table 4 for actions).

The DL_INFO_REQ, DL_INFO_ACK, DL_TOKEN_REQ, and DL_TOKEN_ACK primitives are excluded from the state transition table because they can be issued from many states and, when fully processed, do not cause a state transition to occur. However, the DLS user may not issue a DL_INFO_REQ or DL_TOKEN_REQ if any local acknowledgments are pending. In other words, these two primitives may not be issued until the DLS user receives the acknowledgment for any previously issued primitive that is expecting local positive acknowledgment. Thus, these primitives may not be issued from the DL_ATTACH_PENDING, DL_DETACH_PENDING, DL_BIND_PENDING, DL_SUBS_BIND_PENDING, DL_UNBIND_PENDING, DL_UDQOS_PENDING, DL_CONN_RES_PENDING,DL_RESET_RES_PENDING,DL_DISCON8_PENDING, DL_DISCON9_PENDING, DL_DISCON11_PENDING, DL_DISCON12_PENDING, or DL_DISCON13_PENDING states. Failure to comply by this restriction may result in loss of primitives at the stream head if the DLS user is a user process. Once a DL_INFO_REQ or DL_TOKEN_REQ has been issued, the DLS provider must respond with the appropriate acknowledgment primitive.

The following rules apply to the maintenance of DLPI state:

- The DLS provider is responsible for keeping a record of the state of the interface as viewed by the DLS user, to be returned in the DL_INFO_ACK.
- The DLS provider may never generate a primitive that places the interface out of state.

NOTE

This would correspond to a minus (`-`) cell entry in the state transition table.

- If the DLS provider generates a STREAMS `M_ERROR` message upstream, it should free any further primitives processed by it's write side `put` or `service` procedure.
- The close of a stream is considered an abortive action by the DLS user, and may be executed from any state. The DLS provider must issue appropriate indications to the remote DLS user when a close occurs. For example, if the DLPI state is `DL_DATAXFER`, a `DL_DISCONNECT_IND` should be sent to the remote DLS user. The DLS provider should free any resources associated with that stream and reset the stream to its unopened condition.

The following points clarify the state transition table.

- If the DLS provider supports connection-mode service, the value of the `outcnt` state variable must be initialized to zero for each stream when that stream is first opened.
- The initial and final state for a *style* 2 DLS provider is `DL_UNATTACHED`. However, because a *style* 1 DLS provider implicitly attaches a PPA to a stream when it is opened, the initial and final DLPI state for a *style* 1 provider is `DL_UNBOUND`. The DLS user should not issue `DL_ATTACH_REQ` or `DL_DETACH_REQ` primitives to a *style* 1 DLS provider.
- A DLS provider may have multiple connect indications outstanding at one time, which indicates that the DLS user has not responded to them. See “Connection Establishment Service.” As the state transition table points out, the stream on which those indications are outstanding will remain in the `DL_INCON_PENDING` state until the DLS provider receives a response for all indications.
- The DLPI state associated with a given stream may be transferred to another stream only when the `DL_CONNECT_RES` primitive indicates this behavior. In this example, the responding stream (where the connection will be established) must be in the `DL_IDLE` state. This state transition is indicated by the `PASS_CONN` event in Table 11-10.
- The labeling of the states `DL_PROV_RESET_PENDING` and `DL_USER_RESET_PENDING` indicate the party that started the local interaction, and does not necessarily indicate the originator of the reset procedure.
- A `DL_DATA_REQ` primitive received by the DLS provider in the state `DL_PROV_RESET_PENDING`, for example, after a `DL_RESET_IND` has been passed to the DLS user, or the state `DL_IDLE`, for example, after a data link connection has been released, should be discarded by the DLS provider.
- A `DL_DATA_IND` primitive received by the DLS user after the user has issued a `DL_RESET_REQ` should be discarded.

To ensure accurate processing of DLPI primitives, the DLS provider must adhere to the following rules about the receipt and generation of STREAMS `M_FLUSH` messages during various state transitions.

- The DLS provider must be ready to receive `M_FLUSH` messages from upstream and flush it's queues as specified in the message.
- The DLS provider must issue an `M_FLUSH` message upstream to flush both the read and write queues after receiving a successful `DL_UNBIND_REQ` primitive but before issuing the `DL_OK_ACK`.
- If an incoming disconnect occurs when the interface is in the `DL_DATAXFER`, `DL_USER_RESET_PENDING`, or `DL_PROV_RESET_PENDING` state, the DLS provider must send up an `M_FLUSH` message to flush both the read and write queues before sending up a `DL_DISCONNECT_IND`.
- If a `DL_DISCONNECT_REQ` is issued in the `DL_DATAXFER`, `DL_USER_RESET_PENDING`, or `DL_PROV_RESET_PENDING` states, the DLS provider must issue an `M_FLUSH` message upstream to flush both the read and write queues after receiving the successful `DL_DISCONNECT_REQ` but before issuing the `DL_OK_ACK`.
- If a reset occurs when the interface is in the `DL_DATAXFER` or `DL_USER_RESET_PENDING` state, the DLS provider must send up an `M_FLUSH` message to flush both the read and write queues before sending up a `DL_RESET_IND` or `DL_RESET_CON`.

Common Local Management Phase

Table 11-7 presents the allowed sequence of DLPI primitives for the common local management phase of communication.

Table 11-7. Local Management Phase

STATES	UNATTACHED	ATTACH	DETACH	UNBOUND	BIND	UNBIND	IDLE	SUBS_BIND
		PEND	PEND		PEND	PEND		PEND
EVENTS	0	1	2	3	4	5	6	20
ATTACH_REQ	1	-	-	-	-	-	-	-
DETACH_REQ	-	-	-	2	-	-	-	-
BIND_REQ	-	-	-	4	-	-	-	-
BIND_ACK	-	-	-	-	6	-	-	-
SUBS_BIND_REQ	-	-	-	-	-	-	20	-
SUBS_BIND_ACK	-	-	-	-	-	-	-	6

Table 11-7. Local Management Phase (Cont.)

STATES	UNATTACHED	ATTACH PEND	DETACH PEND	UNBOUND	BIND PEND	UNBIND PEND	IDLE	SUBS_BIND PEND
EVENTS	0	1	2	3	4	5	6	20
UNBIND_REQ	-	-	-	-	-	-	5	-
OK_ACK1	-	3	0	-	-	3	-	-
ERROR_ACK	-	0	3	-	3	6	-	-

Table 11-8 presents the allowed sequence of DLPI primitives for the connectionless data transfer phase.

Table 11-8. Connectionless-Mode Data Transfer Phase

STATES	IDLE	UDQOS PEND
EVENTS	6	7
UDQOS_REQ	**7**	**-**
OK_ACK1	**-**	**6**
ERROR_ACK	**-**	**6**
UNITDATA_REQ	**6**	**-**
UNITDATA_IND	**6**	**-**
UDERROR_IND	**6**	**-**

Table 11-9. Acknowledged Connectionless-Mode Data Transfer Phase

STATES / EVENTS	IDLE 6	UDQOS PEND 7
UDQOS_REQ	7	-
OK_ACK1	-	6
ERROR_ACK	-	6
DATA_ACK_REQ	6	-
REPLY_REQ	6	-
REPLY_UPDATE_REQ	6	-
DATA_ACK_IND	6	-
REPLY_IND	6	-
DATA_ACK_STATUS_IND	6	-
REPLY_STATUS_IND	6	-
REPLY_UPDATE_STATUS_IND	6	-
ERROR_ACK	6	-

Table 11-10 presents the allowed sequence of DLPI primitives for the connection establishment phase of connection mode service.

