68NW9209H52A

SYSTEM V/88 Release 3.2 STREAMS Primer

R5R- ..N 1N 8 1K aK<eKl 3.2

221)(- 2l -e _bK el

" #

T e info r ation is this document has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Fu r hermore, Motorola reserves the right to make changes to any products herein to improve reliability, function, or design. Motorola does not assume any liabili t arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of the others.

PREFACE

T e ***STREAMS*** (A9 < 0A H describes S T S, a major building block of the networking support services. The primer provides a high level, technical overview of S T S; it is intended for managers and developers who have prior knowledge of the SYS T M V/88 operating system and networking or other data communication facilities. For a more detailed description of S T E A S, see the ***STREAMS*** (A?6A, < < 0A B H # E7/0 H

Motorola and the Motorola symbol are registered trademarks of Motorola, Inc. SYST E V/88 is a trademark of Motorola, Inc.

UniSoft is a registered trademark of UniSoft Corporation.

U IX and Teletype are registered trademarks of AT&T.

Portions of this document are reprinted from copyrighted documents ^ÀÈ pe r ission of UniSoft Corporation. Copyright 1985, 1986, 1987, 1988, 1989, UniSoft Corporation. All rights reserved.

Po rions of this document have been previously copyrighted by AT&T and are C . BC ?) L (N.3)IQBQ D : 3 F F 3 ? = ! Q

SYSTEM V/88 Release 32 is based on the AT&T UNIX System V Release 3.2.

All rights reserved. No part of this manual may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by prior written permission of Motorola, Inc.

First Edition February 1990

Copyright 1990 by Motorola, Inc.

Contents

1 4C<8"D C,84L
General Description & &Y

2 F$<F,$HL
A Basic View of a Stream (&Y
Benefits of STREAMS ( )Y
An Advanced View of a Stream (2Y

3 D,/",4)LL C<$ 2L
Building a Stream) &Y

4 ?$<L$F$/LD4 A,94?L
STREAMS System Calls , &Y
An Asynchronous Protocol Stream Example , &Y
Other User Functions , &$Y

5 %=6%/%G%/E6!A-96@L
Introduction . &Y
Messages . &Y
Put and Service Procedures . ,Y
Kernel Processing . 0Y

6 A * % < L - 1 - A - % ? L
Introduction 1 &Y
Message Queue Priority 0"&Y
Flow Control 1 (Y
Multiplexing 1 .Y
Monitoring 1 4Y
Error and Trace Logging 0 5Y

7 < , F % < % ?-) 4 L9 2 ; < , ? 9 4 ? L
Introduction 2 &Y

G / 9 ? ? < K L
Glossary : !&Y

Figures

Figure 2-1: Basic Stream (&Y
Figure 2-2: Protocol Module Portability (,Y
Figure 2-3: Protocol Migration (0Y
Figure 2-4: Module Reusability (2Y
Figure 2-5: Stream In More Detail (4Y
Figure 3-1: Setting Up a Stream * &Y
Figure 4-1: Idle Stream Configuration for Example ,)Y
Figure 4-2: Asynchronous Terminal Streams + 3Y
Figure 5-1: A Message > (Y
Figure 5-2: Messages on a Message Queue .)Y
Figure 5-3: Operational Stream for Example . 0Y
Figure 5-4: Module Put and Service Procedures . 4Y
Figure 6-1: Streams Message Priority 0 (Y
Figure 6-2: Flow Control 0 ,Y
Figure 6-3: Internet Multiplexing Stream 0 0Y
Figure 6-4: X.25 Multiplexing Stream 1 2Y
Figure 6-5: Error and Trace Logging 1 &$Y

1 Introduction

4< /L$? <,;A,94L & &Y
How This Document is Organized & (Y
Other Documents &)Y
Notation Conventions ')Y

General Description

STREAMS is a general, flexible facility and a set of tools for development of operating system communication se r ices. With **STREAMS,** developers can provide services ranging from complete networking prot o ol suites to individual device drivers.

STREAMS defines standard interfaces b tween the ke r el and the rest of the operating system. The ass o iated mechanism is simple and open-ended. It consists of a set of system calls, ke r el resources, and ke r el utility routines. The standard interface and open-ended mechanism enable modular, portable development, and easy integration of h gher p rformance network se r ices and their components. S T E A S d o s not impose **any** G B / (3 network architecture. Instead, it provides a powerful framework with a consistent user interface that is compatible with the existing character I/O interface still available in SYS T M V/68.

STREAMS modularity and design reflect the '1ayers and options" characteristics of contempora r networking architectures. The basic components in a STREAMS implementation are referred to as modules. These modules, w hich reside in the ke r el, offer a set of pr o essing functions and associated se r ice interfaces. From user level, modules can b dynamically selected and interconnected to provide any rational pr o essing sequence. Ke r el programming, assembly, and link editing are not required to create the interconnection. Modules can also b dynamically "plugged into" existing connections from user level. S T EAMS modularity allows:

- User level programs that are independent of underlying protocols and physical communication media.
- Network architectures and higher level prot o ols that are independent of underlying prot o ols, drivers, and physical communication media.
- Higher level se r ices that can b created by selecting and connecting lower level se r ices and prot o ols.
- Enhanced portability of prot o ol modules resulting from S T E A S' well-defined structure and interface standards.

In addition to modularity, S T provides developers with integral functions, a library of utility routines, and facilities that expedite software design and implementation. The principal facilities are:

- Buffer management - To maintain S T ' own, independent buffer p o l.
- Flow control - To conse r e S T ' memory and processing resources.
- Scheduling - To incorporate S T ' own scheduling mechanism.
- Multiplexing - For processing interleaved data streams, such as those in SNA, X.25, and windows.
- Asynchronous operation of STREAMS and user processes - A lows S T 5-related operations to be performed efficiently from user level.
- Error and trace loggers - For debugging and administrative functions.

How This Document Is Organized

The primer is organized as follows:

- Chapter 2 provides an ove r iew of the applications and benefits of STREAMS and the S T mechanism.
- Chapter 3 describes how to set up a Stream from user level and how this initialization affects the ke r el. T is and following chapters are aimed at developers.
- Chapter 4 contains a detailed example and discusses it from user level.
- Chapter 5 describes ke r el operations associated with the Chapter 4 example, together with a discussion of basic STR E S ke r el facilities.
- Chapter 6 includes ke r el and user facilities not othe r ise described.
- Chapter 7 compares certain design features of character IO device drivers with STREAMS modules and drivers.
- The Glossary defines terms that are specific to STRE A S.

Other Documents

The *+)" ! *H(A>6A,;;0AB# F9/0H contains detailed STR E S information for programmers: how programmers can develop networking applications with S T user-level facilities and how system programmers can use S T ke r el-level facilities to build modules and drivers.

Section **2** of the SYSTEM **V/88** (A>6A,;;3B)10A2.0H & =F-:H includes descriptions (manual pages) of STRE A S-related system calls and other information.

Notation Conventions

#2.>.M.CQ.JQ.O4Q(7L)..OHSQ;B7 of Output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout SYS T M **V/88** documentation:

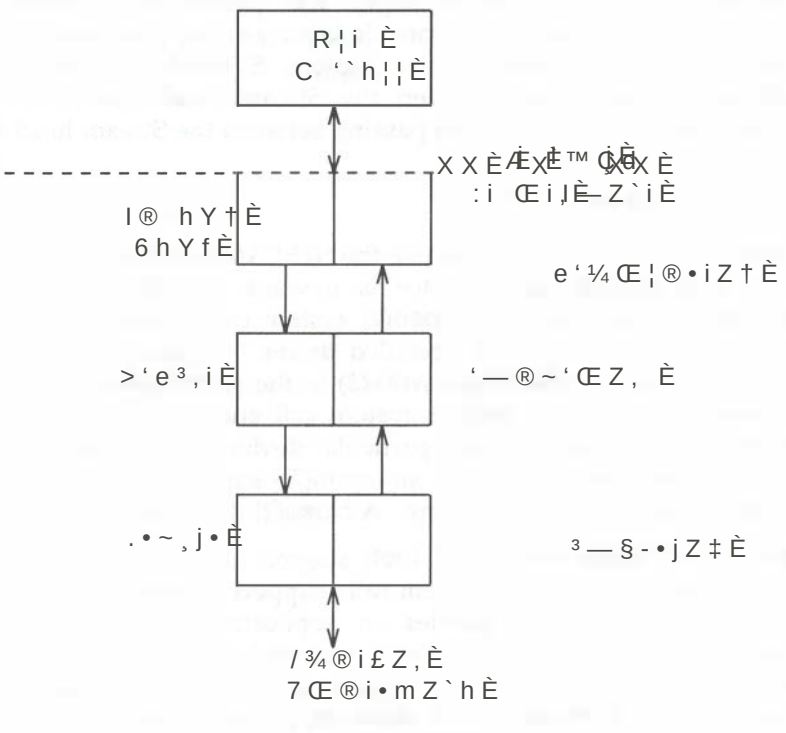
- Commands that you type on your terminal are shown in <]R@pe.
- Names of variables to which values must be assigned (e.g., @,CCG>A/H appear in 9D,:9.CH
- Text that is printed on your terminal by the computer is shown in **constant width** type. Code examples are also shown in **constant width** type to allow for more accurate representation of spacing. (Spacing is often a matter of coding style, but is sometimes critical.)

2 Overview

L > , L , $ H L 8 & LL A < $ 2 L (&Y
System Calls ((Y

$ 4 $ & - A & & L L ( )Y
Creating Service Interfaces 2-3
Manipulating Modules ( )Y
Protocol Portability (,Y
Protocol Substitution (.Y
Protocol Migration (.Y
Module Reusability (/Y

5 L " F 4 $ " L , $ H L 8 & LL A < $ 2 L (2Y
Stream Head (6Y
Modules (6Y
Stream End (&%Y



(NJ{lD„ ÈY¦wl®È iZ†È

Using a combination of system calls, ke r el routines, and ke r el utilities, S T EAMS passes data between a driver and the Stream head in the form of messages. Messages that are passed from the Stream head toward the driver are said to travel downstream, and messages passed in the other direction travel upstream.

T e Stream head transfers data between the data space of a user process and S T EAMS ke r el data space. Data sent to a driver from a user process are packaged into S T S messages and passed downstream. Messages a r ving at the Stream head from downstream are processed by the Stream head, and data are copied into user buffers. STREAMS can insert one or more modules into a Stream between the Stream head and driver to perform intermediate processing of data passing between the Stream head and driver.

d.oTgEa in aaoa

Applications programmers can use the STREAMS facilities via a set of system calls. This system call interface is upward compatible with the existing character I/O facilities. The]aC[system call recognizes a STRE A S file and create a Stream to the specified driver. A user process can send and receive data using hC;A and „•hNsGn the same manner as with character files and devices. The L]>sR system call enables application programs to ltwhgwmU h,yumtgyAUA f cuthtuUigUsU f shitaeushU hpvfapsU nSU addition, a set of generic STREAMS L]>sRm commands [see oshC;XN] support a variety of functions for accessing and controlling Streams. A >R]oCdism antles a Stream.

]aC[>R]oCC;A•h Nsad N]>sR port the basic set of operations on Streams. In addition, new system calls support advanced STRE A S facilities. The b]RR system call enables an application program to poll multiple Streams for various events. When used with the STRE A S I_SETSIG N`>sT command, a]Ra ows an application to process I/O in an asynchronous manner. The bytWoI and ICtXnI system calls enable application programs to interact with STREAMS modules and drivers through a se r ice interface (see *Creating Service Interfaces* section).

