Overview of STREAMS 2

What Is STREAMS?

STREAMS is a general, flexible facility and a set of tools for development of UNIX 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 UNIX 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. The STREAMS framework itself 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`(). Benefits of STREAMS are discussed in more detail later in this chapter.

A *Stream* is a full-duplex bidirectional processing and data-transfer path between a STREAMS driver in kernel space and a process in user space (see Figure 2-1). In the kernel, a Stream is constructed by linking 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 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 (see Figure 2-2) is a full-duplex 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* is 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 transfers data 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 through 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/deleting header and trailer information to data, and/or packetizing/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 communicates with its neighbors by passing messages. The module is not a required component in STREAMS, whereas the driver is, with the exception of a STREAMS-based pipe where only the Stream head is required.

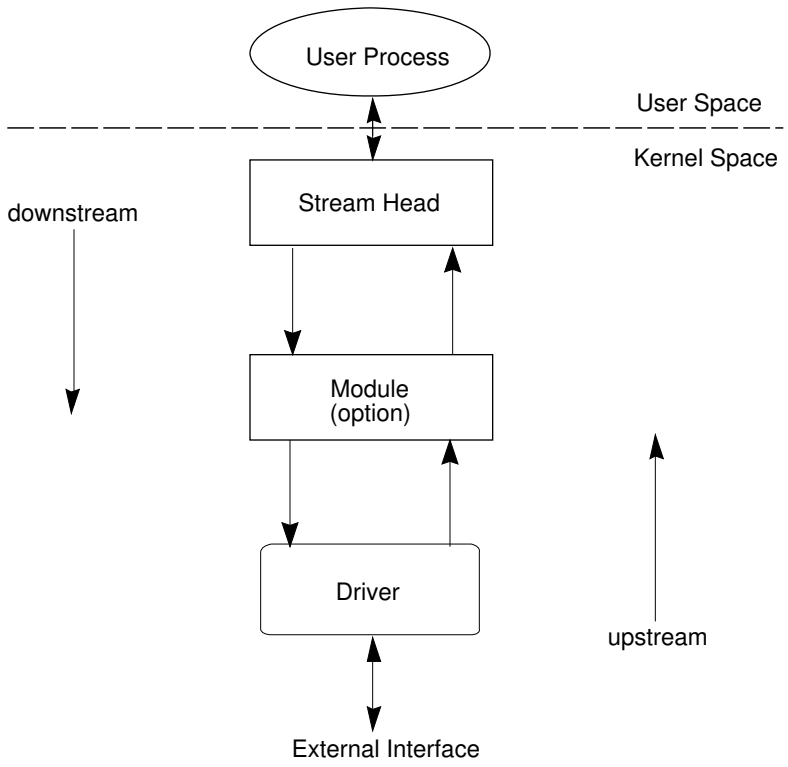

Figure 2-1 Simple Stream

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 connected in a Stream by a user process. No kernel programming, or assembly, is required to create the connection.

2

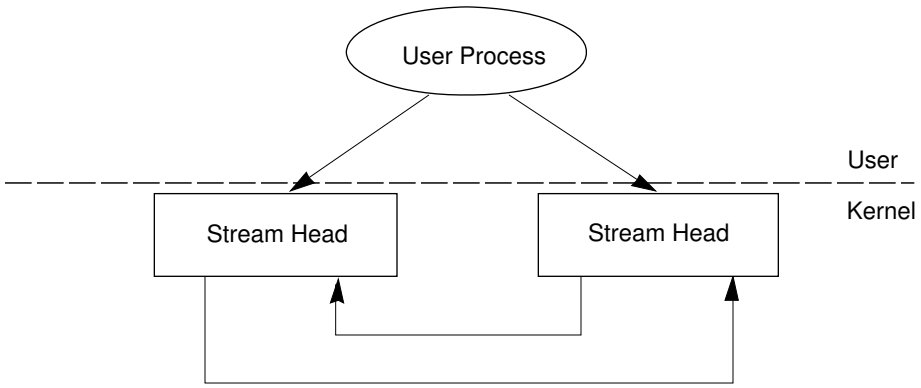

Figure 2-2 STREAMS-based Pipe

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 procedures for processing messages, and pointers for administering the Stream. Queues are always allocated in pairs, one queue for the read side and other for the write side. There is one queue pair for each driver and module and one for 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 passed from the Stream head toward the driver or from the process to the device, are said to travel *downstream* (also called *the 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 *the read side*).