Table 11-10. Connection Establishment Phase

STATES / EVENTS	IDLE 6	OUTCON PEND 8	INCON PEND 9	CONN_RES PEND 10	DATA-XFER 11	DISCON 8 PEND 15	DISCON 9 PEND 16
CONNECT_REQ	8	-	-	-	-	-	-
CONNECT_RES	-	-	10	-	-	-	-
DISCON_REQ	-	15	16	-	-	-	-
PASS_CONN	11	-	-	-	-	-	-
CONNECT_IND	9 [1]	-	9 [1]	-	-	-	-
CONNECT_CON	-	11	-	-	-	-	-
DISCON_IND1 (outcnt == 0)	-	6	-	-	6	-	-
DISCON_IND2 (outcnt == 1)	-	-	6 [2]	-	-	-	-

Table 11-10. Connection Establishment Phase (Cont.)

STATES / EVENTS	IDLE 6	OUTCON PEND 8	INCON PEND 9	CONN_RES PEND 10	DATA-XFER 11	DISCON 8 PEND 15	DISCON 9 PEND 16
DISCON_IND3 (outcnt > 1)	-	-	9 [2]	-	-	-	-
OK_ACK1 (outcnt == 0)	-	-	-	-	-	6	-
OK_ACK2 (outcnt == 1, token == 0)	-	-	-	11 [2]	-	-	6 [2]
OK_ACK3 (outcnt == 1, token != 0)	-	-	-	6 [2,3]	-	-	6 [2]
OK_ACK4 (outcnt > 1, token != 0)	-	-	-	9 [2,3]	-	-	9 [2]
ERROR_ACK	-	6	-	9	-	8	9

Table 11-11 presents the allowed sequence of DLPI primitives for the connection mode data transfer phase.

Table 11-11. Connection Mode Data Transfer Phase

STATES / EVENTS	IDLE 6	DATA-XFER 11	USER RESET PEND 12	PROV RESET PEND 13	RESET_RES PEND 14	DISCON 11 PEND 17	DISCON 12 PEND 18	DISCON 13 PEND 19
DISCON_REQ	-	17	18	19	-	-	-	-
DATA_REQ	-	11	-	-	-	-	-	-
RESET_REQ	-	12	-	-	-	-	-	-
RESET_RES	-	-	-	14	-	-	-	-
DISCON_IND1 (outcnt == 0)	-	6	6	6	-	-	-	-
DATA_IND	-	11	-	-	-	-	-	-
RESET_IND	-	13	-	-	-	-	-	-
RESET_CON	-	-	11	-	-	-	-	-
OK_ACK1 (outcnt == 0)	-	-	-	-	11	6	6	6
ERROR_ACK	-	-	11	-	13	11	12	13

Precedence of DLPI Primitives

This section presents the precedence of DLPI primitives relative to one another. Two queues are used to describe DLPI precedence rules. One queue contains DLS user-originated primitives and corresponds to the STREAMS write queue of the DLS provider. The other queue contains DLS provider-originated primitives and corresponds to the STREAMS read queue of the DLS user. The DLS provider is responsible for determining precedence on its write queue and the DLS user is responsible for determining precedence on its read queue as indicated in the precedence tables below.

For each precedence table, the rows (labeled PRIM X) correspond to primitives that are on the given queue and the columns (labeled PRIM Y) correspond to primitives that are about to be placed on that queue. Each pair of primitives (PRIM X, PRIM Y) may be manipulated resulting in:

- Change of order, where the order of a pair of primitives is reversed if, and only if, the second primitive in the pair (PRIM Y) is of a type defined to be able to advance ahead of the first primitive in the pair (PRIM X).
- Deletion, where a primitive (PRIM X) may be deleted if, and only if, the primitive that follows it (PRIM Y) is defined to be destructive with respect to that primitive. Destructive primitives may always be added to the queue. Some primitives may cause both primitives in the pair to be destroyed.

The precedence rules define the allowed manipulations of a pair of DLPI primitives. Whether these actions are performed is the choice of the DLS provider for user-originated primitives and the choice of the DLS user for provider-originated primitives.

Write Queue Precedence

Figure 11-39 presents the precedence rules for DLS user-originated primitives on the DLS provider's STREAMS write queue. It assumes that only non-local primitives (those that generate protocol data units to a peer DLS user) are queued by the DLS provider.

For connection establishment primitives, this table represents the possible pairs of DLPI primitives when connect indications/responses are single-threaded. For the multi-threading scenario, the following rules apply:

- A `DL_CONNECT_RES` primitive has no precedence over either a `DL_CONNECT_RES` or a `DL_DISCONNECT_REQ` primitive that is associated with another connection correlation number (`dl_correlation`), and should therefore be placed on the queue behind such primitives.
- Similarly, a `DL_DISCONNECT_REQ` primitive has no precedence over either a `DL_CONNECT_RES` or a `DL_DISCONNECT_REQ` primitive that is associated with another connection correlation number, and should therefore be placed on the queue behind such primitives. Notice, however, that a `DL_DISCONNECT_REQ` does have precedence over a `DL_CONNECT_RES` primitive that is associated with the same correlation number. This is indicated in Figure 11-39.

PRIM Y PRIMX (on queue)	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
PI DL_INFO_REQ															
P2 DL_ATTACH_REQ															
P3 DL_DETACH_REQ															
P4 DL_BIND_REQ															
P5 DL_UNBIND_REQ															
P6 DL_UNITDATA_REQ						1									
P7 DL_UDQOS_REQ															
P8 DL_CONNECT_REQ											4				
P9 DL_CONNECT_RES											3	1	1		
P10 DL_TOKEN_REQ															
P11 DL_DISCONNECT_REQ								1							
P12 DL_DATA_REQ											5	1	3	3	
P13 DL_RESET_REQ											3				
P14 DL_RESET_RES											3	1	1		
P15 DL_SUBS_BIND_REQ															

162340

KEY:

CODE	Interpretation
□	Empty box indicates a scenario which cannot take place.
1	Y has no precedence over X and should be placed on queue behind X.
2	Y has precedence over X and may advance ahead of X.
3	Y has precedence over X and X must be removed.
4	Y has precedence over X and both X and Y must be removed.
5	Y may have precedence over X (DLS provider's choice), and if so then X must be removed.

Y may have precedence over X (DLS provider's choice), and if so then X must be removed.

Figure 11-39. Write Queue Precedence

Read Queue Precedence

Figure 11-40 presents the precedence rules for DLS provider-originated primitives on the DLS user's STREAMS read queue.

For connection establishment primitives, this table represents the possible pairs of DLPI primitives when connect indications/responses are single-threaded. For the multi-threading scenario, the following rules apply:

- A `DL_CONNECT_IND` primitive has no precedence over either a `DL_CONNECT_IND` or a `DL_DISCONNECT_IND` primitive that is associated with another connection correlation number (`dl_correlation`), and should therefore be placed on the queue behind such primitives.
- Similarly, a `DL_DISCONNECT_IND` primitive has no precedence over either a `DL_CONNECT_IND` or a `DL_DISCONNECT_IND` primitive that is associated with another connection correlation number, and should therefore be placed on the queue behind such primitives.
- A `DL_DISCONNECT_IND` does have precedence over a `DL_CONNECT_IND` primitive that is associated with the same correlation number (this is indicated in Figure 11-40). If a `DL_DISCONNECT_IND` is about to be placed on the DLS user's read queue, the user should scan the read queue for a possible `DL_CONNECT_IND` primitive with a matching correlation number. If a match is found, both the `DL_DISCONNECT_IND` and matching `DL_CONNECT_IND` should be removed.

