R306-02

OpenVOS Communications Software: STREAMS Programmer's Guide

Stratus Technologies

Notice

Manual Name: *OpenVOS Communications Software: STREAMS Programmer's Guide*

Part Number: R306
Revision Number: 02
OpenVOS Release Number: 17.1.0
Publication Date: April 2011

Stratus Technologies, Inc.
111 Powdermill Road
Maynard, Massachusetts 01754-3409

Preface

The OpenVOS Communications Software: STREAMS Programmer's Guide (R306) documents the application program interface to the OpenVOS STREAMS software. It describes how applications use the OpenVOS STREAMS application program interface to communicate with STREAMS devices.

This manual is for programmers who write or design STREAMS applications under the OpenVOS operating system. Programmers are expected to be familiar with the STREAMS protocol and with the OpenVOS programming environment in order to use the OpenVOS STREAMS software.

Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual. Note, however, that change bars are not used in new chapters or appendixes.

In this revision, the following sections are new or have changed.

- In Chapter 1, "OpenVOS Standard C Language Support" and "Requests of the Analyze System Subsystem"
- In Chapter 2, "OpenVOS Standard C Language Functions" and "Polling Support"
- In Chapter 3:
  - "Sending Messages Using `s$putmsg/s$putpmsg`"
  - "C Language Interface"
  - `open`
  - `poll`
  - `putmsg/putpmsg`
  - `read`
  - `readv`
  - `select`
  - `select_with_events`
  - `write`
  - `writev`

In addition, minor changes appear throughout the manual.

Previous revisions of this manual contained a programming example and descriptions of the `dump_stream` and `search_streams` requests of the `analyze_system` subsystem. The descriptions of the `dump_stream` and `search_streams` requests s now appear in the *OpenVOS System Analysis Manual* (R073). The programming example has been removed.

Manual Organization

Chapter 1, "Introduction to OpenVOS STREAMS," briefly describes STREAMS and STREAMS-related concepts and terms, and introduces the OpenVOS STREAMS software.

Chapter 2, "OpenVOS STREAMS Programming Support," presents various elements of OpenVOS STREAMS programming support and other programming considerations for developing application programs for OpenVOS STREAMS.

Chapter 3, "Application Program Interface," documents the application program interface to OpenVOS STREAMS. OpenVOS STREAMS applications can use either the OpenVOS Standard C language interface or OpenVOS subroutines as the interface to OpenVOS STREAMS, as documented in this chapter.

Appendix A, "PL/I Usage," shows the OpenVOS STREAMS subroutine declarations in OpenVOS PL/I.

Related Manuals

See the following Stratus manuals for related documentation.

- *OpenVOS PL/I Subroutines Manual* (R005)
- *OpenVOS C Subroutines Manual* (R068)
- *OpenVOS System Administration: Configuring a System* (R287)
- *OpenVOS Commands Reference Manual* (R098)
- *OpenVOS PL/I Transaction Processing Facility Reference Manual* (R015)
- *OpenVOS C Transaction Processing Facility Reference Manual* (R069)
- *OpenVOS POSIX.1 Reference Guide* (R502)

For information about the OpenVOS Standard C (that is, the ANSI C-compliant) implementation of the C language, see the *OpenVOS Standard C User's Guide* (R364) and the *OpenVOS Standard C Reference Manual* (R363).

For information on POSIX.1, see the following documentation:

- *OpenVOS POSIX.1: Conformance Guide* (R217M)
- *OpenVOS POSIX.1 Reference Guide* (R502)

See also third-party publications for additional information on STREAMS.

Notation Conventions

This manual uses the following notation conventions.

- Italics introduces or defines new terms. For example:

  The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

  Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

  ```
  change_current_dir (master_disk)>system>doc
  ```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  `list_users -module` *`module_name`*

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

  `display_access_list system_default`

  ```
  %dev#m1>system>acl>system_default

  w  *.*
  ```

Format for Commands and Requests

Stratus manuals use the following format conventions for documenting commands and requests. (A *request* is typically a command used within a subsystem, such as `analyze_system`.) Note that the command and request descriptions do not necessarily include each of the following sections.

`name`
The name of the command or request is at the top of the first page of the description.

Privileged
This notation appears after the name of a command or request that can be issued only from a privileged process.

Purpose
Explains briefly what the command or request does.