A STREAMS message is made up of one or more message blocks. Each *block i*s a 3-tuple consisting 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 Streams 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. Each file system entry 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 will be created; the first `open` call will create the Stream, and subsequent `open` calls will return a file descriptor that references that Stream. Each process that opens the same minor device will share the same Stream to the device driver. This is not true with the clone devices.

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 code 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 is transmitted over the communications line, and data arriving on the line can be retrieved by reading from the device.

Code Example 2-1 Simple Stream

```
#include <sys/fcntl.h>
#include <stdio.h>
main()
{
    char buf[1024];
    int fd, count;

    if ((fd = open("/dev/ttya", O_RDWR)) < 0) {
            perror("open failed");
            exit(1);
    }
    while ((count = read(fd, buf, sizeof(buf))) > 0) {
            if (write(fd, buf, count) != count) {
                perror("write failed");
                break;
            }
    }
    exit(0);
}
```

In the example, `/dev/ttya` 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 2-3 shows the state of the Stream following the call to `open`().

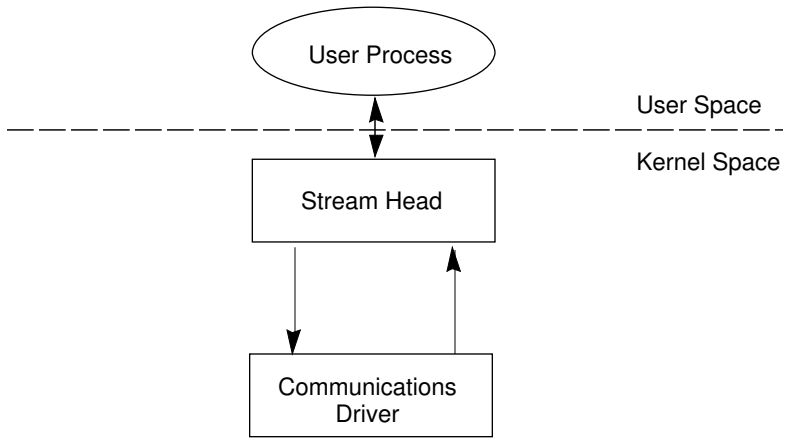

Figure 2-3 Stream to Communications Driver

This example illustrates a user reading data from the communications device and then writing the input back to the same device. In short, this program echoes all input back over the communications line. The example assumes that a user is sending 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.

Note – The application program must loop on a `read`() call until the desired number of bytes are read. The responsibility for the application getting all the bytes it needs is up to the application programmer, not the STREAMS facilities.

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

Flow control is a STREAMS mechanism that controls the rate of message transfer among the modules, drivers, Stream head, and processes. Flow control is local to each Stream and advisory (voluntary). It limits the number of characters that can be queued for processing at any queue in a Stream. This

mechanism limits buffers and related processing at any queue and in any one Stream, but does not consider buffer pool levels or buffer usage in other Streams. Flow control is not applied to high priority messages. Message priority is discussed in the section “Message Queueing Priority".

If the Stream exerts flow control, the `write`() call blocks until flow control has been relieved, unless `O_NDELAY` or corresponding POSIX `O_NONBLOCK` flag has been set. The call will not return until it has sent `count` bytes to the device. `exit`() is called to terminate the user process. This system call also closes all open files, dismantling the Stream, and flushes the data.

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 in later chapters.