If the DLS user is a user-level process, it's read queue is the stream head read queue. Because a user process has no control over the placement of DLS primitives on the stream head read queue, a DLS user cannot straightforwardly initiate the actions specified in the following precedence table. Except for the connection establishment scenario, the DLS user can ignore the precedence rules defined in Figure 11-40. This is equivalent to saying the DLS user's read queue contains at most one primitive.

The only exception to this rule is the processing of connect indication/response primitives. A problem arises if a user issues a `DL_CONNECT_RES` primitive when a `DL_DISCONNECT_IND` is on the stream head read queue. The DLS provider will not be expecting the connect response because it has forwarded the disconnect indication to the DLS user and is in the `DL_IDLE` state. It will therefore generate an error on seeing the `DL_CONNECT_RES`. To avoid this error, the DLS user should not respond to a `DL_CONNECT_IND` primitive if the stream head read queue is not empty. The assumption here is a non-empty queue may be holding a disconnect indication that is associated with the connect indication that is being processed.

When connect indications/responses are single-threaded, a non-empty read queue can only contain a `DL_DISCONNECT_IND`, which must be associated with the outstanding `DL_CONNECT_IND`. This `DL_DISCONNECT_IND` primitive indicates to the DLS user that the `DL_CONNECT_IND` is to be removed. The DLS user should not issue a response to the `DL_CONNECT_IND` if a `DL_DISCONNECT_IND` is received.

The multi-threaded scenario is slightly more complex, because multiple `DL_CONNECT_IND` and `DL_DISCONNECT_IND` primitives may be interspersed on the stream head read queue. In this scenario, the DLS user should retrieve all indications on the queue before responding to a given connect indication. If a queued primitive is a `DL_CONNECT_IND`, it should be stored by the user process for eventual response. If a queued primitive is a `DL_DISCONNECT_IND`, it should be matched (using the correlation number) against any stored connect indications. The matched connect indication should then be removed, just as is done in the single-threaded scenario.

PRIM Y PRIMX (on queue)	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
PI DL_INFO_ACK			1	1	1	1		1	1	1	1			
P2 DL_BIND_ACK			1		1									
P3 DL_UNITDATA_IND	2		1	2								2	2	
P4 DL_UDERROR_IND	2		1	1								2	2	
P5 DL_CONNECT_IND	2						2	4						
P6 DL_CONNECT_CON	2						2	3	1	1				
P7 DL_TOKEN_ACK					1	1		1	1	1	1			
P8 DL_DISCONNECT_IND	2				1		2						2	
P9 DL_DATA_IND	2						2	5	1	3	3		2	
P10 DL_RESET_IND	2						2	3					2	
P11 DL_RESET_CON	2						2	3	1	1			2	
P12 DL_OK_ACK			1	1	1			1	1	1				
P13 DL_ERROR_ACK			1	1	1			1	1	1	1			
P14 DL_SUBS_BIND_ACK			1		1									

162350

KEY:

CODE	Interpretation
☐	Empty box indicates a scenario which cannot take place.
1	Y has no precedence over X and should be placed on queue behind X.
2	Y has precedence over X and may advance ahead of X.
3	Y has precedence over X and X must be removed.
4	Y has precedence over X and both X and Y must be removed.
5	Y may have precedence over X (DLS provider's choice), and if so then X must be removed.

Y may have precedence over X (DLS provider's choice), and if so then X must be removed.

Figure 11-40. Read Queue Precedence

Guidelines for Protocol Independent DLS Users

DLPI enables a DLS user to be implemented in a protocol-independent manner such that the DLS user can operate over many DLS providers without changing the DLS user software. DLS user implementors must adhere to the following guidelines, however, to achieve this independence.

- The protocol-specific service limits returned in the `DL_INFO_ACK` primitive `dl_max_sdu` must not be exceeded. The DLS user should access these limits and adhere to them while interacting with the DLS provider.
- Protocol-specific DLSAP address and PPA identifier formats should be hidden from DLS user software. Hard-coded addresses and identifiers must be avoided. The DLS user should retrieve the necessary information from some other entity (such as a management entity or a higher layer protocol entity) and insert it without inspection into the appropriate primitives.
- The DLS user should not be written to a specific style of DLS provider, for example, *style* 1 vs. *style* 2. The `DL_INFO_ACK` returns sufficient information to identify which style of provider has been accessed, and the DLS user should perform (or not perform) a `DL_ATTACH_REQ` accordingly.
- The names of devices should not be hard-coded into user-level programs that access a DLS provider.
- The DLS user should access the `dl_service_mode` field of the `DL_INFO_ACK` primitive to determine whether connection or connectionless services are available on a given stream.

Guidelines for Using DLPI Under PowerMAX OS

All DLPI device drivers currently in PowerMAX OS, such as **`dec`**, **`egl`**, **`ie`**, and **`cnd`** are style 1 connectionless-mode providers. As previously stated in this chapter, a connectionless-mode data transfer service cannot guarantee reliable delivery of data. As such, applications must be written in such a way that they can recover from dropped messages.

A style 1 provider assigns a PPA (physical point of attachment) based on the major/minor number device file combination that the application has opened. The information about the driver's style can be obtained by using the `DL_INFO_REQ DLPI` primitive.

In the current version of PowerMAX OS, the following DLPI primitives are not supported:

`DL_ATTACH_REQ`	`DL_DETACH_REQ`	`DL_UDQOS_REQ`
`DL_CONNECT_REQ`	`DL_CONNECT_RES`	`DL_TOKEN_REQ`
`DL_DISCONNECT_REQ`	`DL_RESET_REQ`	`DL_RESET_RES`
`DL_ENABMULTI_REQ`	`DL_DISABMULTI_REQ`	`DL_PROMISCON_REQ`
`DL_PROMISCOFF_REQ`	`DL_XID_REQ`	`DL_XID_RES`
`DL_TEST_RES`	`DL_PHYS_ADDR_REQ`	`DL_SET_PHYS_ADDR_REQ`
`DL_GET_STATISTICS_REQ`	`DL_DATA_ACK_REQ`	`DL_REPLY_REQ`
`DL_REPLY_UPDATE_REQ`		

Some of the functionality provided by the above primitives can be obtained through the use of specific **`ioctl`** commands. For example, to turn on promiscuous mode for packet capture, one could use the `DLIOCSPROMISC` primitive to toggle the promiscuous state to **`on`** or **`off`**. For details on **`ioctl`**s for the DLPI layer, see the **`DLioctl(3dlpi)`** man page.

Also note that some of the primitives listed above, such as `DL_ATTACH_REQ`, are actually only needed for style 2 connection-mode providers.

Under the current release of PowerMAX OS, the following **`ioctl`**s are supported in DLPI layer:

Table 11-12. ioctls supported in DLPI Layer

DLPI Primitive	Description
`DLIOCSMIB`	Set MIB
`DLIOCGMIB`	Get MIB
`DLIOCSENADDR`	Set Ethernet address
`DLIOCGENADDR`	Get Ethernet address
`DLIOCSLPCFLG`	Set local packet copy flag
`DLIOCGLPCFLG`	Get local packet copy flag
`DLIOCSPROMISC`	Toggle promiscuous state
`DLIOCGPROMISC`	Get promiscuous state
`DLIOCADDMULTI`	Add multicast address
`DLIOCDELMULTI`	Delete multicast address
`DLIOCDISABLE`	Disable controller
`DLIOCENABLE`	Enable controller

Table 11-12. ioctls supported in DLPI Layer (Cont.)