These calls are discussed in this document and in the *STREAMS Programmer's Guide.* They are specified in the *Programmer's Reference Manual* and the *System Administrator's Reference Manual.*

S T EAMS offers two major benefits for applications programmers: easy creation of modules that offer standard data communications services, and the ability to manipulate those modules on a Stream.

nngnf. SwahgnT.mgamtfg nlnmgoa

One b nefit of S T EAMS is that it 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 neigh b rs. In I P È a service interface is a specified set of messages and the rules for allowable sequences 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.

S T EAMS provides features that make it easier to design various application pr o esses and modules to common service interfaces. If these modules are written to comply with industry-standard service interfaces, they are called protocol modules.

In general, any t o 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 8 „protocol layer, as shown in Figure p„resents a prot o ol 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 8 „service interface.

dnS .ena eT.twadeSnag oa

S T EAMS provides the capabilities to manipulate modules from user level, to interchange modules with common service interfaces, and to present a service interface to a Stream user process. As stated in Chapter 1, these capabilities yield benefits when implementing networking services and protocols:

- 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.

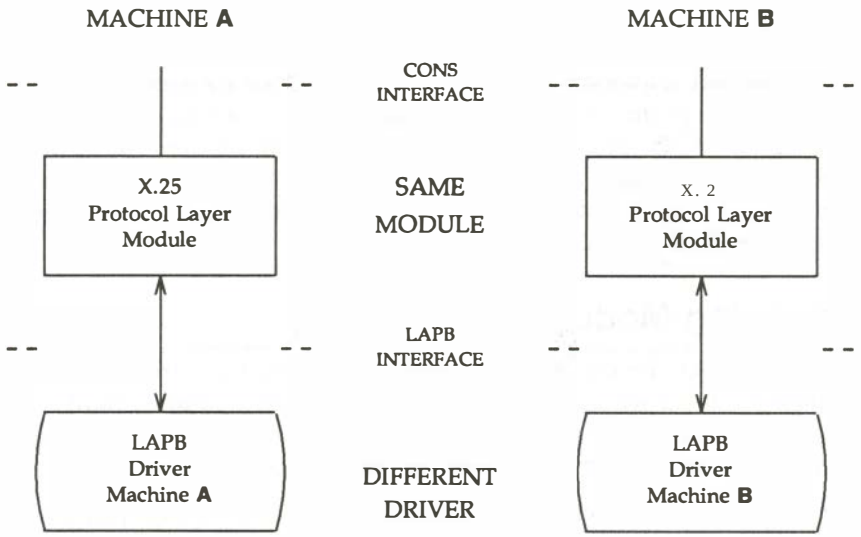
The following are examples of the benefits of STREAMS capabilities to developers for creating service interfaces and manipulating modules.

NOTE

All protocol modules used below were selected for illustrative purposes. Their use does not imply offers such modules as products.

!#

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



'NJy 2-2„ Protocol Module Portability

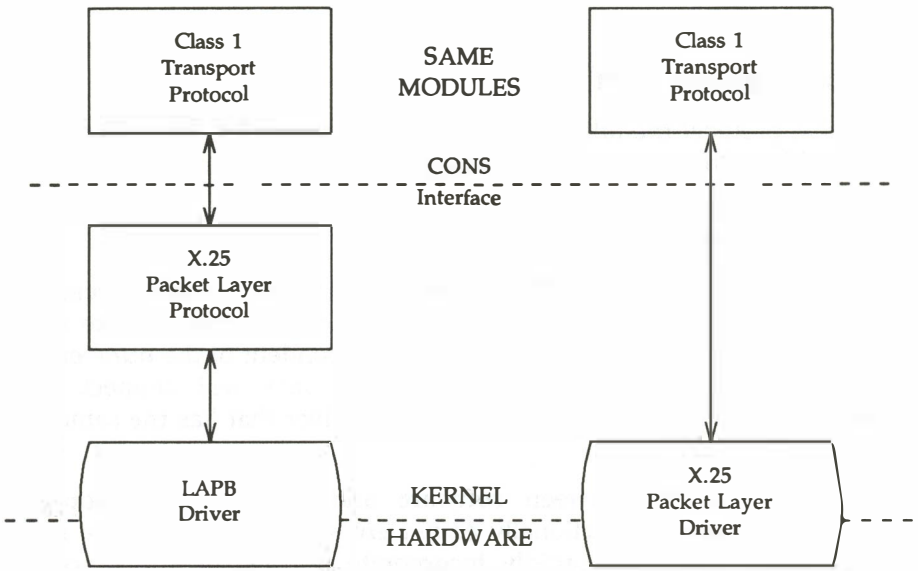
Protocol Substitution

A te r ative prot o ol modules (and device drivers) can be interchanged on the same machine if they are implemented to an equivalent service interface(s).

Protocol Migration

Figure 2-3 illustrates how S T can migrate functions be t een ke r el software and front end firmware. A common downstream se r ice interface allows the transport prot o ol module to be independent of the number or t pe of modules below. T e same transport module will connect without modification to either an X.25 module or X.25 driver that has the same se r ice interface.

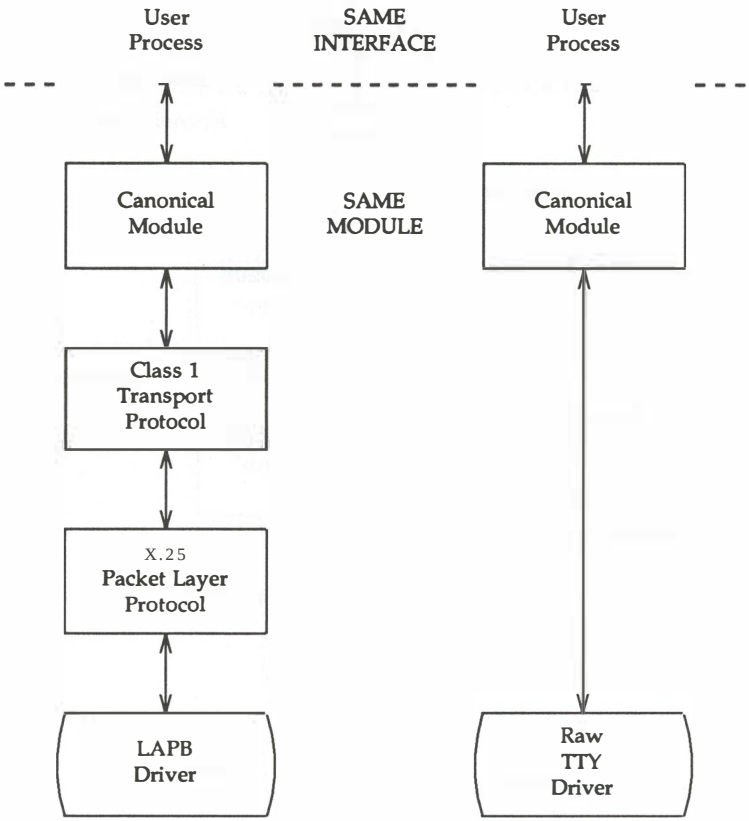
By shifting functions between software and firmware, developers can produce cost effective, functionally equivalent systems over a wide range of configurations. T ey can rapidly incorporate technological advances. The same transport protocol module can be used on a lower capaci t machine, where economics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



'N J y 2-3., Protocol Migration

!

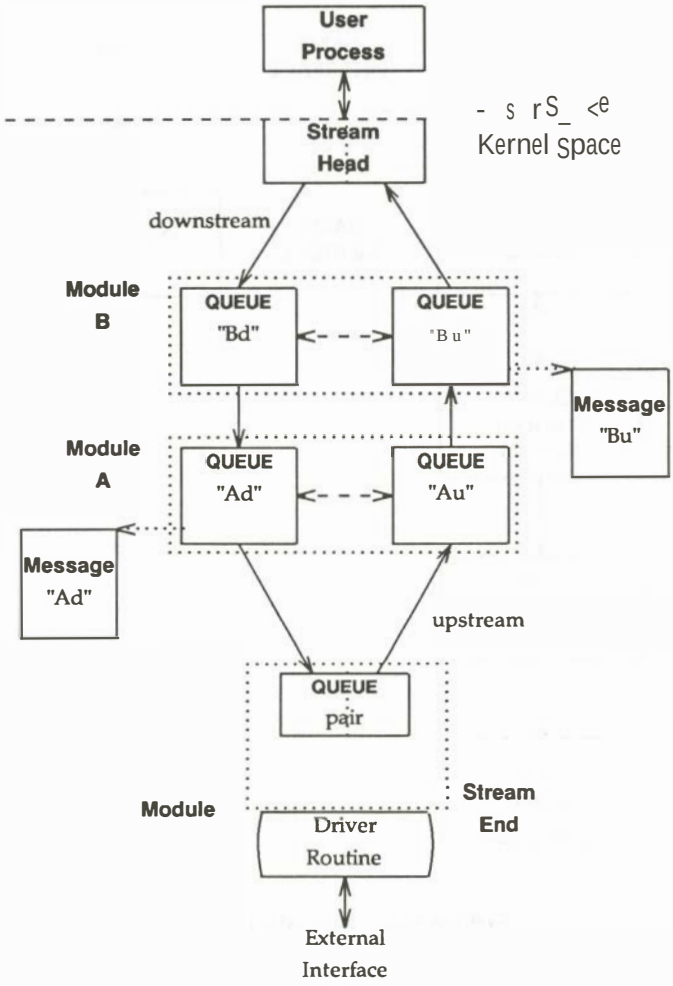
Figure 2-4 shows the same canonical module (e.g., one that provides delete and kill processing on character strings) reused in t o different Streams. This module would typically b implemented as a filter, with no downstream service interface. In both cases, a TTY interface is presented to the Stream's user process since the module is nearest the Stream head.



'N I y h**2-4,** Module Reusability

aSa aSTn SmgSag.g o æla nahTngnia

The STREAMS mechanism constructs a Stream by serially connecting kernel resident STREAMS components, each constructed from a specific set of structures. As described earlier and shown in Figure **2-5,** the primary STRE A S components are the Stream head, optional module(s), and Stream end.



'N J y h C „ È¬ • i Y ‡ ÈÈ È• i È i ¬ Y x , È

Each QUEUE in a module may contain or point to messages, processing procedures, or data:

- Messages - These are dynamically attached to the QUEUE on a linked list ("message queue", see Au and Bd in Figure 2-) as they pass through the module.
- Pr o essing pr o edures - A put pr o edure, to process messages, must be incorporated in each QUE U . A optional service procedure, to share the message processing with the put procedure, can also be incorporated. According to their function, the procedures can send messages upstream an d or downstream; they can also modify the private data in their module.
- Data - Developers may provide private data if required by the QUEUE to pero r message processing (e.g., state information and translation tables).

In general, each of the two QUEUEs in a module has a distinct set of all these elements. Additional module elements are described later in this document. Although depicted as distinct from modules (see Figure a„Stream head and the Stream end also contain a pair of QUEUEs.

dfgvfma End

A Stream end is a module in which the module's processing procedures are the driver routines. The procedures in the Stream end are different from those in other modules because they are accessible from an external device and because the S T EAMS mechanism allows multiple Streams to be connected to the same driver.