Display Form
Shows the form that is displayed when you type the command or request name followed by `-form` or when you press the key that performs the `DISPLAY FORM` function. Each

field in the form represents a command or request argument. If an argument has a default value, that value is displayed in the form.

The following table explains the notation used in display forms.

The Notation Used in Display Forms

Notation	Meaning
	Required field with no default value.
	The cursor, which indicates the current position on the screen. For example, the cursor may be positioned on the first character of a value, as in `all`.
current_user *current_module* *current_system* *current_disk*	The default value is the current user, module, system, or disk. The actual name is displayed in the display form of the command or request.

Command-Line Form

Shows the syntax of the command or request with its arguments. You can display an online version of the command-line form of a command or request by typing the command or request name followed by `-usage`.

The following table explains the notation used in command-line forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Specifying multiple values is **not** the same as specifying a star name. When you specify multiple values, you must separate each value with a space.

The Notation Used in Command-Line Forms

Notation	Meaning
argument_1	Required argument.
argument_1...	Required argument for which you can specify multiple values.
{ *element_1* *element_2* }	Set of arguments that are mutually exclusive; you must specify one of these arguments.
[*argument_1*]	Optional argument.
[*argument_1*]...	Optional argument for which you can specify multiple values.
[*argument_1* *argument_2*]	Set of optional arguments that are mutually exclusive; you can specify only one of these arguments.

Notation	Meaning
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of arguments can contain more than two elements. Brackets and braces are sometimes nested.	

Arguments

Describes the command or request arguments. The following table explains the notation used in argument descriptions.

The Notation Used in Argument Descriptions

Notation	Meaning
CYCLE	This argument has predefined values. In the display form, you display these values in sequence by pressing the key that performs the `CYCLE` function.
Required	You cannot issue the command or request without specifying a value for this argument. If an argument is required but has a default value, it is not labeled **Required** since you do not need to specify it in the command-line form. However, in the display form, a required field must have a value—either the displayed default value or a value that you specify.
(Privileged)	Only a privileged process can specify a value for this argument.

Explanation

Explains how to use the command or request and provides supplementary information.

Error Messages

Lists common error messages with a short explanation.

Examples

Illustrates uses of the command or request.

Related Information

Refers you to related information (in this manual or other manuals), including descriptions of commands, subroutines, and requests that you can use with or in place of this command or request.

Format for Subroutines

Stratus manuals use the following format conventions for documenting subroutines. Note that the subroutine descriptions do not necessarily include each of the following sections.

`subroutine_name`
The name of the subroutine is at the top of the first page of the subroutine description.

Purpose
Explains briefly what the subroutine does.

Usage
Shows how to declare the variables passed as arguments to the subroutine, declare the subroutine entry in a program, and call the subroutine.

Arguments
Describes the subroutine arguments.

Explanation
Provides information about how to use the subroutine.

Error Codes
Explains some error codes that the subroutine can return.

Examples
Illustrates uses of the subroutine or provides sample input to and output from the subroutine.

Related Information
Refers you to other subroutines and commands similar to or useful with this subroutine.

Online Documentation

The OpenVOS StrataDOC Web site is an online-documentation service provided by Stratus. It enables Stratus customers to view, search, download, print, and comment on OpenVOS technical manuals via a common Web browser. It also provides the latest updates and corrections available for the OpenVOS document set.

You can access the OpenVOS StrataDOC Web site, at no charge, at `http://stratadoc.stratus.com`. A copy of OpenVOS StrataDOC on supported media is included with this release. You can also order additional copies from Stratus.

For information about ordering OpenVOS StrataDOC on supported media, see the next section, "Ordering Manuals."

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN™), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.

- Contact the Stratus Customer Assistance Center (CAC), using either of the following methods:

  – From the ActiveService Manager (ASM) web site, log on to your ASM account. Click the `Manage Issues` button, enter your site ID in the `Create New Issue For Site` box, and then click the pencil icon to the right of the box. Fill out the forms and select `Update`.

  – Customers in North America can call the CAC at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see `http://www.stratus.com/support/cac/index.htm` for Stratus CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

Commenting on This Manual

You can comment on this manual using one of the following methods. When you submit a comment, be sure to provide the manual's name and part number, a description of the problem, and the location in the manual where the affected text appears.

- From StrataDOC, click the `site feedback` link at the bottom of any page. In the pop-up window, answer the questions and click `Submit`.

- From any email client, send email to `comments@stratus.com`.