DLPI Primitive	Description
`DLIOCRESET`	Reset controller
`DLIOCCSMACDMODE`	Toggle CSMA-CD mode
`DLIOCGETMULTI`	Get multicast address list

For details on parameters for the **`ioctl`**s listed in Table 11-12 see the **`DLioctls(3dlpi)`** man page.

For style 1 connectionless-mode providers, a **`bind`** call (`DL_BIND_REQ`) is used to establish a data link application's identity, by associating that application with a data link service access point (`DLSAP`), which is the point through which the application will communicate with the data link provider.

The application can determine the `DLSAP` by looking at the bind acknowledge data, or alternatively, the `DLSAP` can be specified by the application during the **`bind`** call (`DL_BIND_REQ`). For the `DL_BIND_REQ` call, a application must provide information in the `dl_bind_req_t` structure, including the *dl_sap* field which must be filled in to properly identify the `DLSAP`.

Once the **`bind`** call has been successfully made, subsequent type fields of incoming frames are compared to the bound *dl_sap* value. If the values are equal, then the frame is placed on the STREAMS read queue of that application.

To provide a correct *dl_sap* field, the application must know the address format. For drivers in PowerMAX OS, the `DLSAP` is composed of two parts: a 6 byte physical (Ethernet address), followed by a 1 or 2 byte SAP identifier. The size and information for decomposing the `DLSAP` address can be obtained from the `dl_info_ack_t` structure that is returned on a `DL_INFO_ACK` call.

A special SAP value, `PROMISCUOUS_SAP`, can be useful for packet capture types of applications. This SAP value matches all SAP values. Therefore, using a `PROMISCUOUS_SAP` SAP value results in a copy of all packets being received to also be received on this SAP.

A privileged process may also **`bind`** to a SAP already bound by another process. In cases where a frame qualifies to be sent to more than one process, independent copies of the frame will be made and placed on the STREAMS read queue of each process.

Under style 1 connectionless-mode providers, frames are sent using `DL_UNITDATA_REQ` messages (via **`putmsg(2)`** calls) and frames are received as `DL_UNITDATA_IND` messages (via **`getmsg(2)`** calls).

When sending a message, the control `strbuf` structure of the **`putmsg(2)`** call should point to a control buffer that contains a filled in `dl_unitdata_req_t` structure, followed by the destination Ethernet/SAP address. The data `strbuf` structure of the **`putmsg(2)`** call should point to a buffer that contains the actual data to be sent to the remote DLSAP application.

When receiving messages via **`getmsg(2)`** calls, the control `strbuf` structure of the **`getmsg(2)`** call will return a `dl_unitdata_ind_t` structure. The *dl_primitive* field will contain a value of `DL_UNITDATA_IND`, and the *dl_src_addr_offset* field of this structure will contain the offset in the `strbuf` control buffer where the source (originating) Ethernet/SAP address is located. The data `strbuf` structure of the **`getmsg()`** call will point to the buffer that contains the actual received data.

Using STREAMS Networking Buffers in a DLPI Application

Concurrent Computer Systems provides the capability to setup the networking buffers for a given STREAM stack such that the buffers are shared between the kernel and the user's address space. This feature minimizes the copying of buffer data between user space and the kernel, and also reduces the overhead of repeated kernel buffer allocations for the STREAM stack.

Note that the STREAMS networking buffer feature currently makes use of shared user/kernel space buffers on STREAMS **`write(2)`** and **`putmsg(2)`** calls; the data is still copied between kernel and user space on **`read(2)`** and **`getmsg(2)`** calls even when the target user space buffer is a STREAMS Networking Buffer.

For more information on STREAMS networking buffers, see Chapter 14 of the *PowerMAX OS Real-Time Guide*.

While use of cached local memory (the `BUFF_FIXED_LOCAL_CACHED` buffer reuse type) is not recommended for general application use, the `BUFF_FIXED_LOCAL_CACHED` buffer reuse type happens to be a potentially good fit for DLPI applications. Unlike all of the other buffer reuse types (`BUFF_FIXED_GLOBAL`, `BUFF_FIXED_NOCACHE`, `BUFF_RELOAD_GLOBAL`), the `BUFF_FIXED_LOCAL_CACHED` type places some additional restrictions on the application. Failure to follow these restrictions can result in incorrect data being sent or received.

The following restrictions for using local memory STREAMS Networking Buffers in a DLPI application must be observed:

- The application must set its CPU bias for the process to one CPU board. The `NBUFF_ALLOC` **`ioctl(2)`** call will return an error if this requirement is not followed.
- Any LWP in the process that issues a DLPI STREAM **`write(2)`**, **`read(2)`**, **`getmsg(2)`** or **`putmsg(2)`** call while using a cached local memory STREAMS networking buffer as the target buffer must have a CPU bias that is equal to the same CPU bias of the process.

  Additionally, even when the target user-space buffer is not a local memory cached networking buffer for a **`read(2)`** or **`getmsg(2)`** call, this same CPU bias restriction also applies to any message received that originated from a loopback (using the source Ethernet address as the destination Ethernet address), multicast, or broadcast message from the same program where a local memory cached networking buffer was used to send that message.
- In addition to the above biasing requirement, the application must bias itself to a SPECIFIC CPU board for the following DLPI devices:

`ie` must set the CPU bias to the board where ISE Ethernet device is located.

`egl` must set the CPU bias to the board where Eagle Ethernet interrupts are handled; this will be the CPU board of the CPU that handles VME level 5 interrupts. Either **`mpadvise(2)`** or **`intstat(1M)`** may be used to determine the CPU that handles the VME level 5 interrupt.

NOTE

Setting the CPU bias to the correct specific CPU board is entirely up to the application; the `NBUFF_ALLOC` **`ioctl(2)`** call will not attempt to enforce the setting of the CPU bias to the CPU board that is correct for **`ie`** or **`egl`** DLPI STREAMS stacks.

A Sample DLPI /STREAMS Networking Buffer Program

The following sample program shown below makes use of cache local memory STREAMS Networking Buffers in a DLPI STREAMS stack.

The following comments can be made about this sample program:

- The sample program shown below has been simplified for the sake of clarity. For example, the routine that reads the messages, **`readdatamsg()`**, could handle dropped messages by using a timeout mechanism, and it might also resend messages that appear to be lost or garbled. Also, the application could be re-coded to handle user-specified destination Ethernet/SAP addresses and DLPI device filenames, instead of using hard-coded values.
- The device used in this program is the ISE Ethernet device, **`/dev/ie0`**. Since this device is located on the first CPU board, the application uses **`cpu_bias(2)`** to set its cpu bias to the first CPU in the system. Note that by using **`mpadvise(3C)`**, it would be possible to determine if more than one CPU is present on the first CPU board, by using the `MPA_CPU_LMEM` command. Note that other DPI devices, such as **`/dev/cnd00, /dev/egl`**, etc., could also have been used.
- This program assumes that a connectionless mode service, style 1 DLPI interface is being provided, since this is the current interface mode and type for Concurrent DLPI devices. The `DL_UNITDATA_REQ` and `DL_UNITDATA_IND` messages are used to send and receive the data messages to/from a remote DLPI Ethernet device. The remote/target Ethernet/SAP address is hardcoded into the *ethersap_destaddr[]* array and it is used for the bind (`DL_BIND_REQ`) request.
- For the `DL_UNITDATA_REQ` messages, the `dl_unitdata_req_t` structure is stored at the front of the control buffer, immediately followed by the destination Ethernet/SAP address. The actual data for the message is stored separately into the local memory buffer data portion of the **`putmsg(2)`** call. Similarly, for the received `DL_UNITDATA_IND` mes-

sages, the *dl_unitdata_ind_t* structure is returned at the beginning of the control buffer, and the *dl_src_addr_offset* field of the *dl_unitdata_ind_t* structure contains the offset from the front of the control buffer to the start of the Ethernet/SAP source address that is also returned within the control buffer. The actual data portion of the message is returned in the local memory data buffer.