The driver can b a device driver, providing an interface between ke r el space and an exte r al communications device, or an internal pseudo-device driver. A pseudo-device driver is not directly related to any external device; it performs functions inte r al to the ke r el. The multiplexing driver discussed in Chapter 6 is a pseudo-device driver.

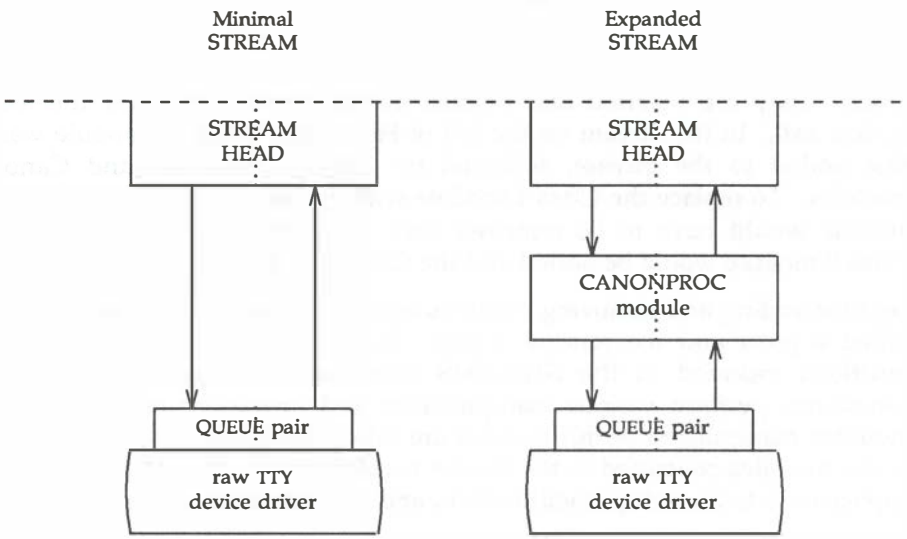
Device drivers must transform all data and status/control information be t een S T EAMS message formats and their exte r al representation. Differences between S T EAMS and character device drivers are discussed in Chapter „

3 Building A Stream

D , / " , 4 ' L L A = $ 2 L * &Y
Expanded Streams) (Y
Pushable Modules))Y

A Stream is created on the first]aC[system call to a character special file corresponding to a STS driver. A STS device is distinguished from other character devices by a field contained in the associated >AD}r• device table entry.

A Stream is usually built in two steps. Step one creates a minimal Stream consisting of just the Stream head and device driver; step two adds modules to produce an expanded Stream (see Figure 3-1).



'NIyh3-1, Setting Up a Stream

The first step has three parts: head and driver structures are allocated and initialized; the modules in the head and end are linked to each other to form a Stream; the driver open routine is called. If the driver performs all the character and device processing required, no modules need to be added to a Stream. Examples of STREAS drivers include a raw T driver (one that passes along input characters without change) and a driver with multiple Streams open to it (corresponding to multiple minor devices opened to a character device driver).

When the driver receives characters from the device, it places them into messages. The messages are then transfe r ed to the next Stream component, the Stream head, which extracts the contents of the message and copies them to user space. Similar pr o essing o curs for downstream character output; the Stream head copies data from user space into messages and sends them to the driver.

os enS ygya hTngni ra

As the second step in building a Stream, you can add modules to the Stream. In the right-hand Stream in Figure 3-1, the CANONPROC module was added to provide additional processing on the characters sent between head and driver.

Modules are added and removed from a Stream in last-in-first-out ;85A È order. They are inserted and deleted at the Stream head via the ioctl(2) system call. In the Stream on the left of Figure 2-4, the X.25 module was the first added to the Stream, followed by Class 1 Transport and Canonical modules. To replace the Class 1 module with a Class 0 module, the Canonical module would have to be removed first, then the Class 1 module, then a Class 0 module would be added and the Canonical module put back.

tcue,AcU e ll tynUey lU wcmgitynU mg l ,ucAUwcAcmrucAUAiedeUgwd.ett,yAeU EldUeccU is called a push and the remove a pop. Push and pop are two of the ioctl functions included in the STREAMS subset of N^>syStem calls. These commands perform various manipulations and operations on Streams. The modules manipulated in this manner are called pushable modules, in contrast to the modules contained in the Stream head and end. This stack terminology applies only to the setup, modification, and breakdown of a Stream.

NOTE

Subsequent use of the word module will refer to those pushable modules between Stream head and end.

The Stream head processes the N^>sad„ executes the push, which is analogous to opening the Stream driver. Modules are referenced by a unique symbolic name, contained in the STREAMS GX^Arm,odule table (similar to the >AD~table ass o iated with a device file). The module table and module name are inte r al to STREAMS and are accessible from user space only

through STREAMS **ioctl** system calls. T e **fmodsw** table points to the module template in the ke r el. When a module is pushed, the template is located, the module structures for both Q U U are allocated, and the template values are copied into the structures.

In addition to the module elements descr b d in *A Basic View of a Stream* section of Chapter **2,** each module contains pointers to an open routine and a close routine. T e open is called when the module is pushed; the close is called when the module is popped. Module open and close procedures are similar to a driver open and close.

As in other files, a S T S file is closed when the last process open to it closes the file by a **close(2)** system call. T s system call causes the Stream to
& / Q 3 G : % = I 8 /: *? Q L 9 B G B / % + K Q 2 0 E 5 M / @ Q //P / (L I / - " Q

0 m e P @ T I- w \ F m E v

Modules are pushed onto a Stream to provide special functions and/or additional prot o ol layers. In Figure **3-1,** the Stream on the left is opened in a minimal configuration with a raw tty driver and no other module added. The driver receives one character at a time from the device, places the character in a message, and sends the message upstream. The Stream head receives the message, extracts the single character, and copies it into the reading process buffer to send to the user pr o ess in response to a h C ; @syste m call. When the user process wants to send characters back to the driver, it issues a • h N s C syste m call, and the characters are sent to the Stream head. The head copies the characters into one or more multi-character messages and sends them downstream. A application program requiring no further kernel character processing would use this minimal Stream.

A user requiring a more terminal-like interface would need to insert a module to perform functions such as echoing, character-erase, and line-kill. Assuming that the CANONPROC module in Figure **3-1** fulfills this need, the application program first opens a raw tty Stream. T en, the + ' A @ C G A , È module is pushed above the driver to create a Stream of the form shown on the right of the figure. T e driver is not aware that a module has been placed above it, and continues to send single character messages upstream. The module receives single character messages from the driver, processes the characters, and accumulates them into line strings. Each line is placed into a message and sent to the Stream head. T e head now finds more than one character in the messages it receives from downstream.

Stream head implementation accommodates this change in format automatically and transfers the multiple-character data into user space. The Stream head also keeps track of messages partially transferred into user space (e.g., when the current user buffer can only hold part of the current message). Downstream operation is not affected: the head sends, and the driver receives, multiple character messages.

Note that the Stream head provides the interface between the Stream and user process. Modules and drivers do not have to implement user interface functions other than open and close.

4 User Level Functions

L K ? A % 3 L / / ? L , &Y

4 L ? K 6 * < 9 4 9 E ? L < 9 A 9 9 / L A = % 3 L
I 3 ; 0 % L , &Y

Initializing the Stream 4-3
Message Types , ,Y
Sending and Receiving Messages , .Y
Using Messages in the Example , .Y

B + % < 2 % < LE 7 ! B . : 6 ? L , 6Y

After a Stream has been opened, S T AMS-related system calls allow a user process to insert and delete (push and pop) modules. T at process can then communicate with and control the operation of the Stream head, modules, and drivers, and can send and receive messages containing data and control information. This chapter presents an example of some of the basic functions available to STREAMS-based applications via the system calls. Additional functions are described at the end of t h s chapter and in Chapter 6.

The full set of STREAMS-related system calls is:

^ c Œ „ Open a Stream (described in Chapter 3)

> S ^ r D „ Close a Stream (described in Chapter 3)

h D ; A „ Read data from a Stream

• h N s D „ Write data to a Stream

N ^ > s S „ Control a Stream

J D s X r J „ Receive the message at Stream head

d { s X r J „ Send a message downstream

d ^ S S „ Notify the application program when selected events occur on a Stream

The following two-part example describes a Stream that controls the data communication characteristics of a connection between an asynchronous terminal and a TTY port. It illustrates basic user level S T E A S features, then shows how messages can be used. Chapter 5 discusses the ke r el level Stream operations corresponding to the user level operations described in this chapter. See the *STREAMS Programmer's Guide* for more detailed examples of STREAMS applications, modules, and drivers.

s,a aoy ,dh ge,e RoaigeTe de aa hTggeEaoEe EUaga

In the example, our computer runs the operating system and 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 suppo r for these terminals is provided via a S T implemented asynchronous prot o ol. The protocol includes a varie t of options that are set when a terminal operator dials in to log on. The options are determined by a H C w x e$ T user process, *getstrm,* which analyzes data sent to it through a series of dialogs (prompts and responses) between the pr o ess and terminal operator.

The pr o ess 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 are:

- 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:

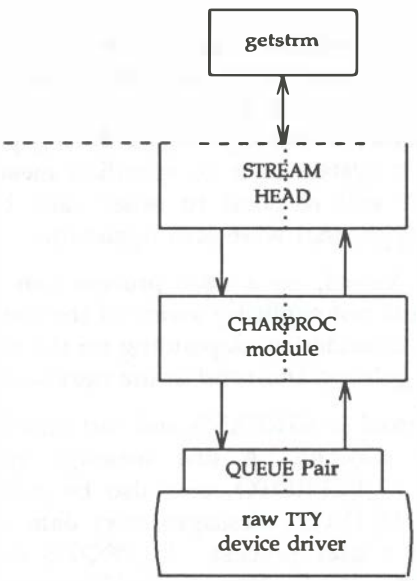
CHARPROC
: provides input character pr o essing functions, including dynamically settable (via control messages passed to the module) character echo and parity checking. The modules default settings echo characters but do not check character parity.

CANONPROC
: performs canonical processing on ASCII characters upstream and downstream (note that this performs some processing in a different manner from the standard operating system character I/O TTY subsystem).

ASCEBC
: translates EBCDIC code to ASCII upstream and ASCII to /),.9,È downstream.

At system initialization, a user process, *getstrm,* is created for each T port. *getstrm* opens a Stream to its port and pushes the CHARPROC module onto the Stream by use of an **ioctl** ! PUSH command. Then, the process issues Y È I C s W n 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 T driver connected to a limited-function communication port. The driver Y Œ e port transparently transmit and receive one unbuffered character at a time.



& L l y **4-1.** Idle Stream Configuration for E ample

Upon receipt of initial input from a 1 port, *getstrm* establishes a connection with the terminal, analyzes the option requests, verifies them, and issues S T E A S system calls to set the options. After setting up the options, *getstrm* creates a user application pr o ess. Later, when the user terminates that application, *getstrm* restores the Stream to its idle state by using system calls.

The next step is to analyze in more detail how the Stream sets up the commun cations options. Before doing so, the following section examines how messages are handled in S T S.

dgooewga lyUgoa

All S T S 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; a module can modify a message's type during processing. T e Stream head converts certain system calls to specified message types and sends them downstream; it will respond to other calls by copying the contents of certain message types that were sent upstream.

Messages exist only in the ke r el, so a user process can only send and receive buffers. T e process is not explicitly aware of the mtA AentUSltaU m,mU it may be aware of message boundaries, depending on the system call used (see the distinction between JDsXand hC;i@t,he next section).