Queues

A queue is an interface between a STREAMS driver or module and the rest of the Stream. Each instance of an open driver or pushed module has a pair of queues allocated, one for read-side and one for write-side. Queues are always allocated as an adjacent pair, similar to an array of structures (see Figure 2-4). 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. The `RD`(), `WR`(), and `OTHERQ`() routines move you from one to the other. See man pages RD(9F), WR(9F), OTHERQ(9F), queue(9S).

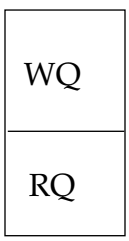

Figure 2-4 Queue Pair Allocation

A queue's `service` procedure is invoked to process messages on the queue. It usually removes successive messages from the queue, processes them, and calls the `put` procedure of the next module in the Stream to give the processed message to the next queue.

A queue's `put` procedure is invoked by the preceding queue's `put` and/or `service` procedure to add a message to the current queue. If a module does not need to queue messages, its `put` procedure can call the neighboring queue's `put` procedure. Chapter 4, "STREAMS Processing Routines" discusses the `service` and `put` procedures in more detail.

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 `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. The `close` routine of the driver is called when the last reference to the Stream is given up and the Stream is dismantled.

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`(9S) and `datab`(9S)) 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 will convert certain system calls to specified message types and send them downstream, and it will respond 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.

As shown in Figure 2-5, a STREAMS message consists of one or more linked message blocks that are attached to the first message block of the same message.

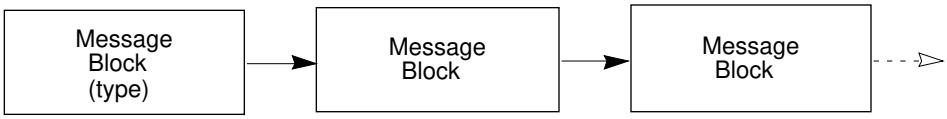

Figure 2-5 A Message

Messages can exist stand-alone, as in Figure 2-5, 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 2-6, Message 2 is linked to Message 1.

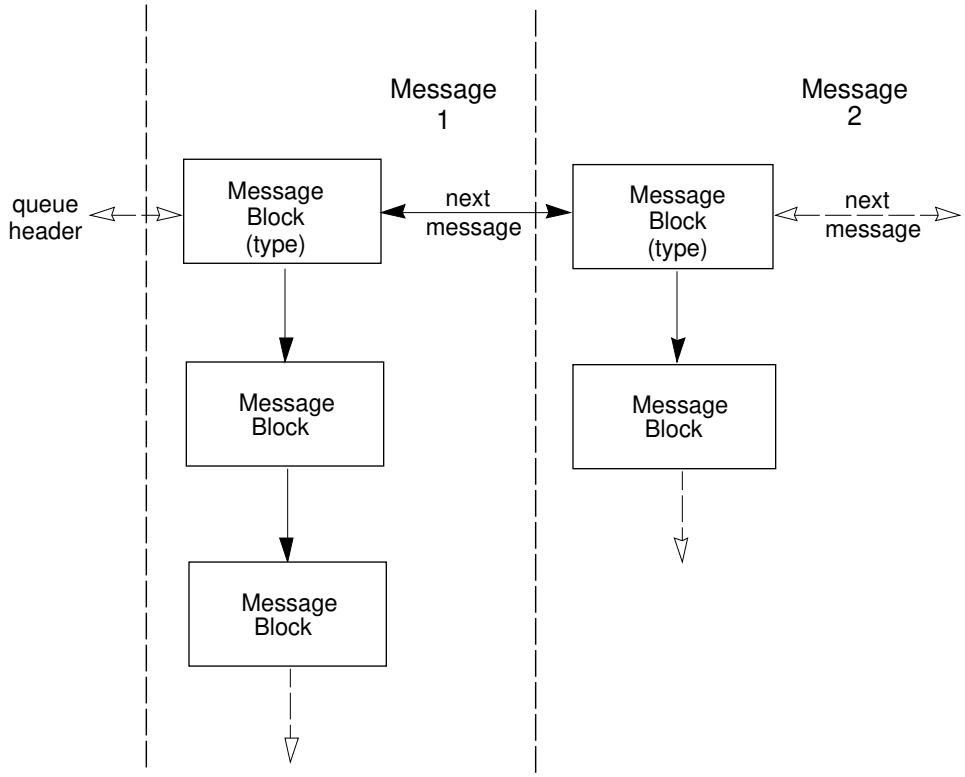

Figure 2-6 Messages on a Message Queue

When a message is in 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 lets developers manipulate messages and message queues.