- This program sends messages to a corresponding remote DLPI application that is assumed to be already opened, bound and ready to receive messages. This remote application would be very similar to the code in this sample program, except that the message loop just receives and sends back the received messages to the originator. The sample program shown below sends the messages and then reads them back from the remote application and then checks the contents of the buffer to see that the correct data was returned.
- Multiple (4) buffers are setup and used. Each buffer is used to write out a message before the same buffers are used again to read the data back from the remote DLPI application.
- Note that since a fixed buffer reuse type is being used, the `NBUFF_WAIT` **`ioctl(2)`** call is made before re-using the same buffer again.

```
/*
 * cc -D_KMEMUSER example.c
 */
#include <sys/types.h>
#include <stropts.h>
#include <sys/ksynch.h>
#include <sys/fcntl.h>
#include <sys/stat.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ether.h>
#include <sys/dlpi_common.h>
#include <sys/procset.h>
#include <sys/bind.h>
#include <errno.h>
#include <stdio.h>

/* DLPI device file descriptor.
 */
int fd_ether;

/* Buffer size for reading and writing.
 * Use a value that is less than the maximum DLPI
 * transmission size, dl_max_sdu.
 */
#define BUFFER_SIZE1400

/* Control message info buffer.
 */
#define MAXDLBUF 1024
char ctlbuf[MAXDLBUF];
```

```
/* Internal routines.
 */
int dlinforeq(void);
int dlinfoack(char *);
int dlbindreq(unsigned int);
int dlbindack(char *);
int putcntlmsg(char *, int);
int recvackmsg(int, const char *, char *);
void dlunitdatareq(caddr_t, caddr_t, int);
void open_and_bind(void);
void senddatamsg(int);
void readdatamsg(int);
void setupnbuff(void);
void nbuffwait(int);

int dl_addr_length;

/* Ethernet device name.  May be modified to
 * /dev/cnd00, /dev/egl, etc.
 */
char *device_name = "/dev/ie0";

/* Hard-coded destination ethernet address and sap.
 * The ethernet address is 0.0.c3.01.65.11
 * and the sap value is 0x600.
 */
unsigned char ethersap_destaddr[8] =
    { 0x0, 0x0, 0xc3, 0x2, 0x7e, 0x65, 0x6, 0x0};
unsigned int sap_value = 0x600;

/* STREAMS Networking Buffer info structure array.
 * One entry per buffer.
 */
struct netbuffers {
    struct str_netbuff_info nb_info;
    int nb_data_sval;
};

/* Number of STREAMS Networking Buffers to use.
 */
#define NUM_BUFFERS4
struct netbuffers netbuff_array[NUM_BUFFERS];

/* Holds the current write buffer data value.
 * Each word in the buffer holds an incremented value.
 */
int msg_data_value;
```

```
/* Send and receive this many messages in each buffer.
 */
#define NUM_MESSAGES200


main(argc, argv)
int argc;
char **argv;
{
        int i, j;

        /* Open the device and bind using our sap value.
         */
        open_and_bind();

        /* Setup Streams Networking Buffers for the I/O.
         */
        setupnbuff();

        /* Send and receive NUM_MESSAGES messages in each buffer.
        */
        for (j = 0; j < NUM_MESSAGES; j++) {

            /* Use all the buffers to first send out messages.
            * Then read the messages back and check their contents.
             */
            for (i = 0; i < NUM_BUFFERS; i++)
                    senddatamsg(i);

            for (i = 0; i < NUM_BUFFERS; i++)
                    readdatamsg(i);
        }
}


/*
 * Open the Data Link Provider Inteface device and bind to it.
 */
void
open_and_bind()
{
        unsigned int *intp;
        dl_info_ack_t *infop;


        /* Open the DLPI device.
        */
        printf("device used is %s\n", device_name);
        if ((fd_ether = open(device_name, O_RDWR)) < 0) {
                perror("open");
                exit(-1);
        }
        /* Send an info request and then acknowledge it.
         */
        if (dlinforeq() < 0) {
                perror("dlinforeq");
                exit(-1);
        }
        if (dlinfoack(ctlbuf) < 0) {
                perror("dlinfoack");
                exit(-1);
        }

        /* Check that the medium type is ethernet,
         * and check that the provider style is style 1.
         */
        infop = &((union DL_primitives *)ctlbuf)->info_ack;
        if (infop->dl_mac_type != DL_ETHER) {
                printf("NON ETHERNET medium type %d\n", infop->dl_mac_type);
                exit(-1);
        }
        if (infop->dl_provider_style != DL_STYLE1) {
                printf("Unexpected provider style.  %d\n",
                        infop->dl_provider_style);
                exit(1);
        }
        /* Make a bind request with a specific SAP value
         * and then acknowledge it.
         */
        if (dlbindreq(sap_value) < 0) {
                perror("dlbindreq");
                exit(-1);
        }
        if (dlbindack(ctlbuf) < 0) {
                perror("dlbindack");
                exit(-1);
        }

        /* Print out the local ethernet/sap address.
         */
        intp = (unsigned int *)ethersap_destaddr;
        printf("SOURCE ETHERNET/SAP ADDRESS: %x ", *intp);
        intp++;
        printf("%x\n", *intp);
}
```

```
/* Allocate the STREAMS Networking Buffers.
 * Use the local memory cached buffers.
 * Bind the process to the first CPU, since we're using /dev/ie0,
 * which is located on the first CPU board.
 */
void
setupnbuff()
{
        int i, error;
        struct str_netbuff_info *infop;
        struct netbuffers *netbufp;
        cpuset_t cpumask = 0x1; /* bind to first cpu */

        /* Bind the process to the 1st CPU.
         */
        error = cpu_bias(CPU_SETBIAS, P_PID, P_MYID, &cpumask);
        if (error) {
                printf("cpu_bias(2) returned %d\n", error);
                exit(-1);
        }

        /* Allocate the local memory STREAMS Networking Buffers.
         */
        for (i = 0, netbufp = &netbuff_array[0];
                i < NUM_BUFFERS; i++, netbufp++)
        {
                infop = &netbufp->nb_info;
                infop->length_requested = BUFFER_SIZE;
                infop->req_type = NBUFF_ALLOC;
                infop->buff_type = BUFF_FIXED_LOCAL_CACHED;

                if (ioctl(fd_ether, I_NBUFF, infop) < 0) {
                        printf("I_NBUFF ioctl failed.\n");
                        perror("ioctl");
                        exit(-1);
                }
                printf("buffer %d setup at 0x%x\n",i, (char *) infop->address);

                /* Change request type field for nbuffwait() calls.
                 */
                infop->req_type = NBUFF_WAIT;
        }
}


/* Acknowledge the prior info request.
 */
int
dlinfoack(char *bufp)
{
        return(recvackmsg(DL_INFO_ACK_SIZE, "info", bufp));
}
```

```
/* Acknowledge the bind request.
 */
int
dlbindack(char *bufp)
{
        return(recvackmsg(DL_BIND_ACK_SIZE, "bind", bufp));
}


/* Acknowledge a previous request and print the
 * information returned by the acknowledge request.
 */
int
recvackmsg(int size, const char *what, char *bufp)
{
        int i, flags;
        union DL_primitives *dlp;
        dl_info_ack_t *infop;
        dl_bind_ack_t *bindp;
        char *charp;
        struct strbuf ctl;
        char c;

        ctl.maxlen = MAXDLBUF;
        ctl.len = 0;
        ctl.buf = bufp;
        flags = 0;

        if (getmsg(fd_ether, &ctl,(struct strbuf*)NULL, &flags)< 0) {
                printf("recvackmsg: %s getmsg: %s \n");
                return (-1);
        }
        dlp = (union DL_primitives *) ctl.buf;

        switch (dlp->dl_primitive) {
            case DL_BIND_ACK:
                bindp = (dl_bind_ack_t *)ctl.buf;
                printf("BIND ACKNOWLEDGE\n");
                dl_addr_length = bindp->dl_addr_length;
                printf("ETHERNET/SAP Address: ");
                for (i = 0, charp = ctl.buf + bindp->dl_addr_offset;
                        i < bindp->dl_addr_length; i++, charp++)
                {
                        c = *charp;
                        printf("%x ", c);
                }
                printf("\n");
                break;

            case DL_INFO_ACK:
            case DL_OK_ACK:
                break;

            case DL_ERROR_ACK:
                /* Acknowledgement error.
                 */
                switch (dlp->error_ack.dl_errno) {