Most message types are inte r al to S T E A S and can only be passed from one S T E A S module to another. A few message types, including M_DATA, M_PROTO, and M_PCPROTO, can also be passed between a Stream and user processes. M_DATA messages ca r y data within a Stream and between a Stream and a user pr o ess. M_PROTO or M_PCPROTO messages ca r both data and control info r ation. However, the distinction between control information and data is generally determined by the developer when implementing a particular Stream.

Control information includes service interface info r ation, carried between t o Stream entities that present service interfaces, and condition or status information, which may be sent between any two Stream entities regardless of their interface. A M_PCPROTO message has the same general use as an M_PROTO, but the former moves faster through a Stream (see *Message Queue Priority* in Chapter 6).

aysWnsHa„ST E S-related system call that sends messages; it is similar to •iLsCbysWnpHb„ provides a data buffer that is converted into an *:$ 5 „ message, and can also provide a separate control buffer to be placed into an *:/0-6-„ or *:/#/0-6-„ block. •iLsCb„ provides byte-stream data to be converted into *:$!6!„messages.

IDsWnsHa„S T related system call that accepts messages; it is similar to iC;@O„ne difference be t een the two calls is that iD;Ac„ accepts only data (messages sent upstream to the Stream head as message type *:$ 5 „ e.g., the characters entered from the terminal. HCsWcah„ simultaneously accept **both data and control information (message sent upstream as types** *:/0-6-„ or *:/#/0.5. „ IDsXoh„ also differs from iD;Ab„ because it preserves message boundaries so that the same boundaries exist above and below the Stream head (i.e., between a user process and a Stream). hD;A, generally ignores message boundaries, processing data as a byte stream.

Certain S T A S M]=soRm„ands, e.g., I_ST, also cause messages to be sent or received on the Stream. I_STR provides the general L^>saSa„bility of the character I/O subsystem. !„user process above the Stream head can issue aysWnI „ HDstWnI:S„T L]=scRm„and, and certain other S T S related system calls. Other S T S L]>spef„ perform functions that include changing the state of the Stream head, pushing and popping modules, or retu r ing special information. L^>soRm„ands are descr bed in more detail the *STREAMS Programmer's Guide.*

In addition to message types that explicitly transfer data to a process, some messages sent upstream result in information transfer. When these messages reach the Stream head, they are transformed into various forms and sent to the user process. The forms include signals, e r or codes, and call retu r values.

to.Swa dg ooefg ra.SaThgaose ieaga

Retu r ng to the asynchronous prot o ol example, the Stream was in its idle configuration (see Figure 4-1). *getstrm* had issued a IDsXoh„ and was sleeping until the arrival of a message from the Stream head. Such a message results from the driver detecting activity on the associated T port.

A incoming call arrives at port one and causes a ring detect signal in the modem. The driver receives the ring signal, answers the call, and sends upstream an M_PROTO message that contains information indicating an incoming call. *getstrm* is notified of all incoming calls, although it can choose to refuse the call because of system limits. In this idle state, *getstrm* also accepts M_PROTO messages indicating, for example, error conditions such as detection of line or modem problems on the idle line.

The M_PROTO message containing notification of the incoming call flows upstream from the driver into CHARPROC. CHARPROC inspects the message type, determines that message pr o essing is not required, and passes the unmodified message upstream to the Stream head. The Stream head copies the message into the H C t W b uHffers (one buffer for control information, the other for data) ass o iated with *getstrm* and wakes up the process. *getstrm* sends its acceptance of the incoming call with a c{ s Z r J „ system call which results in a downstream M_PROTO message to the driver.

Then, *getstrm* sends a prompt to the operator with a € h N a d „ issues a I C t W o t H „ receive the response. A i C ; @ o „ ld have been used to receive the response, but the H C s W o a H „ allows concurrent monitoring for control (M_PROTO and M_PCPROTO) information. *getstrm* now sleeps until the response characters, or information about possible error conditions detected by modules or driver, are sent upstream.

The first response, sent upstream in a M_DATA block, indicates that the code set is ASCII and canonical processing is requested. *getstrm* implements these options by pushing CANONPROC onto the Stream, above CHARPROC, to perform canonical processing on the input ASCII characters.

The response to the next prompt requests even pari t checking. *getstrm* sends an L ^ > I S T R command to CHARPROC, requesting the module to perform even parity checking on upstream characters. When the dialog indicate protocol option setting is complete, *getstrm* creates an application process. At the end of the connection, *getstrm* pops CANONPROC and then sends a I_STR to CHARPROC requesting the module to restore the no-parity idle state (CHARPROC remains on the Stream).

As a result of the above dialogs, the terminal at port one operates in the following configuration:

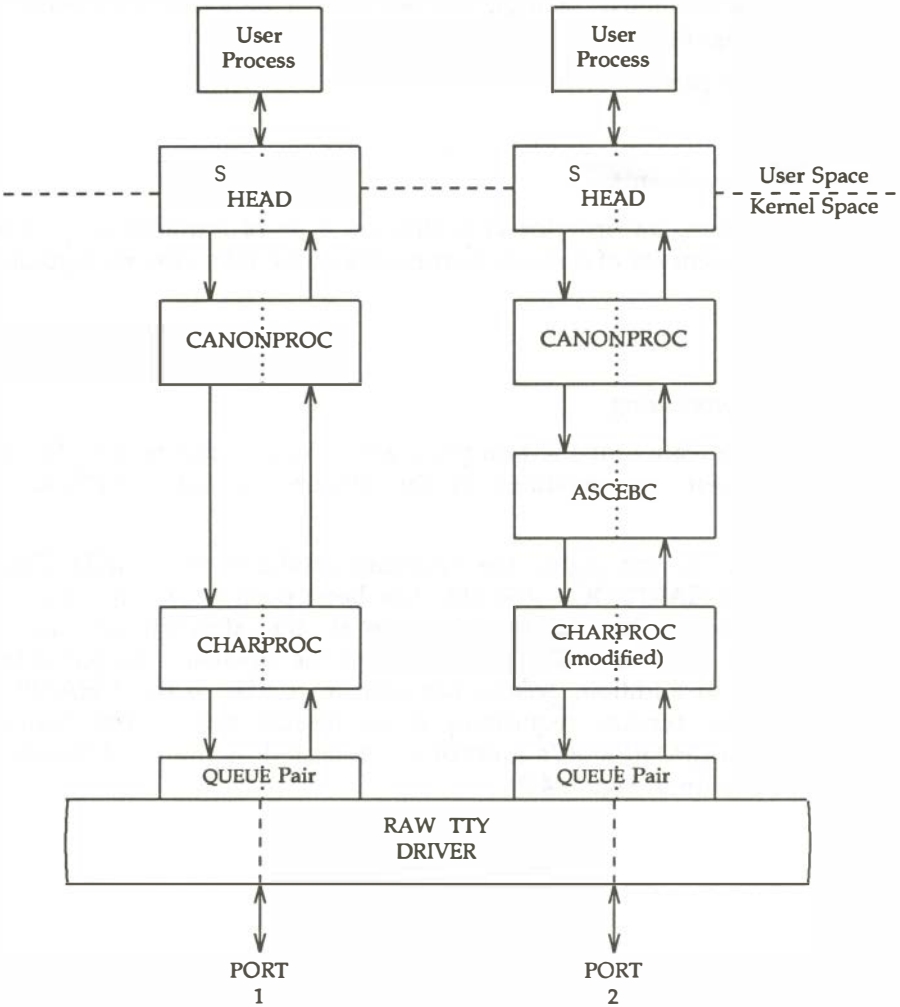
- ASCII, even par t
- Echo
- Canonical processing

In similar fashion, an operator at a different t of terminal on port t o requests a different set of options, that results in the following configuration:

- EBCDIC
- No Echo
- Canonical processing

The resultant Streams for the two ports are shown in Figure 4-2. For port one, on the left, the modules in the Stream are CANONPROC and CHARPROC.

For por two, on the rght, the resultant modules are CANONPROC, ASCEBC and CH A PROC. ASCEBC has been pushed on this Stream to translate between the ASCII interface at the downstream side of CANONPROC and the EBCDIC interface of the upstream output side of CHARPROC. In addition, *getstrm* has sent an I_STR to the CHARPROC module in this Stream requesting it to disable echo. The resultant modification to CHARPROC's functions is indicated by the word "modified" in the right Stream of Figure 4-2.



'L J y h**4€2„** Asynchronous Terminal Streams

Since CHARPROC is now performing no function for port t o, it might have been popped from the Stream to be reinserted by *getstrm* at the end of connection. However, the low overhead of S T S does not require its removal. The module remains on the Stream, passing messages unmodified between ASCEBC and the driver. At the end of the connection, *getstrm* restores this Stream to its idle confi g ration of Figure 4-1 by popping the added modules and then sending an I_S T to CHARPROC to restore the echo default.

Note that the T driver shown in Fi g re **4-2** handles minor devices. Each minor device has a distinct Stream connected from user space to the driver. This ability to handle multiple devices is a standard STRE A S feature, similar to the minor device mechanism in character I/O device drivers.

lf hvnato vnaeR td f.etoa

The previous example illustrates basic STRE A S concepts. Alternate, more efficient, STRE A S calls or mechanisms could have been used in place of those described earlier. Some of the alte r atives are described in Chapter 6 and others are addressed in the *STREAMS Programmer's Guide.*

For example, the initialization pr o ess that created a *getstrm* for each TTY port could have been implemented as a "supergetty" by using the STREAMS-related a] Tsystem call. As described in Chapter 6, a] Tallows a single process to efficiently monitor and control multiple Streams. The "supergetty" process would handle all Stream and terminal protocol initialization and create application processes only for established connections.

The M PROTO notification sent to *getstrm* could have been sent by the driver as an M_SIG message that causes a specified signal to be sent to the process. As discussed previously under *Message Types,* error and status information can also be sent upstream from a driver or module to user processes via different message types. These messages are transformed by the Stream head into a signal or error code.

Finally, an I_STR command could have been used in place of a putmsg M_PROTO message to send information to a driver. The sending process must receive an explicit response from an I_STR by a specified time period or an error is retu r ed. A response message must be sent upstream by the destination module or driver to be translated into the user response by the Stream head.

5 Kernel Level Functions

4A<8"D A,84L . 'Y

$>> ($>L . &Y
Message Allocation .)Y

DAL4#L $<F, $L<9 $#D<$>L . -Y
Put Procedures . -Y
Service Procedures . .Y

<4/L<8 $>>,4)L . /Y
Read Side Processing . 2Y
Driver Processing . 2Y
a ARPROC . 2Y
CANONPROC . 6Y
Write Side Processing . 6Y
Analysis . &$Y

ı ⊑S t

T is chapter introduces the use of the S T A S mechanism in the kernel and describes some of the t o ls provided by S T E A S to assist in the development of modules and drivers. In addition to the basic message passing mechanism and Q U U Stream linkage described previously, the S T AMS mechanism consists of various facilities including buffer management, the S T E A S scheduler, pr o essing and message priori t , flow control, and multiplexing. Over 30 S T S utili t routines and macros are available to manipulate and utilize these facilities.

The key elements of a S T EAMS ke r el implementation are the processing routines in the module and drivers, and the preparation of required data structures. The structures are described in the *STREAMS Programmer's Guide.* The following sections provide further information on messages and on the processing routines that operate on them. The example of Chapter 4 is continued, associating the user-level operations described there with kernel operations.