Message Queueing 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. By convention, 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 is placed in the queue after high priority messages but before ordinary messages.

Message priority is defined by the message type. High-priority message types cannot be changed to be normal message types. 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 2-7). One queue performs functions on messages passing upstream through the module ("Au" and "Bu" in Figure 2-7). The other set ("Ad" and "Bd") performs another set of functions on downstream messages.

Each of the two queues in a module will generally have 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 can directly access 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.

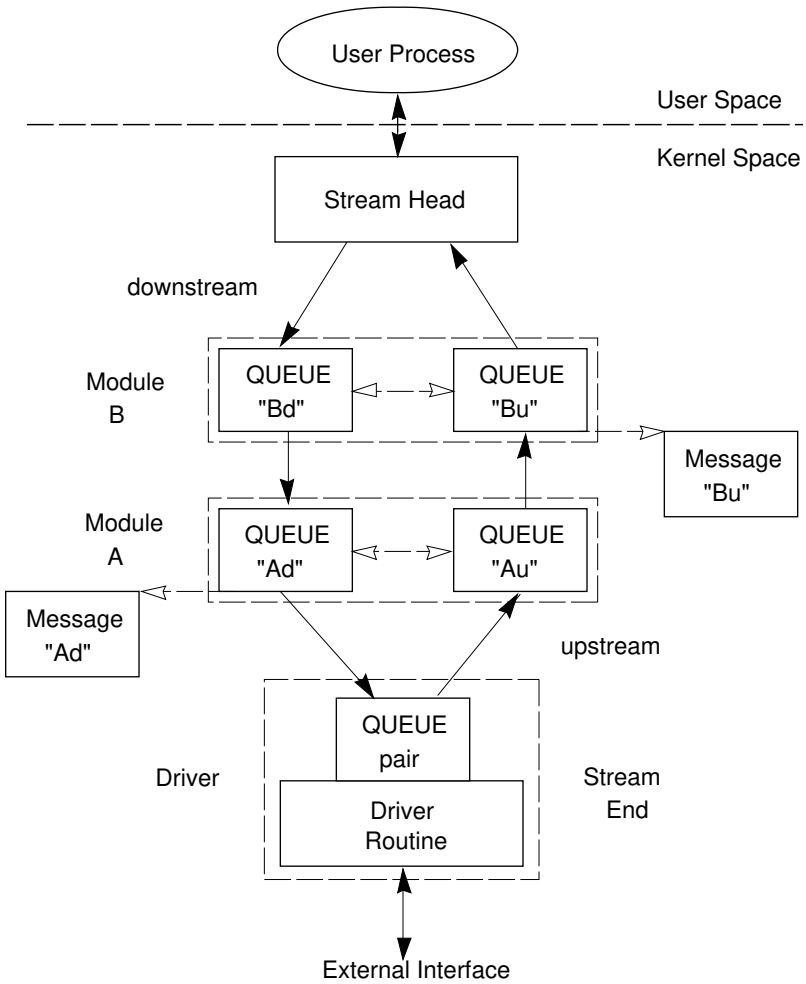

Figure 2-7 A Stream in More Detail

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

- Messages – These are dynamically attached to the queue on a linked list ("message queue", see "Ad" and "Bu" in Figure 2-7) 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 messages upstream and/or downstream, and they can also modify the private data in their module.
- Data –You may use a private field in the queue to reference private data structures (for example, state information and translation tables).

In general, each of the two queues in a module has a distinct set of all of 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.

There are three significant differences 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/deinitialized via `open` and `close`. A module may be initialized by either an `I_PUSH ioctl` (and thus deinitialized via the `(I_POP ioctl)` or an `open`. Modules are pushed automatically during an `open` if a stream has been configured by the `autopush`(1M) mechanism.