- From the ASM web site, log on to your ASM account and create a new issue as described in the preceding section, "Ordering Manuals."

- From an OpenVOS window, specify the command `comment_on_manual`. To use the `comment_on_manual` command, your system must be connected to the RSN. This command is documented in the manual *OpenVOS System Administration: Administering and Customizing a System* (R281) and the *OpenVOS Commands Reference Manual* (R098). You can use this command to send your comments, as follows.

  – If your comments are brief, type `comment_on_manual`, press Enter or Return, and complete the data-entry form that appears on your screen. When you have completed the form, press Enter.

  – If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press Enter or Return. Enter this manual's part number, `R306`, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press Enter.

  Note: If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail` request of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Figures

Tables

Chapter 1:
Introduction to OpenVOS STREAMS

This chapter briefly describes STREAMS and introduces basic STREAMS concepts and terms. It then describes the OpenVOS STREAMS software available on a Stratus OpenVOS system, and introduces the OpenVOS STREAMS application program interface. The chapter contains the following sections.

- "Overview of STREAMS"
- "Overview of OpenVOS STREAMS"

The following chapters and appendix provide additional information:

- Chapter 2 discusses programming considerations for using OpenVOS STREAMS.
- Chapter 3 presents detailed documentation on the application program interface to OpenVOS STREAMS.
- Appendix A, "PL/I Usage" shows the PL/I usage of some subroutines.

Overview of STREAMS

STREAMS is a set of software interfaces and facilities – routines, functions, and services – that defines a standard architecture for communications services. STREAMS enables communications protocol software implemented by independent vendors to interoperate or be combined without requiring substantial changes to the system software or application software. As a result, STREAMS facilitates the development, porting, and integration of communications software.

STREAMS was originally developed by AT&T, and incorporated into UNIX System V Release 3 (UNIX V.3) to enhance the character I/O subsystem and to provide support for communications software development. The AT&T implementation of STREAMS is distributed beginning with UNIX System V.3, and is available on other vendor systems.

The following sections provide additional information on STREAMS.

- "STREAMS Terminology"
- "Modules and Drivers"
- "Service Interfaces"

STREAMS Terminology

A *Stream* is a communications path between a STREAMS driver in kernel space and a user process. A Stream consists of a Stream head, a STREAMS driver, and optionally, one or more STREAMS modules.

> **Note**: The term STREAMS (in all capitals) refers to the protocol standard, while the term Stream (initial capital) refers to a communication path between a STREAMS driver and a user process.

A *Stream head*, always at one end of a Stream, serves as the interface between a Stream and a user process.

A *STREAMS driver* is either a device driver to an external device or a pseudo-device driver, which is not directly associated with an external device. STREAMS drivers typically handle data transfer between the operating system and the device. Pseudo-device drivers is used to implement multiplexers.

A *STREAMS module* consists of routines and internal data structures used to process data flowing on the Stream. User messages and device data, control, and status information can be processed by a STREAMS module. Each module is self-contained and communicates with neighboring components by passing messages that are formatted according to STREAMS standards.

Figure 1-1 illustrates a basic Stream containing a single STREAMS module.