                    case DL_BADPPA:
                        printf("recvackmsg: %s bad ppa (device unit)\n",
                                what);
                        break;
                    case DL_SYSERR:
                        printf("recvackmsg: %s\n", what);
                        break;
                    case DL_UNSUPPORTED:
                        printf(
                        "recvackmsg: %s: Service not supplied by provider\n",
                                what);
                        break;
                    case DL_NOTSUPPORTED:
                        printf(
    "recvackmsg: %s: Primitive known but not supported by DLS provider\n",
                            what);
                        break;
                    default:
                        printf("recvackmsg (default): %s error 0x%x \n",
                                what, (unsigned int) dlp->error_ack.dl_errno);
                        break;
        }
        return (-1);

            default:
                printf("recvackmsg: %s unexpected primitive ack 0x%x\n",
                        what, (unsigned int)dlp->dl_primitive);
        return (-1);
    }

    if (ctl.len < size) {
            printf("recvackmsg: %s ack too small (%d < %d)\n",
                    what, ctl.len, size);
            return (-1);
    }
    return (ctl.len);
}


/* Issue a bind request with the specified SAP value.
 */
int
dlbindreq(unsigned int sap)
{
        dl_bind_req_t req;

        memset((char *)&req, 0, sizeof(req));
        req.dl_primitive = DL_BIND_REQ;
        req.dl_sap = sap;
        return(putcntlmsg((char *)&req, sizeof(req)));
}
```

```
/* Send a DL_INFO_REQ primitive
 */
int
dlinforeq()
{
        dl_info_req_t req;

        memset((char *)&req, 0, sizeof(req));
        req.dl_primitive = DL_INFO_REQ;
        return(putcntlmsg((char *)&req, sizeof(req)));
}


/* Send a putmsg() control message request.
 */
int
putcntlmsg(char *ptr, int len)
{
        struct strbuf ctl;

        ctl.maxlen = 0;
        ctl.len = len;
        ctl.buf = ptr;
        if (putmsg(fd_ether, &ctl, (struct strbuf *) NULL, 0) < 0) {
            perror("putcntlmsg ");
            return (-1);
        }
        return (0);
}


/* Send a unit data request message to the target ethernet.
 * The data in the buffer is different for each message.
 *
 * Parameter:
 *      index   netbuff_array[] index of buffer to use.
 */
void
senddatamsg(int index)
{
        int i;
        unsigned int *intp;
        dl_unitdata_req_t dlu;
        char *charp;
        vaddr_t bufaddr = netbuff_array[index].nb_info.address;


        /* Setup a DL unitdata request structure.
         */
        dlu.dl_primitive = DL_UNITDATA_REQ;
        dlu.dl_dest_addr_length = dl_addr_length;
        dlu.dl_dest_addr_offset = sizeof(dl_unitdata_req_t);
        dlu.dl_priority.dl_min = 100;
        dlu.dl_priority.dl_max = 0;

        /* Put the dl_unitdata_req_t structure at the
         * very front of the buffer.
         */
        bcopy(&dlu, ctlbuf, sizeof(dl_unitdata_req_t));
        charp = ctlbuf + sizeof(dl_unitdata_req_t);

        /* Put the destination ethernet address and sap just after
         * the dl_unitdata_req_t structure in the buffer.
         */
        bcopy(ethersap_destaddr, charp, dl_addr_length);

        /* Check the current data value being used.
         * Save off the starting value into the netbuff_array entry.
         */
        if (msg_data_value > 0x0fffffff)
                msg_data_value = 1;
        netbuff_array[index].nb_data_sval = msg_data_value;

        /* Write the data into the write buffer.
         */
        for (i = 0, intp = (unsigned int *)bufaddr;
                i < BUFFER_SIZE; i +=4 , msg_data_value++, intp++)
        {
                *intp = msg_data_value;
        }

        /* Send the mesage.
         */
        dlunitdatareq(ctlbuf, (caddr_t)bufaddr, BUFFER_SIZE);
}


/* Write out data using the DL_UNITDATA_REQ primitive.
 * The front of the ctlbuffer should aleady have
 *  - the dl_unitdata_req_t structure followed by
 *  - the destination ethernet address and SAP
 */
void
dlunitdatareq(caddr_t ctlbuffer, caddr_t databuffer, int len)
{
        struct strbuf ctl;
        struct strbuf dctl;


        ctl.maxlen = 0;
        ctl.len = sizeof(dl_unitdata_req_t) + dl_addr_length;
        ctl.buf = ctlbuffer;

        dctl.maxlen = 0;
        dctl.len = len;
        dctl.buf = databuffer;
        if (putmsg(fd_ether, &ctl, &dctl, 0) < 0) {
                perror("dlunitdatareq");
                exit(-1);
        }
}


/* Attempt to read back the message that we just sent.
 * Validate the received message source address, length of data,
 * and the buffer contents.
 *
 * Parameter:
 *  index   netbuff_array[] index of buffer to use.
 */
void
readdatamsg(int index)
{
        int i, value, bad_data, flags;
        unsigned int *intp;
        char *charp;
        dl_unitdata_ind_t *unit_indp;
        struct strbuf strdata;
        struct strbuf strctl;
        struct netbuffers *nbp = &netbuff_array[index];

        /* Make sure that the message that was sent with this buffer
         * has already be sent. If not, wait until the buffer is free
         * for re-use.
         */
        nbuffwait(index);

        while (1) {
                /* Now attempt to read back the message.
                */
                strdata.maxlen = BUFFER_SIZE;
                strdata.len = 0;
                strdata.buf = (char *)nbp->nb_info.address;
                strctl.maxlen = MAXDLBUF;
                strctl.len = 0;
                strctl.buf = ctlbuf;
                flags = 0;

                if (getmsg(fd_ether, &strctl, &strdata, &flags) < 0) {
                        perror(" ead_data_message:getmsg");
                        exit(-1);
                }

                /* We're expecting to get only unit data indication messages.
                 */
                unit_indp = (dl_unitdata_ind_t *)strctl.buf;
                if (unit_indp->dl_primitive != DL_UNITDATA_IND) {
                        printf("NOT RECEIVED DL_UNITDATA_IND: %d received\n",
                                unit_indp->dl_primitive);
                        exit(-1);
                }

                /* The sender should match our target address.
                 * Ignore messages received from elsewhere.
                 */
                charp = strctl.buf;
                charp += unit_indp->dl_src_addr_offset;
                if (strncmp(charp, ethersap_destaddr,
                        unit_indp->dl_src_addr_length))
                {
                        continue;
                }

                /* The amount of data received should match the amount sent.
                 */
                if (BUFFER_SIZE != strdata.len)
                        continue;

                /* Check that the data pattern is good.
                 */
                for (i = 0, intp = (unsigned int *)strdata.buf, bad_data = 0,
                        value = nbp->nb_data_sval;
                        i < BUFFER_SIZE; i += 4, value++, intp++)
                {
                        if (*intp != value) {
                                if (!bad_data) {
                                        bad_data++;
                                        break;
                                }
                                printf(
                                    "Second data mismatch on same message.\n");
                                exit(-1);
                        }
                }

                /* If bad data, try to read the message again.
                 */
                if (bad_data)
                        continue;
                break;
        }
}


/* Called to wait until the buffer is free after a putmsg() operation.
 *
 * Parameter:
 *  indexnetbuff_array[] index of buffer to use.
 */
void
nbuffwait(int index)
{
        struct str_netbuff_info*infop = &netbuff_array[index].nb_info;
        do {
                if (ioctl(fd_ether, I_NBUFF, infop) < 0) {
                        printf("NBUFF_WAIT, ioctl failed.\n");
                        perror("ioctl");
                        exit(-1);
                }
        } while (infop->ref_count != 0) ;
}
```