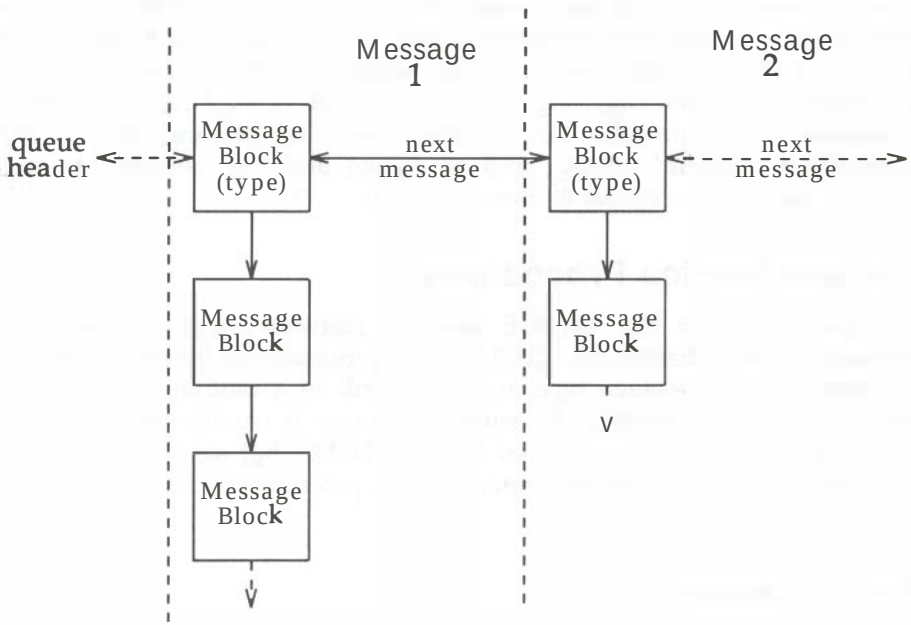
dv ooefv oa

As shown in Figure 5-1, a S T E A S message consists of one or more linked message blocks; i.e., the first message block of a message may be attached to other message blocks that are part of the same message. Multiple blocks in a message can occur, for example, as the result of processing that adds header or trailer data to the data contained in the message, or because of message buffer size limitations that cause the data to span multiple blocks. When a message is composed of multiple message blocks, the message t pe of the first block determines the type of the entire message, regardless of the types of the attached message blocks.

' OyJh **6-1.** A Message

S T S allocates a message as a single block containing a buffer of a certain size (see the next section). If the data for a message exceeds the size of the buffer containing the data, the procedure can allocate a new block containing a larger buffer, copy the current data to it, insert the new data and deallocate the old bl o k. Alte r ately, the procedure can allocate an additional (smaller) bl o k, place the new data in the new message block and link it after or before the initial message block. Both alternatives yield one new message.

Messages can exist standalone, as shown in Figure w,hen the message is being processed by a procedure. Alte r ately, a message can await processing on a linked list of messages, called a message queue, in a E R / R / È n Figure **5-2,** Message ¡s linked to Message **2.**



' N J y 5-2„ 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 containing a link to the second block of the message (if present). The message queue head and tail are contained in the E R / R / È

S T E A S utility routines enable developers to manipulate messages and message queues.

dg ooefga eaaedepbita

S T S maintains its own storage pool for messages. „procedure can request the allocation of a message of a specified size at one of three message pool priorities. The ; T ⅁ ⁴ utility retu r s a message containing a single block with a buffer of at least the size requested, providing there is a buffer available at the priority requested. When requesting priority for messages, developers must weigh their process' need for resources against the needs of other processes on the same machine.

Message pool priori t usually has no effect on allocation until the pool falls below inte r al S T thresholds. When this o curs, ; S S] may refuse a lower priority request for a message of size "x" while granting a higher priority request for the same size message. As examples of priority usage, storage for an urgent control message, e.g., an M HANGUP or M_PCPROTO, could be requested at high prio rty. A M DATA buffer for holding input might be requested at medium priority, and an output buffer (presuming the output data can wait in user space) at lowest priori t .

tn.a nSyahgnTAça tnv cgynngra

The pr o edures in the Q U UE are the software routines that process messages as they t ansit the QUE U . T e pr o essing is generally performed according to the message type and can result in a modified message, new message(s) or no message. A resultant message is usually sent in the same direction in which it was received by the QUEUE, but may be sent in either direction. A QUEUE always contains a put pr o edure and may also contain an ass o iated service procedure.

hn.a hgemgSn ggoa

A put pr o edure is the QUEUE routine that receives messages from the preceding QUEUE in the Stream. Messages are passed be t een 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 generally the only way to pass messages between modules (unless otherwise indicated, "modules" infers "module, driver and Stream head"). QUEUEs in pushable (see Chapter 3) modules contain a put pr o edure. In general, there is a separate put procedure for the read and write Q U UEs in a module because of the "full duplex" operation of most Streams.

A put procedure is associated with immediate (as opposed to deferred, see b low) processing on a message. E ch module accesses the adjacent put procedure as a subroutine. For example, consider that modA, modB, and modC are three consecutive modules in a Stream, with modC connected to the Stream head. If modA receives a message to be sent upstream, modA pr o esses that message and calls modB's put pr o edure, which processes it and calls modC's put procedure, which pr o esses it and calls the Stream head's put procedure. Thus, the message is passed along the Stream in one continuous processing sequence. On one hand, this sequence has the benefit of completing 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 multi-user system, this manner of processing may be good for this Stream but may be detrimental for others because they may have to wait "too long" to get their turn.

In addition, there are situations where the put procedure cannot immediately process the message but must hold it until processing is allowed. The most typical examples of this are a driver that must wait until the current output completes before sending the next message and the Stream head, which may have to wait until a process initiates a **read(2)** on the Stream.

rbn TA lmatn vmbynnbra

S T allows a service procedure to be contained in each QUEUE, in addition to the put procedure, to address the above cases and for additional purposes. A service procedure is not required in a QUEUE and is associated with deferred processing. If a QUEUE has both a put and service procedure, message processing is usually divided between the procedures. The put procedure is always called first, from a preceding QUEUE. After the put procedure completes its part of the message processing, it arranges for the service procedure to be called by passing the message to the aysfo routine. aytgl does two things: it places the message on the message queue of the QUEUE (see Figure 5-2) and links the QUEUE to the end of the STREAMS scheduling queue. When aytgreturns to the put procedure, the procedure typically exits. Some time later, the service procedure is automatically called by the STREAMS scheduler.

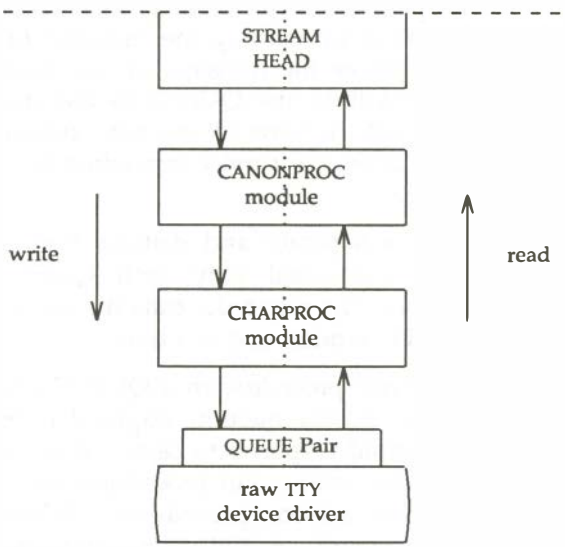
The STREAMS scheduler is separate and distinct from the SYSTEM V/88 process scheduler. It is concerned only with QUEUEs linked on the ST R S scheduling queue. The scheduler calls the service procedure of the scheduled QUEUE in a FIFO manner, one at a time.

Having both a put and service procedure in a QUEUE enables STREAMS to provide the rapid response and the queuing required in multi-user systems. The put procedure allows rapid response to certain data and events, such as software echoing of input characters. Put procedures effectively have higher priority than any scheduled service procedures. When called from the preceding STREAMS component, a put procedure executes before the scheduled service procedures of any QUEUE are executed.

The se r ice pr o edure implies message queuing. Queuing results i Œ È e j m k • • j processing of the se r ice pr o edure, following all other QUEUEs ` ´ • • j Œ ® • „ Æ È the scheduling queue. For example, terminal output and i Œ – t ´ h • \ § ] È g È x , „ È processing would typically b performed in a ª h ¤ w ` – h è ‘ ` h e ´ ª h ` È ´ § j ® u { § È type of pr o essing d o s not have to b Y s timely as j ` s “ x Œ r R È § j “ ê m \ È § j ¡ { þ È procedure also allows pr o essing time to b † ‘ • h È m · j • , À ¦ È – • j \ f È ‰ “ • q È multiple Streams. As with the put pr o edure there is ´ § ´ \ , , À \ È § j ˜] ¡ \ ® k È se r ice pr o edure for each QUEUE in a module. T e œ ¼ È ‘ Œ ¯ • ‡ ¡ È t] • { ‰ È (see Chapter 6) uses the se r ice pr o edures.

ug nSgiahned goo .S wa

Figure 5-3 continues the example of Chapter 4, describing STREAMS • j • • j „ È operations and ass o iates them, where relevant, with Chapter 4 ´ § j • „ k ¸ k „ È s À t jm calls in ¯ s jh È ¼ \ ‡ – , h ¦ È Y è h ¦ ´ o f – ê È t i \ i z i • r È – j • \ ¬ x “ Œ • f È µ © v { • q È \ è ‘ e ´ , h ¯ È s J Œ • j \ ‡ m È “ œ “ è ‘ • È È \ ¦ È t j ê “ , , “ ¼ x Œ • m È x r ´ • \ ® x “ • È



(N J { l **5-3.** „ Operational l ¬ œ j \ ‡ n È b ¾ È m – j È

As shown in Figure 5-3, the upstream QUEUE is also referred to as the read QUEUE, reflecting the message flow in response to a hC;@ystem call. Correspondingly, downstream is referred to as the write QUEUE. Read side pr o essing is discussed first.

cgnya r.y gatn vmgrr .Swa

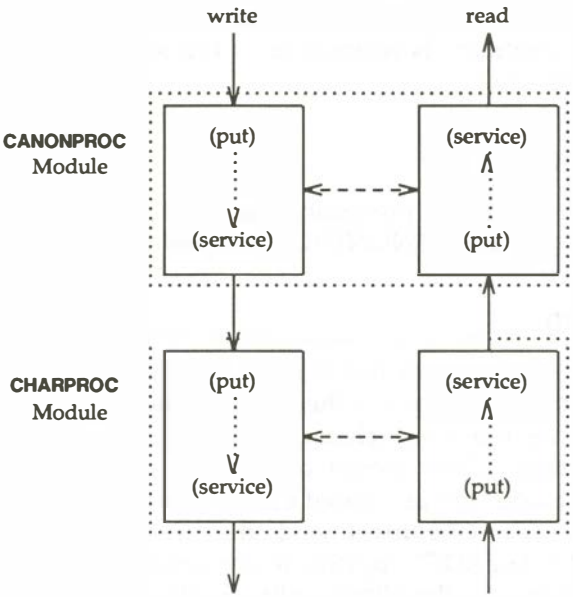
In our example, read side pr o essing consists of driver processing, CHARPROC pr o essing, and C A ONPROC pr o essing.