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

Multiplexing

Earlier, Streams were described as linear connections, or chains 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).

An example of a multiplexer is one that multiplexes data from several upper Streams over a single lower Stream, as shown in Figure 2-8. An *upper Stream* is one that is upstream from a multiplexer, and a *lower Stream* is one that is downstream from a multiplexer. A terminal windowing facility might be implemented in this fashion, where each upper Stream is associated with a separate window.

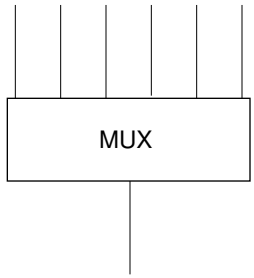

Figure 2-8 Many-to-one Multiplexer

A second type of multiplexer might route data from a single upper Stream to one of several lower Streams, as shown in Figure 2-9. An internetworking protocol could take this form, where each lower Stream links the protocol to a different physical network.

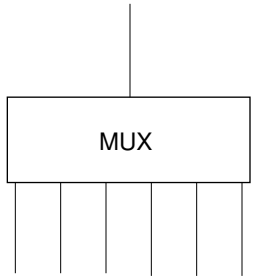

Figure 2-9 One-to-many Multiplexer

A third type of multiplexer might route data from one of many upper Streams to one of many lower Streams, as shown in Figure 2-10.

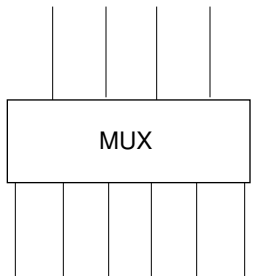

Figure 2-10 Many-to-many Multiplexer

The STREAMS mechanism supports the multiplexing of Streams through special pseudo-device drivers. Using a linking facility mechanism within the STREAMS framework, users can dynamically build, maintain, and dismantle multiplexed Stream configurations. Simple configurations like the ones shown in three previous figures 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 multiplexers: upper and lower multiplexers. *Lower multiplexers* have multiple lower Streams between device drivers and the multiplexer, and *upper multiplexers* have multiple upper Streams between user processes and the multiplexer.