Glossary

The following is a list of terms used throughout this manual:

alignment

The position in memory of a unit of data, such as a word or half-word, on an integral boundary. A data unit is properly aligned if its address is evenly divisible by the data unit's size in bytes. For example, a word is correctly aligned if its address is divisible by four. A half-word is aligned if its address is divisible by two.

ARP

Address Resolution Protocol

asm macro

The macro that defines system functions used to improve driver execution speed. They are assembler language code sections (instead of C code).

asynchronous

An event occurring in an unpredictable fashion. A signal is an example of an asynchronous event. A signal can occur when something in the system fails, but it is not known when the failure will occur.

automatic calling unit (ACU)

A device that permits processors to dial calls automatically over the communications network.

base level

The code that synchronously interacts with a user program. The driver's initialization and switch table entry point routines constitute the base level. Compare **`interrupt level`**.

block and character interface

A collection of driver routines, kernel functions, and data structures that provide a standard interface for writing block and character drivers.

block data transfer

The method of transferring data in units (blocks) between a block device such as a magnetic tape drive or disk drive and a user program.

block device switch table

The table constructed during automatic configuration that contains the address of each block driver entry point routine (for example, **`open(D2)`**, **`close(D2)`**, **`strat-egy(D2)`**). This table is called `bdevsw` and its structure is defined in **`conf.h`**.

block device

A device, such as a magnetic tape drive or disk drive, that conveys data in blocks through the buffer management code. Compare **`character device`**.

block driver

A device driver, such as for a magnetic tape device or disk drive, that conveys data in blocks through the buffer management code (for example, the `buf` structure). One driver is written for each major number employed by block devices.

block I/O

A data transfer method used by drivers for block access devices. Block I/O uses the system buffer cache as an intermediate data storage area between user memory and the device.

block

The basic unit of data for I/O access. A block is measured in bytes. The size of a block differs between computers, file system sizes, or devices.

boot device

The device that stores the self-configuration and system initialization code and necessary file systems to start the operating system.

bootable object file

A file that is created and used to build a new version of the operating system.

bootstrap

The process of bringing up the operating system by its own action. The first few instructions load the rest of the operating system into the computer.

boot

The process of starting the operating system. The boot process consists of self-configuration and system initialization.

buffer

A staging area for input-output (I/O) processes where arbitrary-length transactions are collected into convenient units for system operations. A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.

cache

A section of computer memory where the most recently used buffers, i-nodes, pages, and so on are stored for quick access.

called DLS user

The DLS user in connection mode that processes requests for connections from other DLS users.

calling DLS user

The DLS user in connection mode that initiates the establishment of a data link connection.

canonical processing

Terminal character processing in which the erase character, delete, and other commands are applied to the data received from a terminal before the data is sent to a receiving program. Other terms used in this context are canonical queue, which is a buffer used to retain information while it is being canonically processed, and canonical mode, which is the state where canonical processing takes place. Compare **`raw mode`**.

character device

A device, such as a terminal or printer, that conveys data character by character. Compare `block device`.

character driver

The driver that conveys data character by character between the device and the user program. Character drivers are usually written for use with terminals, printers, and network devices, although block devices, such as tapes and disks, also support character access.

character I/O

The process of reading and writing to/from a terminal.

CLNS

Connectionless Network Service, the datagram version of the OSI network layer

clone driver

A software driver used by STREAMS drivers to select an unused minor device number, so that the user process does not need to specify it.

communication endpoint

The local communication channel between a DLS user and DLS provider.

connection establishment

The phase in connection mode that enables two DLS users to create a data link connection between them.

connection management stream

A special stream that will receive all incoming connect indications destined for Data Link Service Access Point (DLSAP) addresses that are not bound to any other streams associated with a particular Physical Point of Attachment (PPA).

connection mode

A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a sequenced manner.

connection release

The phase in connection mode that terminates a previously established data link connection.

connectionless mode

A mode of transfer in which data is passed from one user to another in self-contained units with no logical relationship required among the units.

control and status register (CSR)

Memory locations providing communication between the device and the driver. The driver sends control information to the CSR, and the device reports its current status to it.

controller

The circuit board that connects a device, such as a terminal or disk drive, to a computer. A controller converts software commands from a driver into hardware commands that the device understands. For example, on a disk drive, the controller accepts a request to read a file and converts the request into hardware commands to have the reading apparatus move to the precise location and send the information until a delimiter is reached.

critical code

A section of code is critical if execution of arbitrary interrupt handlers could result in consistency problems. The kernel raises the processor execution level to prevent interrupts during a critical code section.

CSMA/CD

Carrier Sense Multiple Access/Collision Detection

cyclic redundancy check (CRC)

A way to check the transfer of information over a channel. When the message is received, the computer calculates the remainder and checks it against the transmitted remainder.

data structure

The memory storage area that holds data types, such as integers and strings, or an array of integers. The data structures associated with drivers are used as buffers for holding data being moved between user data space and the device, as flags for indicating error device status, as pointers to link buffers together, and so on.

data terminal ready (DTR)

The signal that a terminal device sends to a host computer to indicate that a terminal is ready to receive data.

data transfer

The phase in connection and connectionless modes that supports the transfer of data between two DLS users.

DDI/DKI

Device Driver Interface/Device Kernel Interface

demand paging

A memory management system that allows unused portions of a program to be stored temporarily on disk to make room for urgently needed information in main memory. With demand paging, the virtual size of a process can exceed the amount of physical memory available in a system.

device number

The value used by the operating system to name a device. The device number contains the major number and the minor number.

dev_t

The C programming language data type declaration that is used to store the driver major and the minor device numbers.

diagnostic

A software routine for testing, identifying, and isolating a hardware error. A message is generated to notify the tester of the results.

DLIDU

Data Link Interface Data Unit. A grouping of DLS user data that is passed between a DLS user and the DLS provider across the data link interface. In connection mode, a DLSDU may consist of multiple DLIDUs.