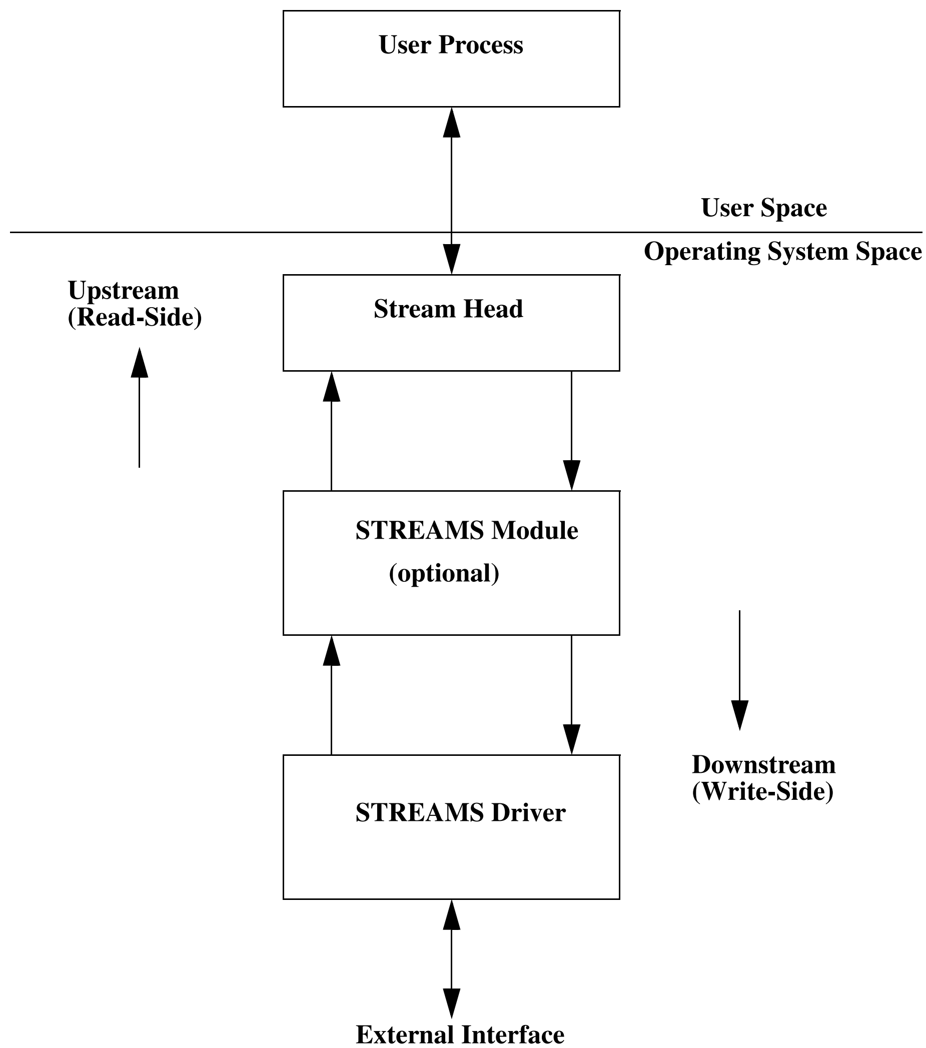

Figure 1-1. A Basic Stream

Messages flowing from the Stream head toward the driver travel *downstream*, and messages moving toward the Stream head travel *upstream*. A message is actually a set of data structures used to pass data on a Stream.

Each STREAMS module and driver uses a pair of message queues to hold messages; one queue contains messages traveling upstream (read-side queue) and the other contains messages traveling downstream (write-side queue). STREAMS modules and drivers can flow control the messages passed from these message queues to regulate the flow of data in a Stream.

A user process opens a Stream much as it opens any device or file. When a STREAMS device is opened, a Stream is created and connected to a STREAMS driver, and the message queues are created and initialized. The user process can then exchange data with the STREAMS device. Other processes may also open the same STREAMS device and share the Stream created by the process that initially opened the device, if the STREAMS device driver supports shared access.

Upon completing data transfer, a user process closes the STREAMS device. When the last user process using the Stream closes the device, the Stream is dismantled. Note that if the Stream is being shared, it remains open as long a process is still using it (has not closed the Stream).

Modules and Drivers

STREAMS software consists of protocol modules and drivers. The interfaces between modules, and between modules and drivers, are specified by the STREAMS standard to enable modules and drivers to be interchanged and combined. The following sections provide additional information.

- "Modules"
- "Drivers"

Modules

A STREAMS module performs intermediate processing of messages flowing on the Stream between the driver and the Stream head. A Stream can exist without a module, in which case the driver performs the necessary character and device I/O processing. STREAMS modules are added, or *pushed*, onto a Stream immediately below the Stream head. Modules can also be removed, or *popped*, from a Stream in the reverse order in which they were added.

Messages can contain protocol control information, status information, user data, or device data. Messages have a defined type, and are processed by STREAMS modules accordingly.

As previously stated, each module has a queue for messages traveling upstream and a queue for messages traveling downstream. Each message queue in a STREAMS module contains pointers to the following entities.

- **Messages** – These are dynamically attached to the queue on a linked list as they arrive at a module.
- **Procedures** – A "put" procedure to transfer messages and, optionally, a service procedure to process messages received by the module.
- **Data** – Module writers (programmers) can access and use data internal to the module.

STREAMS modules are extensible with other modules. Modules produced by independent vendors that adhere to the STREAMS standards can be combined to form a single protocol stack.