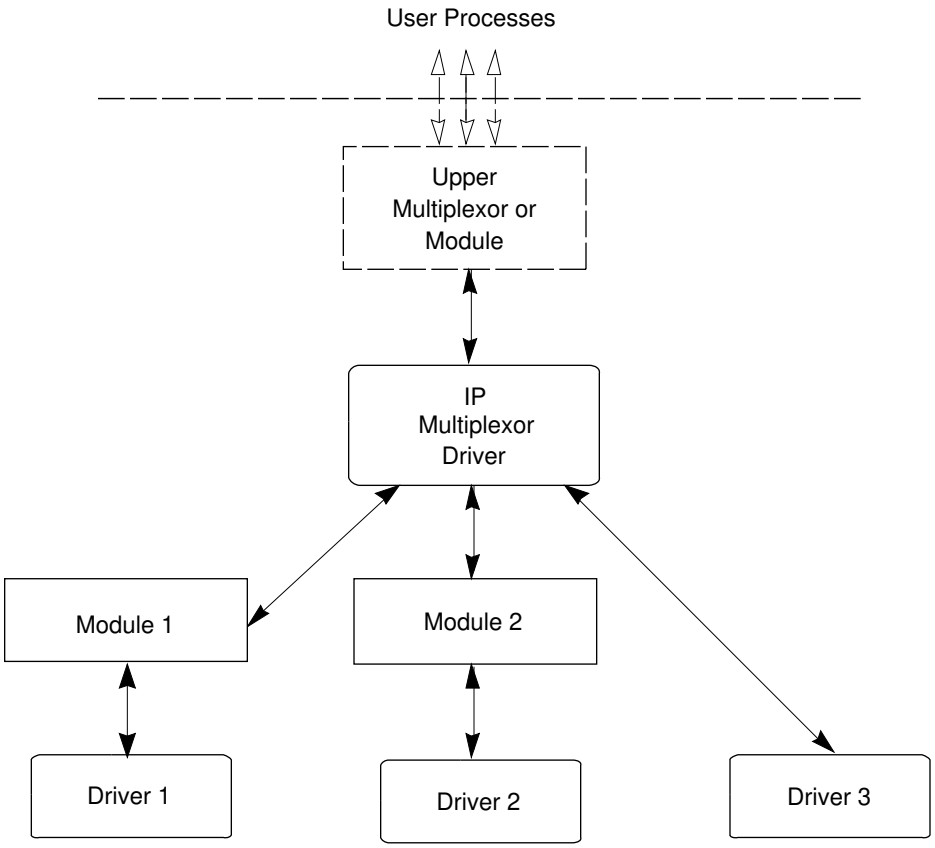

Figure 2-11 Internet Multiplexing Stream

Figure 2-11 is an example of the multiplexer configuration that would typically occur where internetworking functions were included in the system. This configuration contains three hardware device drivers. The IP (Internet Protocol) is a multiplexer.

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

Figure 2-11 shows the IP multiplexer as part of a larger configuration. The multiplexer configuration, as shown in the dashed rectangle, would generally have an upper multiplexer and additional modules. Multiplexers could also be cascaded below the IP multiplexer driver if the device drivers were replaced by multiplexer drivers.

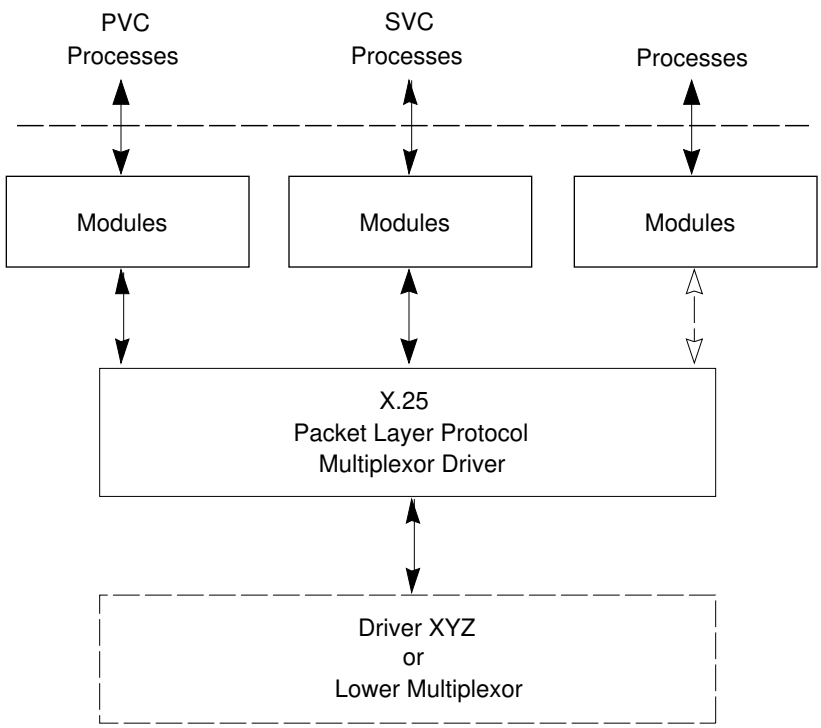

Figure 2-12 X.25 Multiplexing Stream

Figure 2-12 shows a multiplexer configuration where the multiplexer (or multiplexing driver) routes messages between the lower Stream and one of the upper Streams. This Stream performs X.25 multiplexing to multiple independent SVC (Switched Virtual Circuit) and PVC (Permanent Virtual Circuit) user processes. Upper multiplexers 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 multiplexer.

You can choose either upper or lower multiplexing, or both, when designing their applications. For example, a window multiplexer would have a similar configuration to the X.25 configuration of Figure 2-12, with a window driver replacing 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 multiplexer drivers would differ significantly. The IP multiplexer of Figure 2-11 has a different configuration than the X.25 multiplexer, and the driver would implement its own set of processing and routing requirements in each configuration.