Driver Processing

In J 2 . .QO % B 9 . JQ2 .LQF . CBQ A . F F2Q% F Q9 A 6 . ÞnQ J 2 .getmsg(2) adaa.Fe -)EEe 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. Upon receipt of an input character interrupt from the port, the driver places the associated character in an M_DATA message, all o ated previously. T en, the driver sends the message to the CHARPROC module by calling CH A PROC's upstream put procedure. On return from CHARPROC, the driver calls the ;SS^ routine to get another message for the next character.

#

CHARPROC has both put and service pr o edures on its read side. In Figure 5-4, the other QUEUEs in the modules also have put and se r ice procedures.



(P l y **5-4.** „ ? ‘ e ³ , h Œ ³ «YÈŒE ehÈœe » wCaœh‘Èa h e ³ œ h ¥ È

W s h Ž È«s h È œe w · ha œ,È¸¥–È6 # H CH B +¥ ÈÊœe h YQ¢UÈEUE —³ «–È¸ œe ‘ ` h e ³ œ«s h È – œ ’ a h e ³ œ«s h È a €–¥ Èe w ·eY¥«‹hN¸ÈŸ p ¥w ÈŒ«s È‹QÈUEUE. 6 h œ h« s È mÈ¸ Y pw¥ŒÈ e x ` Y « « s Y h È a s ‘ w ŒÈw p¥ŒÈ Ÿ^ h È– h œe l ‘ œ † be eh È Y«¸s ŸÈ h È a s ‘ w ŒÈx p ‘È— « x ‘ ŒEYYŒ¸ÈeYÈœe h È ½ ’ œ € w ½Œ|p«È s-È œe s ŸÈ œe e ½ Y« se Y h a È Ÿ Œ Œ Y ³ «Ÿ È † Y « ·wÁ `È Y ` s ‘ È 6 " H CH B + È a Y ³ ¥ h «¥ s È eh È a s ‘« Ÿ È h È œe Y Œ ¥ † ·w Y« a ·¥ È È s h «È h œe † w^ŒÁ È M x È œe ¥ € È x Œ Y p È — Á È ’ l È« s h †È h ¥ ¥ Y p ½ h Ÿ È« s Y È T E A S ³ « w , w ·M Á s h È Œ È– Œ È# H CH B ³ ¥ È h ¥ Y È Œ ‘ « s h œ ³ « w , w «’Æ Ÿ^ « Y w «Œ s È h Ÿ È e e œ e h ¥ ¥ È È w « ¥ ½ Œ È½ œ e w Q h È UEUE. 4 w Œ Y , «Á s h È È + 6 H CH B + œ È h Y– ê È– È œ ‘ a h e a ² Ÿœ ¥ h È È« ¼ ¥ È e w –«² k– È œ ‘ a h e Y² Œ È a – È Y ¥ ¥ x h «¥ È È h † h ¥ ¥ Y p a h ‘ È– Á L È s h ½ œ e w –« be È ` h e ¥ œ Œ È È« ¥ s È h È h ¥ ¥ Y p « h È s h e È œ e x º « h ‘ œ È È m h ` « « s h h È a ‘Y Œ Œ e È s È h Œ È È « ³ ¢« ¥ È s h œ È h Y– e œ ‘ ` h e ³ œ h È

Q w ¥– È Y œ é« È œ È h Y¥ e x È e h – È œ ‘ a h ¥ ¥ w ¥ Œ Œ È p – È¸ h † h Œ ½ « x h « e s È È³ «– È œ ‘ ` h e ³ ¥ œ È È È s ŸÈ « « s h h È Œ È « w – œ e h È a h ¥ ¥ ¥ w ¥ s È È p Œ È ` h a È a ³ œ ¥ ¥ È Ÿ Œ È h È¼ « h Œ È ¥ m ¥ È Œ È s h È È œ e x º w Œ È È— ³ « a s Y œ e Y a w « Œ È e h È œ e d M ³ s – h – È6 # H CH B e È h Y Ÿ e Œ È e ½ È œ e x – ¬ h ² È« È œ e ‘ ` h e ³ Ÿ œ e h – ¥ È h Y œ [ ¥ È ³ ^ œ ‘ ³ « x Œ È È h ¥ ¥ È« w h Œ È È h ` È ¥ h l È È« s h ½ È œ e w –« be È a h e ³ « d s È h s h È È œ e x º h M œ s Ÿ È È Š Y Œ Œ ‘ h l È e– È œ e ‘ a h ¥ ¥ w ¥ È È È p « È h Œ È« e È– e È ‘ e ³ « ls È a È s Y œ e Y h « ls t Œ È È Ÿ È È x Œ x ‡ Y , È « w † l h Œ È Y ^ h È

After retu r ing from echo pr o essing, the CHARPROC read put procedure checks another of its private data fags and determines that parity checking should be performed on the input character. You should check parity as part of echo processing. However, for this example, pari t is checked only w hen the characters are sent upstream. This relaxes the timing in w hich the checking must o cur, i.e., it can b deferred along with the canonical processing.

CHARPROC uses **putq** to schedule the (original) message for parity check pr o essing by its read se r ice pr o edure. When the CHARPROC read se r ice pr o edure is complete, it fo r ards the message to the read put procedure of CANONPROC. Note that if parity check ng is not required, the CH A PROC put pr o edure calls the CANONPROC put pr o edure directly.

#

C A ONPROC perfo r s canonical pr o essing. As implemented, all read QUEUE processing is perfo r ed in its service procedure so that CANONPROC's put pr o edure simply calls b z t g„schedule the message for its read service pr o edure and then exits. The se r ice procedure extracts the character from the message buffer and place it in the '1ine buffer" contained in another M_DATA message it is constructing. Then, the message that contained the single character is retu r ed to the buffer pool. If the character received w as not an end-of-line, CANONPROC exits. Othe r ise, a complete line has been assembled and CANONPROC sends the message upstream to the Stream head which unblocks the user process from the I C s X oda„ll and passes it the contents of the message.

pe.Tga h.Sga tn emgoo .t w a

The write side of this Stream carries two kinds of messages from the user process: L] = tnSe„ssages for CHARPROC, and M_DATA messages to be output to the terminal.

L] = snRe„ssages are sent downstream as a result of an I_S T L] = sySt„em call. When CHARPROC receives an L] = tnRe„ssage type, it processes the message contents to modi f inte r al QUEUE flags and then uses a utility to send an acknowledgement message upstream (read side) to the Stream head. The Stream head acts on the acknowledgement message by unblocking the user from the L] = s„R

For terminal output, it is presumed that M_DATA messages, sent by •hQC„ system calls, contain multiple characters. S T usually retu r s to the user pr o ess immediately after pr o essing the •h L ucEl„ so that the process may send additional messages. Flow control, described in the next chapter, will eventually bl o k the sending pr o ess.

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 pr o edure, the pr o edure uses b y t go„ 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 se r ice 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.

ateay o.oa

For reasons of efficiency, a module implementation avoids placing one character per message and using separate routines to echo and parity check each character, as was done in this example. However, this design does yield potential benefits. Consider a case where alte r ate, more intelligent port hardware was substituted. If the hardware processed multiple input characters and performed the echo and parity checking functions of CHARPROC, the new dr ver could b implemented to present the same interface as CHARPROC. Other modules, e.g., C A ONPROC, could continue to be used without modification.

6 Other Facilities

6 A < 8 " D ! A - 8 6 L 1 &Y

% > > ' % LD % D % l< - 8 < - A K L 1 &Y

/ 8 H L 8 6 A < 8 / L 1 (Y

D / A - ; 0 % J - 6) L 1 .Y

8 4 - A 8 < - 4) L 1 3Y

< < 8 < l6 " L < ! % L 8 ' ' - 7 ' L 1 5Y

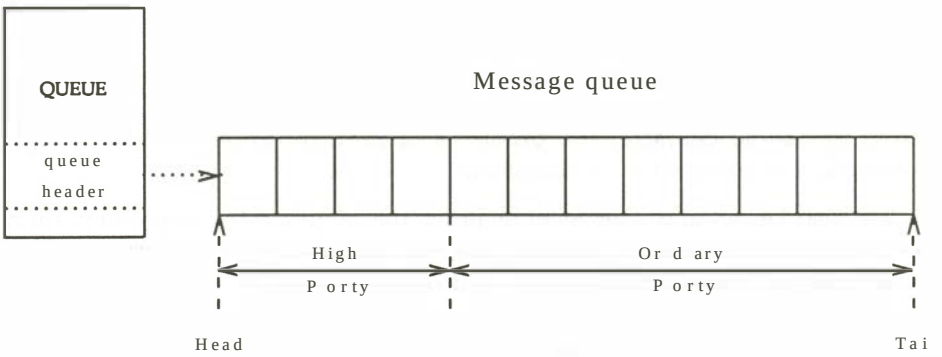
The previous chapters described the basic concepts of constructing a Stream and utilizing the S T EAMS mechanism. Additional STRE A S features are provided to handle characteristic problems of protocol implementation, e.g., flow control, and to assist in development.

There are also ke r el and user-level facilities that support the implementation of advanced functions, e.g., multiplexors, and allow asynchronous operation of a user process and S T EAMS input and output.

dgoonwga ongnga tg .e g.Tta

As mentioned in the previous chapter, the STREAMS scheduler operates strictly FIFO so that each QUEUE's service procedure receives control in the order it was scheduled. When a service procedure receives control, it may encounter multiple messages on its message queue. This buildup can occur if there is a long inte r al between the time a message is queued by a put procedure and the time that the STREAMS scheduler calls the associated service procedure. In this interval, there can be multiple calls to the put procedure causing multiple messages. The service procedure always processes all messages on its message queue unless prevented by flow control (see next section). Each message must pass through all the modules connecting its origin and destination in the Stream.

If se r ice procedures were used in all QUEUES and there was no message priority, the most recently scheduled message is processed after all other scheduled messages on all Streams are processed. In certain cases, message types containing urgent information (e.g., break or alarm conditions) must pass through the Stream quickly. To accommodate these cases, I M G / ! > K È provides two classes of message queuing priority: ordinary and high. STREAMS prevents high-priority messages from being blocked by flow control and causes a service procedure to process them ahead of all ordinary priority messages on the procedure's queue. This results in the high-priority message transiting each module with minimal delay.



'NJy 6-1. „ Streams Message Priori t

The prio rty mechanism operates as shown in Figure 6-1. Message queues are usually not present in a QUEUE unless that QUEUE contains a service pr o edure. When a message is passed to a y t g, to schedule the message for service pr o edure processing, a y t g places the message on the message queue in priority order. High priority messages are placed ahead of all ordinary p rority messages, but behind any other high priority messages on the queue. S T EAMS utilities deliver the messages to the processing service procedure FIFO within each priority class. T e service procedure is unaware of the message priority and simply receives the next message.

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

sae o ane tTge aa

Even on a well-designed system, general system delays, malfunctions, and excessive message accumulation on one or more Streams can cause the message buffer pools to become depleted. In addition, processing bursts can arise when a service procedure in one module has a long message queue and

pr o esses all its messages in one pass. S T EAMS provides two independent mechanisms to guard its message buffer pools from being depleted and to minimize long processing bursts at any one module.

NOTE

Flow control is only applied to normal priority messages (see previous section) and not to high priority messages.