DLPI

Data Link Provider Interface

DLS provider

The data link layer protocol that provides the services of the Data Link Provider Interface.

DLS user

The user-level application or user-level or kernel-level protocol that accesses the services of the data link layer.

DLS

Data Link Service

DLSAP address

An identifier used to differentiate and locate specific DLS user access points to a DLS provider.

DLSAP

A point at which a DLS user attaches itself to a DLS provider to access data link services.

DLSDU

Data Link Service Data Unit. A grouping of DLS user data whose boundaries are preserved from one end of a data link connection to the other.

downstream

The direction of STREAMS messages flowing through a write queue from the user process to the driver.

driver entry points

Driver routines that provide an interface between the kernel and the device driver.

driver routines

See `routines`.

driver

The set of routines and data structures installed in the kernel that provide an interface between the kernel and a device.

DSAP

Destination Service Access Point

EDLIDU

Expedited Data Link Interface Data Unit

error correction code (ECC)

A generic term applied to coding schemes that allow for the correction of errors in one or more bits of a word of data.

expedited data transfer

A DLPI service that transfers data subject to separate flow control than that applying to normal data transfer. The service is intended to deliver the data ahead of any DLSDUs that may be in transit.

FDDI

Fiber Distributed Data Interface

function

A kernel utility used in a driver. The term function is used interchangeably with the term kernel function. The use of functions in a driver is analogous to the use of system calls and library routines in a user-level program.

initialization entry points

Driver initialization routines that are executed during system initialization (for example, **`init(D2)`**, **`start(D2)`**).

interface

The set of data structures and functions supported by the UNIX kernel to be used by device drivers.

interprocess communication (IPC)

A set of software-supported facilities that enable independent processes, running at the same time, to share information through messages, semaphores, or shared memory.

interrupt level

Driver interrupt routines that are started when an interrupt is received from a hardware device. The system accesses the interrupt vector table, determines the major number of the device, and passes control to the appropriate interrupt routine.

interrupt priority level (IPL)

The interrupt priority level at which the device requests that the CPU call an interrupt process. This priority can be overridden in the driver's interrupt routine for critical sections of code with the **`spl(D3)`** function.

interrupt vector

Interrupts from a device are sent to the device's interrupt vector, activating the interrupt entry point for the device.

IP

Internet Protocol

ISO

International Organization for Standardization

kernel buffer cache

A linked list of buffers used to minimize the number of times a block-type device must be accessed.

LLC

Logical Link Control, a sub-layer of the data link layer for media independent data link functions.

low water mark

The point at which more data is requested from a terminal because the amount of data being processed in the character lists has fallen creating room for more. It also applies to STREAMS queues regarding flow control.

MAC

Media Access Control, a sub-layer of the data link layer for media specific data link functions.

memory management

The memory management scheme of the UNIX operating system imposes certain restrictions on drivers that transfer data between devices.

message block

A STREAMS message is made up of one or more message blocks. A message block is referenced by a pointer to a `mblk_t` structure, which in turn points to the data block (`dblk_t`) structure and the data buffer.

message

All information flowing in a stream, including transferred data, control information, queue flushing, errors and signals. The information is referenced by a pointer to a `mblk_t` structure.

MIB

Management Information Base

modem

A contraction of modulator-demodulator. A modulator converts digital signals from the computer into tones that can be transmitted across phone lines. A demodulator converts the tones received from the phone lines into digital signals so that the computer can process the data.

module

A STREAMS module consists of two related `queue` structures, one each for upstream and downstream messages. One or more modules may be pushed onto a stream between the stream head and the driver, usually to implement and isolate a line discipline or a communication protocol. virtual to physical memory.

panic

The state where an unrecoverable error has occurred. Usually, when a panic occurs, a message is displayed on the console to indicate the cause of the problem.

PDU

Protocol Data Unit

PPA identifier

An identifier of a particular physical medium over which communication transpires.

PPA

The point at which a system attaches itself to a physical communications medium.

prefix

A character name that uniquely identifies a driver's routines to the kernel. The prefix name starts each routine in a driver. For example, a RAM disk might be given the `ramd` prefix. If it is a block driver, the routines are **`ramdopen`**, **`ramdclose`**, **`ramdstrategy,`** and **`ramdprint`**.

priority message

STREAMS messages that must move through the stream quickly are classified as priority messages. They are placed at the head of the queue for processing by the **`srv(D2)`** routine.

quality of service (QOS)

Characteristics of transmission quality between two DLS users.

queue

A data structure, the central node of a collection of structures and routines, which makes up half of a STREAMS module or driver. Each module or driver is made up of one queue each for upstream and downstream messages. Location: **`stream.h`**.

raw I/O

Movement of data directly between user address spaces and the device. Raw I/O is used primarily for administrative functions where the speed of a specific operation is more important than overall system performance.

raw mode

The method of transmitting data from a terminal to a user without processing. This mode is defined in the line discipline modules.

read queue

The half of a STREAMS module or driver that passes messages upstream.

routines

A set of instructions that perform a specific task for a program. Driver code consists of entry-point routines and subordinate routines. Subordinate routines are called by driver entry-point routines. The entry-point routines are accessed through system tables.

SAP

Service Access Point, conceptually the “point” at which a layer in the OSI model make its services available to the layer above it.

SCSI driver interface (SDI)

A collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface for writing Small Computer System Interface (SCSI) drivers.

SDU

Service Data Unit

semantic processing

Semantic processing entails input validation of the characters received from a character device.

small computer system interface (SCSI)

The American National Standards Institute (ANSI) approved interface for supporting specific peripheral devices.

SNMP

Simple Network Management Protocol

Source Code Control System (SCCS)

A utility for tracking, maintaining, and controlling access to source code files.

special device file

The file that identifies the device's access type (block or character), the external major and minor numbers of the device, the device name used by user-level programs, and security control (owner, group, and access permissions) for the device.

SSAP

Source Service Access Point

stream end

The stream end is the component of a stream farthest from the user process, providing the interface to the device. It contains pointers to driver (rather than module) routines.

stream head

Every stream has a stream head, which is inserted by the STREAMS subsystem. It is the component of a stream closest to the user process. The stream head processes STREAMS-related system calls and performs the transfer of data between user and kernel space.

STREAMS

A kernel subsystem used to build a stream, which is a modular, full-duplex data path between a device and a user process.

stream

A linked list of kernel data structures providing a full-duplex data path between a user process and a device or pseudo-device.

switch table entry points

Driver routines that are activated through `bdevsw` or `cdevsw` tables.

switch table

The operating system that has two switch tables, `cdevsw` and `bdevsw`. These tables hold the entry point routines for character and block drivers and are activated by I/O system calls.

synchronous data link interface (SDLI)

A UN-type circuit board that works subordinately to the input/output accelerator (IOA). The SDLI provides up to eight ports for full-duplex synchronous data communication.

system initialization

The routines from the driver code and the information from the master file that initialize the system (including device drivers).

TCP

Transmission Control Protocol, a connection oriented transport in the Internet suite

upstream

The direction of STREAMS messages flowing through a read queue from the driver to the user process.

user space

The part of the operating system where programs that do not have direct access to the kernel structures and services execute. The UNIX operating system is divided into two major areas: the user program and the kernel. Drivers execute in the kernel, and the user programs that interact with drivers generally execute in the user program area. This space is also referred to as user data area.

volume table of contents (VTOC)

Lists the beginning and ending points of the disk partitions by the system administrator for a given disk.

write queue

The half of a STREAMS module or driver that passes messages downstream.

Index

Spine for 1” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

Progr

STREAMS Modules and Drivers

0890426