Drivers

A STREAMS driver is structurally similar to a STREAMS module. The call or service interfaces to a STREAMS driver are identical to the interfaces used for modules. However, the following differences exist between drivers and modules.

- A STREAMS driver must handle interrupts and signals from an external device; STREAMS modules do not interface directly with external devices.
- A STREAMS driver can have multiple modules pushed above it.
- A STREAMS driver is initialized when the Stream is opened and created, whereas modules are added (pushed) onto a Stream after it is created.

Like a module, a STREAMS driver can pass signals, error codes, and other information to user processes, using message types defined for that purpose.

A pseudo-device driver is used to enable multiplexing on a Stream. A *multiplexing driver* enables multiple upstream Streams to be linked with a single downstream Stream. For example, a windowing terminal facility might use a separate Stream for each window, with the different upstream Streams multiplexed into a single downstream Stream to the actual terminal device. In this case, the multiplexing driver would be responsible for splitting and combining the data transmitted to and from each of the windows on the terminal.

Multiplexing drivers can also link a single upstream Stream with a single downstream Stream, or link multiple upstream Streams with multiple downstream Streams. Multiple downstream Streams that connect to different device drivers can provide flexibility for networking by providing STREAMS applications with access to multiple lower-layer communications protocols.

Service Interfaces

STREAMS defines a service interface that consists of a specified set of messages and rules for passing messages. STREAMS modules that interface with user processes, other modules, or with drivers must adhere to the service interface rules.

Service interface standards enable application programs to be written for STREAMS independently of underlying protocols and physical communications media. High-level protocols, such as those used in networking architectures, can be developed independently of the lower-layer protocols and drivers.

The STREAMS-related system calls that are used by user applications are shown in Table 1-1, along with the description of their function.

Table 1-1. STREAMS-Related System Calls

STREAMS/C Call	Function
`open`	Open a Stream
`close`	Close a Stream
`read`	Read data from a Stream
`write`	Write data to a Stream
`fcntl`	Control a Stream
`ioctl`	Control a Stream
`getmsg/getpmsg`	Receive a message from downstream
`putmsg/putpmsg`	Send a message downstream
`poll`	Notify the caller when selected events occur on a Stream

The service user, service provider, and the service interface itself are the three components used in a STREAMS service interface, as shown in Figure 1-2.

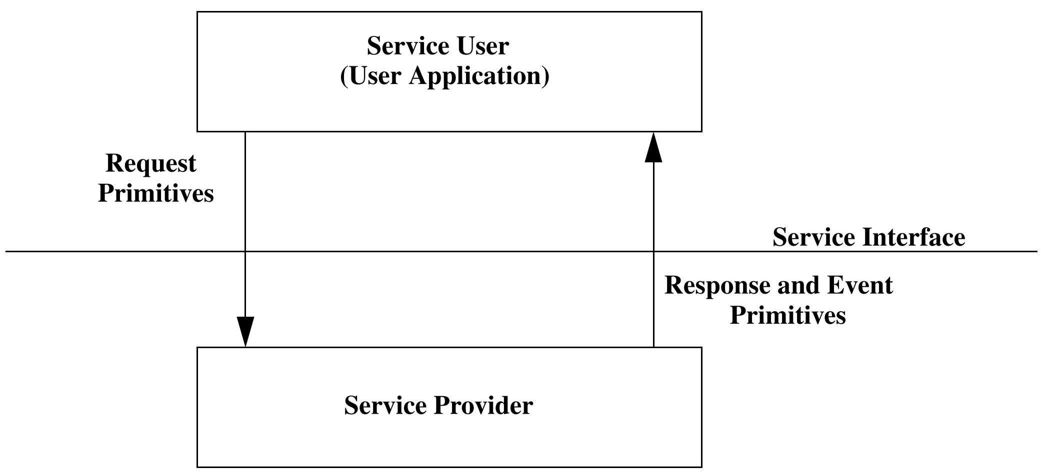

Figure 1-2. A STREAMS Service Interface

In STREAMS, each service interface primitive consists of a control part and a data part. The control portion contains information on the type of primitive and associated parameters, while the data portion contains user data. The service provider can be the Stream head, a STREAMS module, or a STREAMS driver.

Overview of OpenVOS STREAMS

OpenVOS STREAMS is an implementation of AT&T UNIX System V.4 STREAMS. OpenVOS STREAMS software executes in the OpenVOS operating system kernel and provides applications with a standard interface to STREAMS devices.