T e first f ow control mechanism is global and automatic and is related to the message p o l priority, discussed in the *Message Storage Pool* section of Chapter 5. When the Stream head requests a message buffer in response to a **putmsg** or € h L ssy Cstem call, it uses the lowest level of priority. Since buffer availability is based on priori t and buffer pool levels, the Stream head is among the first modules refused a buffer when the pool becomes depleted. In response, the Stream head blocks user output until the STRE A S buffer pool recovers. As a result, output has a lower priority than input.

The second flow control mechanism is local to each Stream and adviso r (voluntary), and limits the number of characters that can be queued for processing at any QUEUE in a Stream. This mechanism limits the buffers and related processing at any one QUEUE and in any one Stream, but does not consider buffer pool levels or buffer usage in other Streams.

T e advisory mechanism operates between the two nearest QUEUEs in a Stream containing service procedures (see Figure 6-2). Messages are usually held on a message queue only if a service procedure is present in the associated QUEUE.

Messages accumulate at a QUEUE when its se r ice 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, which are set when a developer specifies high-water and low-water control values for the QUEUE. 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.

Flow control operates as follows:

1. Each time a STREAMS message handling routine (e.g., putq, getq, adds or removes a message from a message queue in a QUEUE, the limits are checked. STREAMS calculates the total size of all message blocks 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 6-2, if D is full and C has no service procedure, then B is the nearest preceding QUEUE.

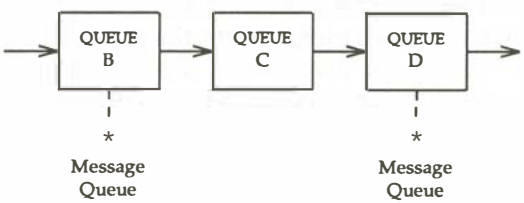


Figure 6-2. Flow Control

4. The service procedure in 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 B from further execution. B remains blocked until the low-water mark of the full QUEUE, D, is reached.

5. While B is blocked, any non-priority messages that arrive at B accumulate on its message queue (recall that priority messages are not blocked). B can reach a full state and the full condition propagates back to the last module in the Stream.

6. When the service procedure processing on D causes the message block total to fall below the low water mark, the full indicator is turned off. STREAMS automatically schedules the nearest preceding blocked QUEUE (B in this case), getting things moving again. This automatic scheduling is known as back-enabling a QUEUE.

Note that to utilize flow control, a developer need only call the utili t that tests if a full condition exists ahead, plus perform some housekeeping if it does. Everything else is automatically handled by S T E A S. Additional flow control features are described in the *STREAMS Programmer's Guide.*

Multiplexing

S T multiplexing supports the development of interne t orking prot o ols, e.g., IP and ISO CLNS, and the pr o essing of interleaved data streams, e.g., in SNA, X.25, and te r inal window facilities.

S T multiplexors (also called pseud o device drivers) are created in the ke r el by interconnecting multiple Streams. Conceptually, there are t o **kinds of multiplexors that developers can build with STREAMS:** upper and lower multiplexors. Lower multiplexors have multiple lower Streams be t een device drivers and the multiplexor, and upper multiplexors have multiple upper Streams between user processes the multiplexor.

Figure 63 shows an example of a lower multiplexor. This configuration typically occurs where inte r etworking functions are included in the system. This Stream contains three types of drivers: the Ethe r et, L B, and IEEE 802.2; hardware device drvers that terminate links to other nodes. The Inte r et Protocol (IP) is a multiplexor.

The IP multiplexor switches messages among the varous nodes (lower Streams) or sends them upstream to user processes in the system. In this example, the multiplexor expects to see an 802.2 interface downstream; for the Ethe r et and L B drivers, the Net 1 and Net 2 modules provide se r ice inte r aces to the two the non-802.2 drivers and the IP multiplexor.

Figure 63 depicts the IP multiplexor as part of a larger Stream. The Stream, as shown in the dotted rectangle, usually has an upper TCP multiplexor and additional modules. Multiplexors could also b cascaded below the IP driver if the device drivers were replaced by multiplexor drivers.

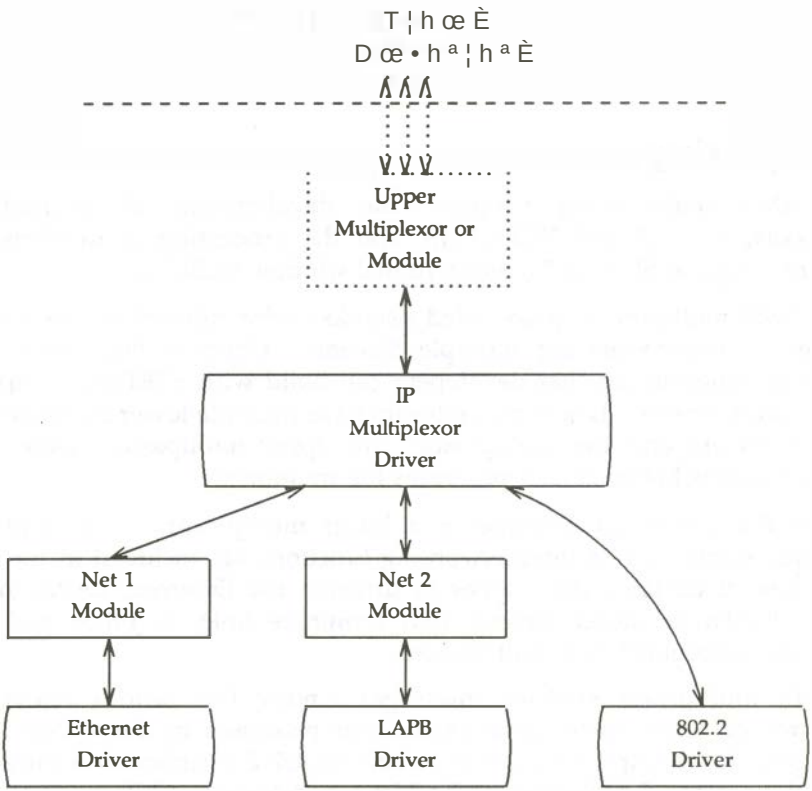
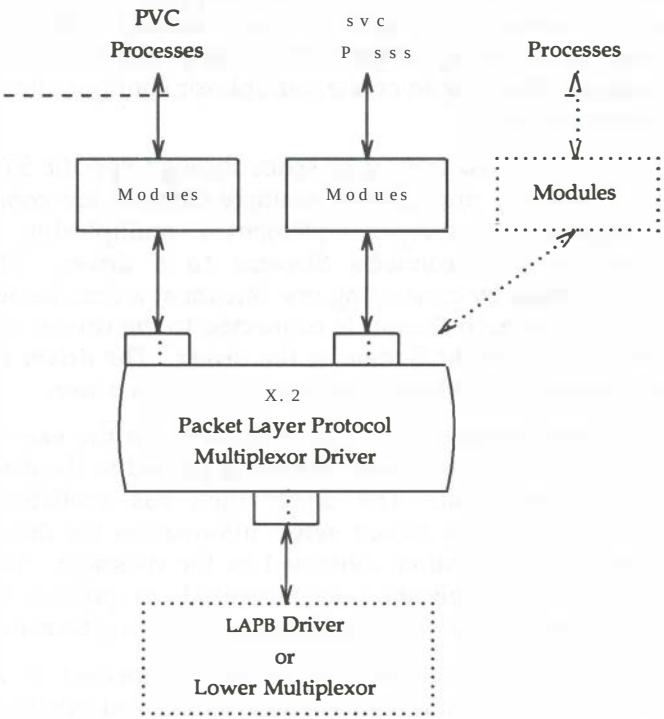


Figure **6-3.** 7 Œ « h ¢=Hª⁄ƒ ß w – ƒ hl¾ wh Œ Y p t È

5 w p ³ •6h- È ¦ s ‘ ¼ ¦ ÐÝ Œ³È– h œ È ƒ « } – ƒ h ¾ ‘ Œ ß «ÈsÈw `¦ ‘ÐŒ m w p ³ • Y « ẁh ‘ÐŒ w È¿j • È
œ ‘ ³ « h†¦h È¦ ¦ Y p h^¦ È « ¼ h h«Œ È h ß È ¼ h bœ È h Y †Ý ÐŒ é Œ h ‘ È m «ß s h³ È– j • È « • h Z ‡ ¦
M s w l¦«È œ h Y–†h È• m ‘ • Ø¦25 † ³ ƒ « w – ƒ h ¾ ‘ Ø Œ ƒp«È w — ƒ h È w Œ f h – h Œ f h Œ
V w • « ³ -Ywƒ œ È ` ³ w V« È Ø Œ Œ È h V Œ « «È³ -Ywƒ •È ³ w Œ Ø - ¶È¦ j • È– • “ b j ¦ ¦ j ¦ È
R – – h †Ø ƒ « w – ƒ h ¥ •‘ h Ø ÐŒ – j ` w mÝw – ` È ƒ w ` Y‘ «m È« Œ Œ f Y † M È / ! > J È Z b ~ , ~ ¬
« s Y¦« È – ‘ • « È † ³ ƒ « ~ – ƒ h È † w Œ Œ È ƒ h w ẁ h È ę È M ş h m¦ ÐÈq ¶Z •, h ¦‘ÐÈ “ ¼ ¦ È
« s Y « È • h È ‘ † – , h ¾ ` È Œ m ~ p ³ • Y « Ẁ Œ Ø Ø h Ø ³ w , « Æ È Z ¸ w Œ ‘ p Œ h ‘ È • È ‡ “ • j È
† ³ ƒ « w – , h « (h È È • w ¸ h ^¦ h È ƒ ‘ ½ Ø Ð Ø ę È ƒ « w – † ƒ h ¶ È ƒ h ¦ Ý È ^ ‘ ¸ j È



'L J y j6-4. X.25 Multiplexing Stream

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 6-4, with a window driver replacing Packet Layer, a tty driver replacing LAPB, 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 of Figure 6-2 has a different configuration than the X.25 multiplexor and the driver would implement its own set of pr o essing and routing requirements.

In addition to upper and lower multiplexors, more complex configurations can b 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. Instead, S T EAMS provides a general purpose multiplexing facility. T e facility allows users to set up the inter-module/driver plumbing to create multiplexor configurations of generally unlimited interconnection.

T e connections are created from user space through specific STREAMS Q = s S „ system calls. In a lower multiplexor, multiple Streams are connected below an application-specific, developer-implemented multiplexing driver. The multiplexing facility only connects Streams to a driver. The Q = s S all configures a multiplexor by connecting one Stream at a time below the opened multiplexor driver. As each Stream is connected to the driver, the connection setup pr o edure identifies the Stream to the driver. T e driver usually stores this setup information in a private data structure for later use.

Subsequently, when messages flow into the driver on the various connected Streams, the identity of the associated Stream is passed to the driver as part of the standard procedure call. The driver then has available the Stream identification, the previously stored setup information for this Stream, and any inte r al routing information contained in the message. These data are used, according to the aCCIgoativn iml lt mtS tt d, avUlwv otbbU artU gyovmpy nU message and route the output to the appropriate outgoing Stream.

In addition, new Streams can b dynamically connected to an operating multiplexor without interferring with ongoing traffic, and existing Streams can b disconnected with similar ease.

de S.fe n.Sfa

S T EAMS allows user processes to monitor and control Streams so that system resources (e.g., as CPU cycles and process slots) can be used effectively. Monitoring is especially useful to user-level multiplexors, in which a user process can create multiple Streams and switch messages among them (similar to S T EAMS ke r el-level multiplexing, described previously).