In addition to upper and lower multiplexers, more complex configurations can be created by connecting Streams containing multiplexers to other multiplexer drivers. With such a diversity of needs for multiplexers, it is not possible to provide general purpose multiplexer drivers. Rather, STREAMS provides a general purpose multiplexing facility. The facility allows users to set up the inter-module/driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Benefits of STREAMS

STREAMS provides a flexible, portable, and reusable set of tools for development of UNIX system communication services. STREAMS allows an 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. This is discussed in Chapter 12, “STREAMS-Based Terminal Subsystem".

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 set of messages and the rules that allow passage of these messages across the boundary. A module that implements a service interface will receive a message from a neighbor and respond with an appropriate action (for example, send 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, as shown in Figure 2-13, presents a protocol service interface at its input and output sides. In this case, 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 abilities to manipulate modules from 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 2-13 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).

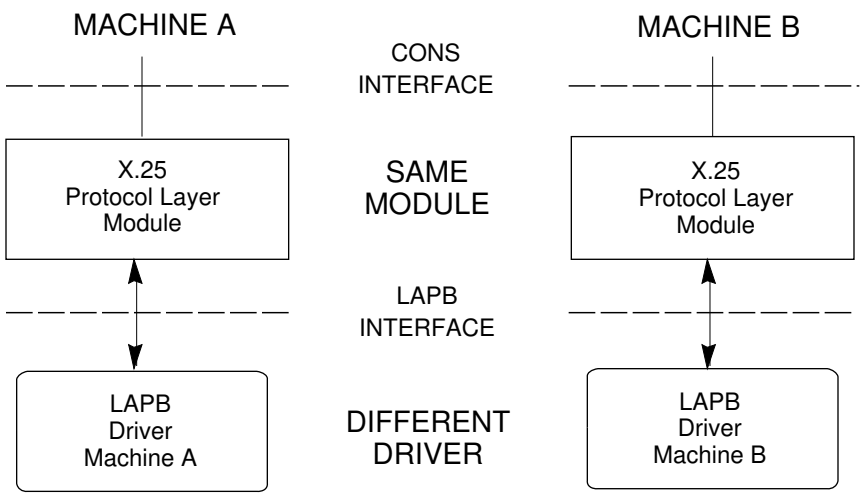

Figure 2-13 Protocol Module Portability

Protocol Substitution

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

Protocol Migration

Figure 2-14 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 will connect without modification to either an X.25 module or X.25 driver that has the same service interface.

By shifting functions between software and firmware, you 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.