OpenVOS STREAMS provides the following enhancements to the standard AT&T STREAMS, without changing the standard interfaces to STREAMS modules or applications.

- Supports fully asynchronous I/O.
- Operates in a fully preemptive kernel.
- Supports dynamic loading of STREAMS modules.
- Supports a fault-tolerant, multiprocessor architecture.
- Supports both standard C functions and OpenVOS subroutines.

OpenVOS STREAMS enables the use of standard STREAMS protocol modules on OpenVOS systems. This allows OpenVOS applications to make use of STREAMS modules developed by independent developers for UNIX systems. Figure 1-3 shows the logical structure of the OpenVOS STREAMS software architecture.

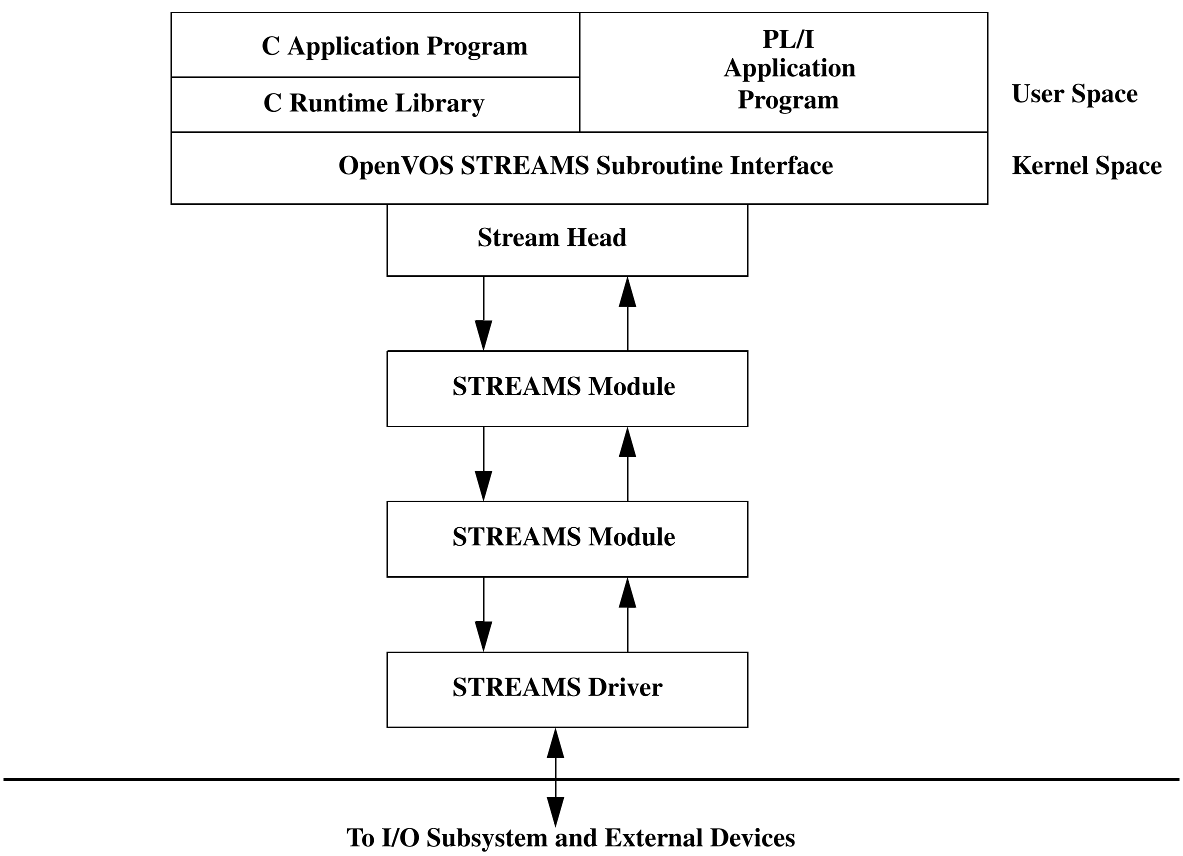

Figure 1-3. Logical Structure of OpenVOS STREAMS Software Architecture

Programming with OpenVOS STREAMS

Applications for OpenVOS STREAMS can be written in OpenVOS C or OpenVOS PL/I. The application programming interface for OpenVOS STREAMS consists of OpenVOS subroutines the application calls in order to create a Stream (open a STREAMS device), send and receive data on an opened Stream, and close a Stream. In addition, various control operations can be performed on an opened Stream, such as pushing modules on the Stream, sending control messages to modules, setting the blocking/nonblocking I/O mode, and requesting Stream-related operations.