User processes can efficiently monitor and control multiple Streams with t o S T EAMS system calls: a ^ S Sand „the Q = s S I_SETSIG command. These calls allow a user process to detect events that occur at the Stream head on one or more Streams, including receipt of a data or protocol message on the read queue and cessation of flow control.

Synchronous monitoring is provided by use of c^SaSo„ne; here, the user process cannot continue processing until after the system call completes. When the calls are used together, they allow asynchronous, or concurrent, operation of the pro ess and STRE A S input output. This allows the user process to monitor the Stream while ca r ying on other activities.

To monitor Streams with b]SS u„ser process issues that system call and specifies the Streams to b monitored, the events to look for, and the amount of time to wait for an event. b]SbSo„ks the process until the time expires or until an event o curs. If an event o curs, b]SreSu„r s the type of event and the Stream on which the event occu red.

Instead of waiting for an event to o cur, a user process may want to monitor one or more Streams while processing other data. It can do so by issuing the L]?s_SE„TSIG command, specifying one or more Streams and events (as with c]SV U„nlike a c^StSis„ L^>dosSs„ not force the user process to wait for the event but retu r s immediately and issues a signal when an event occurs. The process must also request nNJ[;Sor „nNJnDso ca„tch the resultant SIGPOLL signal.

If any selected event occurs on any of the selected Streams, STRE A S causes the SIGPOLL catching function to be executed in all associated requesting processes. However, the process(es) will not know which event occurred, nor on what Stream the event occurred. A process that issues the I_SETSIG can get more detailed information by issuing a c^SaSe„r, it detects the event.

onnenantSalnedga ae ff.tfa

STRE A S includes error and trace loggers (see Figure 6-5) useful for debugging and administering modules and drivers.

A y module or driver in any Stream can call the STRE A S logging function nthS]de„scribed in S]J W„hen called, nshSsen„ds formatted text to the e ror logger nshChh t„he* trace logger nsh;?D U„or* b„oth. The call parameters for qskSin^Ju„de the module/driver identification, a severity level, and the formatted text describing the condition causing the call. The call also identifies the process nslCahn„d/„or nsh;?D „receive the resultant output message.

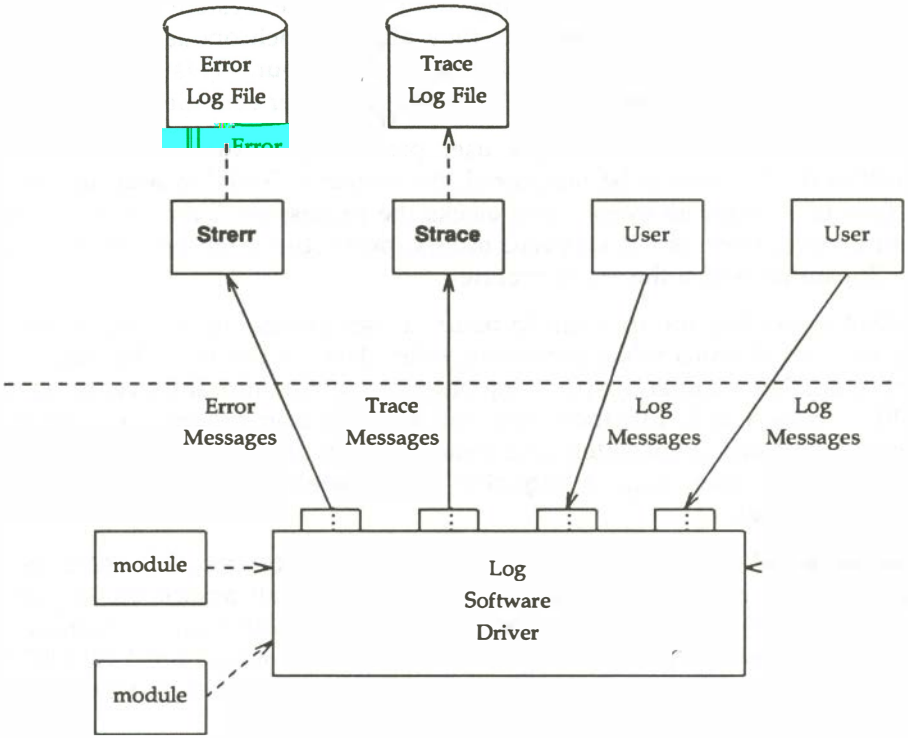


Figure 6-5: Error and Trace Logging

The error logger is intended to operate as a daemon process initiated at system startup. A call to the error function requesting an error to be logged causes an M_PROTO message to be sent to the error logger, which formats the contents and places them in a daily file. The utility errdead is provided to periodically purge aged, unreferenced daily log files.

A call to the trace function requesting trace information to be logged causes a similar M_PROTO message to be sent to the trace logger, which places it in a user designated file. strace is intended to be initiated by a user. The user can designate the modules/drivers and severity level of the messages to be accepted for logging by the trace logger.

A user process can submit its own M_PROTO messages to the log driver for inclusion in the logger of its choice through bytWnl „The messages must b in the same format required by the lo g ing pr o esses and will be switched to the logger(s) requested in the message.

T e output to the log files is formatted, A CII text. The files can be processed by standard system commands such as HïCb or„C@ or„ by developer-provided routines.

7 Driver Design Comparisons

Introduction 2 &Y
Enviro n ent 2 &Y
D vers 2 &Y
Modules 2 (Y

This chapter compares operational features of character I/O device drivers with S T EAMS drvers and modules. It is intended for experienced developers of SYSTEM V/8 system character device drivers. Details are provided in the *STREAMS Programmer's Guide.*

-KSF OMKG9KRX

No user environment is generally available to S T EAMS module procedures and drivers. The exception is the module and driver open and close routines, rgitU bhUttutU tsitU suut AAIgUirtU eoswtsU ghUttU usuutynUlwgut AAeyiU can sleep. Otherwise, a S T driver, module put procedure, and module service procedure has no user context and can neither sleep nor access any u_area.

Multiple Streams can use a copy of the same module (i.e., the same fmodsw), each containing the same processing procedures. This means that module code is reentrant, so care must be exercised when using global data in a module. Put and service procedures are always passed the address of the QUEUE (e.g., in Figure 2-5 Au calls Bu's put procedure with Bu as a parameter). The processing procedure establishes its environment solely from the QUEUE contents, typically the private data (e.g., state information).

rn.Tgnoa

At the interface to hardware devices, character I/O drivers have interrupt entry points; at the system interface, those same drivers usually have direct entry points (routines) to process]cD[?„To]Dh„D;Awrite and ioctl system calls.

STREAMS device drivers have similar interrupt entry points at the hardware device interface and direct entry points only for]cD[and>^To Dsystem calls. These entry points are accessed via IMG/&IÈthe call formats differ from character device drivers. The put procedure is a driver's third ent r point, but it is a message (not system) interface. The Stream head translates write and N]?vaSs, into messages and sends them downstream to be processed by the drivers write QUEUE put procedure. hD;iAseen directly only by the Stream head, which contains the functions required to process system calls. !Èdriver does not know about system interfaces other than]cD[and ?To]Dbut it can detect absence of a hD;iAdirectly 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 drver is ready to send data or other information to a user pr o ess, it d o s not wake up the process. It prepares a message and sends it to the read QUEUE of the appropriate (minor device) Stream. The drivers open routine generally stores the QUEUE address co r esponding to this Stream.

For output pr o essing, the driver receives messages in place of a • h N uaD.„ If the message cannot b sent immediately to the hardware, it may be stored on the drivers write message queue. Subsequent output interrupts can remove messages from this queue.

Drivers and modules can pass signals, e r or codes, and retu r values to processes via message types provided for that purpose.

de SRagoa

As described previously, modules have user context available only during the execution of their open and close routines. Othe r ise, the QUEUEs forming the module are not associated with the user process at the end of the Stream, nor with any other process. Because of this, QUEUE procedures must not sleep when they cannot proceed; instead, they must explicitly return control to the system. The system saves no state information for the QUEUE. The QUEUE must store this information inte r ally ghgaghbvUwvott lU hwvmUrtU bemtU point on a later entry.

When a module or driver that requires private working storage (e.g., for state information) is pushed, the open routine must obtain the storage from exte r al sources. STREAMS copies the module template from GX^A for the ! PUSH, so only fixed data can be contained in the module template.

S T EAMS has no automatic mechanism to allocate working storage to a module when it is opened. The sources for the storage typically include a module-specific ke r el array, installed when the system is configured, or the STREAMS buffer pool. When using an array as a module storage pool, the maximum number of copies of the module that can exist at any one time must b determined. For drivers, this is typically determined from the physical devices connected, e.g., as the number of ports on a multiplexor. However, certain types of modules may not be associated with a particular exte r al physical limit. For example, the C A ONICAL module shown in Figure 2-4 could be used on different types of Streams.

G Glossary

Glossary

B_•\ntiC;W„
T e direction from Stream head to driver.

BiL|Fm„
T e end of the Stream closest to an exte r al interface. The principal functions of the drver are handling any associated device, and transforming data and information b t een the external interface and Stream. It can also b a pseudo-driver, not directly associated with a device, that performs functions inte r al to a Stream, e.g., as a multiplexor or log driver.

YCnn;IC„
One or more linked blocks of data or information, with associated S T EAMS control structures containng a message type. Messages are the only means of transferring data and communicating within a Stream.

WCnn;IfQy„DyC„
A linked list of messages connected to a QUEUE.

WCnn;ItQ„aC„
A defined set of values identifying the contents of a message.

W_@ySC„
Software that performs functions on messages as they flow between Stream head and driver. A module is the S T EAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (downstream and upstream) data flow and processing.

WySsLaSD,_i„
A mechanism for connecting multiple Streams to a multiplexing driver. rtU mtchangbm b,llgwabU aroUl wgutbbgynUghUgyaownoeiolUl eaeUiawoecAU eylU the processing of inte r etworking protocols. The multiplexing driver routes messages among the connected Streams. The other end of a Stream connected to a multiplexing driver is typically connected to a device driver.

aynK;<X^CA„ySD„
A module between the Stream head and driver. A driver is a non-pushable module and a Stream head includes a non-pushable module.

QUEUE

T e set of structures that forms a module. A module is composed of t o Q U UEs, a read (upstream) Q U U and a write (downstream) QUEUE.

h C ; f@y „C y C „
The message queue in a module or driver containing messages moving upstream. Ass o iated with input from a d r ver.

2 s h C ; W „
The ke r el aggregate created by connecting STRE A S components, resulting from an application of the S T S mechanism. The primary com p nents are a Stream head, a driver and zero or more pushable modules b t een the Stream head and driver. A Stream forms a full duplex processing and data transfer path in the kernel, between a user process and a driver. A Stream is analogous to a Shell pipeline except that data flow and processing are bidirectional.

3 s h C ;KWC„; A „
The end of the Stream closest to the user process. The Stream head provides the interface between the Stream and the user process. The principal functions of the Stream head are processing STRE A S-related system calls, and bidirectional transfer of data and information between a user process and messages in STRE A S' ke r el space.

2 7 1% " + 4 „
A ke r el mechanism that supports development of network se r ices and data communication drivers. It defines interface standards for character input toutput within the ke r el, and between the kernel and user level. The STRE A S mechanism comprises integral functions, utility routines, ke r el facilities and a set of structures.

y a o s h C ; X „
The direction from driver to Stream head.

• h N sf@C„D y C „
The message queue in a module or driver containing messages moving downstream. Associated with output from a user process.