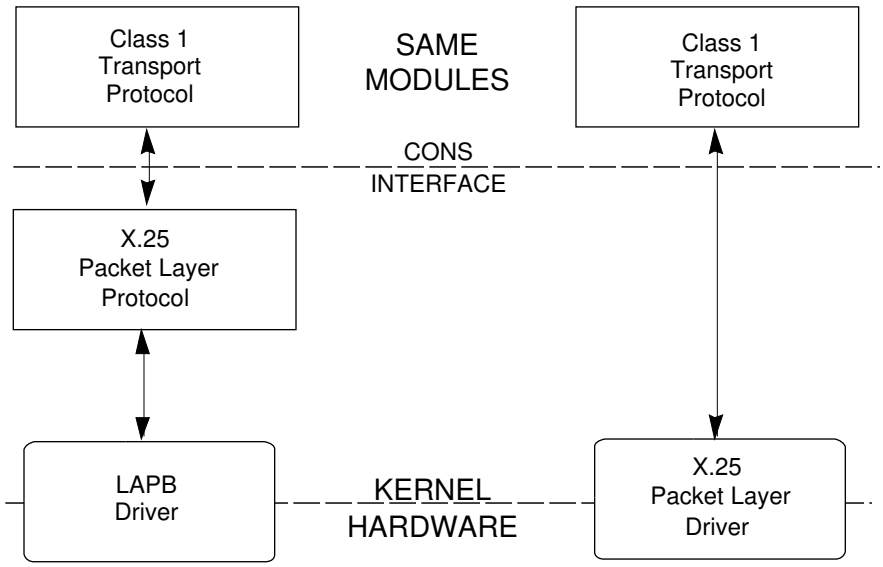

Figure 2-14 Protocol Migration

Module Reusability

Figure 2-15 shows the same canonical module (for example, one that provides delete and kill processing on character strings) reused in two different Streams. This module would typically be implemented as a filter, with no downstream service interface. In both cases, a tty interface is presented to the Stream's user process since the module is nearest the Stream head.

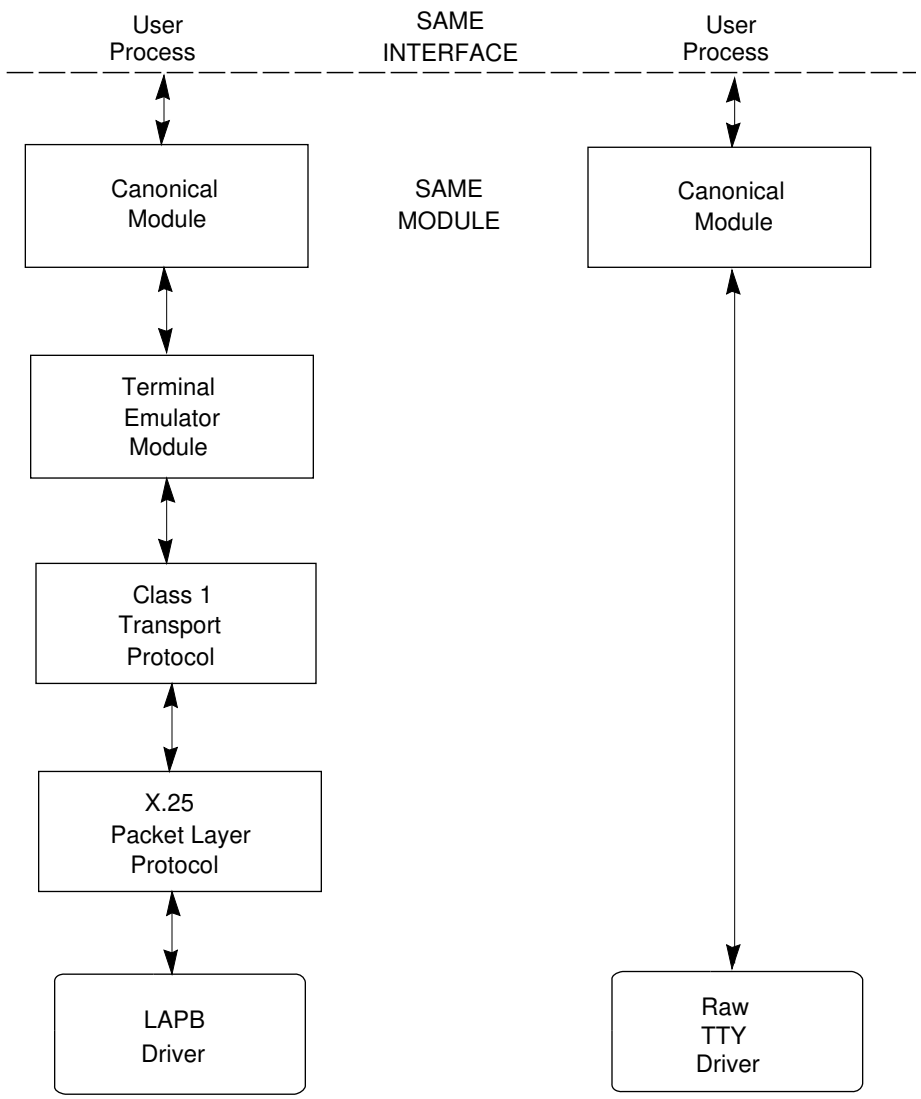

Figure 2-15 Module Reusability

2