Applications written in OpenVOS C can also use C language function calls to interface with OpenVOS STREAMS. Both the OpenVOS subroutines and the OpenVOS C language support for OpenVOS STREAMS are described further below.

OpenVOS Subroutine Calls

In OpenVOS, all communication between user applications and the operating system is via the OpenVOS subroutine call interface. The subroutines that are used for supporting traditional OpenVOS devices are also used for STREAMS devices. However, additional OpenVOS subroutines have been provided specifically for communication with STREAMS devices.

Table 1-2 lists the UNIX calls most often used with STREAMS on UNIX systems, and shows the corresponding OpenVOS subroutine.

Table 1-2. OpenVOS and UNIX STREAMS System Calls

STREAMS/UNIX Call	OpenVOS Equivalent Subroutine
`open`	`s$streams_open`
`close`	`s$streams_close`
`read`	`s$read_raw`
`write`	`s$write_raw`
`ioctl`	`s$ioctl`
`putmsg/putpmsg`	`s$putmsg/s$putpmsg`
`getmsg/getpmsg`	`s$getmsg/s$getpmsg`
`poll`	`s$poll`

Refer to Chapter 3 for detailed information about using these OpenVOS subroutines for OpenVOS STREAMS. For information on the `s$poll` subroutine, see the OpenVOS Subroutines manuals.

OpenVOS Standard C Language Support

OpenVOS also provides the OpenVOS Standard C (that is, the ANSI C-compliant) and OpenVOS POSIX.1 implementations of the C languages, which includes C language support for STREAMS. The C language support for STREAMS simplifies the porting of UNIX or C language STREAMS applications to the Stratus OpenVOS environment. Applications written in OpenVOS C can use either the OpenVOS subroutine or C language interface for OpenVOS STREAMS.

The OpenVOS Standard C language functions that support OpenVOS STREAMS are listed below.

`close`	`open`	`select`
`fcntl`	`poll`	`select_with_events`
`getmsg`	`putmsg`	`signal`
`getpmsg`	`putpmsg`	`write`
`ioctl`	`read`	`writev`
	`readv`	

Refer to Chapter 3 for detailed information about using these OpenVOS Standard C function calls with OpenVOS STREAMS. For information on the `s$poll` subroutine, see the *OpenVOS Standard C Reference Manual* (R363). For information on features of OpenVOS POSIX.1, see *OpenVOS POSIX.1 Reference Guide* (R502).

OpenVOS STREAMS Modules, Drivers, and Devices

OpenVOS STREAMS modules are provided with software products that are based on OpenVOS STREAMS. Users of OpenVOS STREAMS only design and write STREAMS applications, not modules. If OpenVOS STREAMS modules are used, they are provided with the associated STREAMS-based software product.

OpenVOS STREAMS drivers are also provided with STREAMS-based software products.

An OpenVOS STREAMS device is a device driver that resides in the OpenVOS kernel. An OpenVOS STREAMS driver is provided for use with each type of STREAMS device that is supported. For each type of STREAMS-based protocol firmware, a specific device driver is provided.

OpenVOS STREAMS Logging Facility

OpenVOS STREAMS provides a facility to log module error and trace messages. This facility is implemented using a special STREAMS device called the log device. Modules and drivers call the routine `strlog` to write error and trace information to the log device. A user application must be written to retrieve logged messages, using the `strace` routine. Refer to the AT&T manual *UNIX System V Release 4 Programmers Guide: STREAMS* for further information on using `strace` to retrieve logging messages.

Requests of the Analyze System Subsystem

The following `analyze_system` requests are available to obtain internal state information about OpenVOS STREAMS.

- `dump_stream`
- `scan_streams_msgs`
- `search_streams`

For information on these requests, see the *OpenVOS System Analysis Manual* (R073); however, these requests should be used only at the request of support personnel to assist with problem determination at a user site.

Chapter 2:
OpenVOS STREAMS Programming Support

This chapter contains the following sections.

- "Elements of OpenVOS STREAMS Programming Support"
- "Programming Considerations"

Elements of OpenVOS STREAMS Programming Support

The following sections describe elements of OpenVOS STREAMS programming support.

- "OpenVOS I/O Subroutines"
- "OpenVOS Standard C Language Functions"
- "Polling Support"

OpenVOS I/O Subroutines

Table 2-1 briefly describes some OpenVOS I/O subroutines used with OpenVOS STREAMS.

Table 2-1. OpenVOS Subroutines Used with OpenVOS STREAMS

Subroutine	**Function**
`s$streams_open`	Attaches an OpenVOS port to a STREAMS device and opens the device.
`s$streams_close`	Closes a STREAMS device and detaches the OpenVOS port from the device.
`s$read_raw`	Reads a data message from a STREAMS device.
`s$write_raw`	Writes a data message to a STREAMS device.
`s$ioctl`	Performs a control operation on a Stream.
`s$putmsg` `s$putpmsg`	Writes a data and/or control message to a Stream.
`s$getmsg` `s$getpmsg`	Reads a data and/or control message from a Stream.
`s$poll`	Polls one or more STREAMS devices for an OpenVOS or STREAMS event occurrence. Also, polls files, pipes, window terminals, and STREAMS devices on remote modules.

Refer to Chapter 3 for a detailed description of each of these OpenVOS subroutines. For information on the `s$poll` subroutine, see the OpenVOS Subroutines manuals.

OpenVOS Standard C Language Functions

OpenVOS Standard C provides support for the native UNIX C functions that are used by STREAMS applications. Note that OpenVOS Standard C does not support all UNIX C functions, but it does support a subset of those functions commonly associated with STREAMS, as shown in Table 2-2.

Table 2-2. OpenVOS Standard C Language Functions for OpenVOS STREAMS

OpenVOS C Function	Description
`close`	Closes a STREAMS device.
`fcntl`	Performs control operations, such as placing a STREAMS device in blocking or non-blocking mode.
`getmsg/getpmsg`	Reads a data and/or control message from a Stream.
`ioctl`	Performs a control operation on a Stream.
`open`	Opens a STREAMS device.
`poll`	Polls STREAMS device(s) for an event occurrence.
`putmsg/putpmsg`	Writes a data and/or control message to a Stream.
`read`	Reads a data message from a STREAMS device.
`readv`	Reads a data message from a STREAMS device. The input buffer can be broken into pieces.
`select`	Indicates which file descriptors can be read or written.
`select_with_events`	Indicates which file descriptors can be read or written and which operating-system events have been notified.
`signal`	Catches a `SIGPOLL` signal; used in conjunction with the `I_SETSIG` `ioctl` command.
`write`	Writes a data message to a STREAMS device.
`writev`	Writes a data message to a STREAMS device. The output buffer can be broken into pieces.

Refer to Chapter 3 for further information on using these OpenVOS C language functions for OpenVOS STREAMS. For information on the `poll` function, see the *OpenVOS Standard C Reference Manual* (R363).

Polling Support

OpenVOS STREAMS allows applications to poll a STREAM for the occurrence of an event, using either the C language `poll` function or the `s$poll` subroutine. In general, polling allows an application to multiplex input and output over multiple Streams.

The `poll` function is used to poll STREAMS-related events. The `poll` function returns an indication of the file descriptors (Streams) on which messages can be sent or received, or on which certain events have occurred. Note that only STREAMS-related events (not OpenVOS events) can be polled using the `poll` function.

The `s$poll` subroutine supports polling both STREAMS events and OpenVOS events. It uses OpenVOS port IDs to check for STREAMS events. It uses event IDs to indicate which OpenVOS events have occurred, in a manner similar to the OpenVOS subroutine `s$wait_event`. In addition, the `s$poll` subroutine supports polling of files, pipes, window terminals, and STREAMS devices on remote modules.

For further information, refer to Chapter 3 as well as to the *OpenVOS Standard C Reference Manual* (R363) for a description of the OpenVOS C `poll` function and to the OpenVOS Subroutines manuals for a description of the `s$poll` subroutine.

Programming Considerations

The following sections describe considerations for developing application programs for OpenVOS STREAMS:

- "General Structure of an Application Program"
- "Using Blocking Mode and Nonblocking Mode"
- "Flow Control and Buffering"
- "Tasking"
- "Compiling, Binding, and Debugging"
- "Using Include Files"
- "Device Configuration Requirements"

General Structure of an Application Program

The applications written for OpenVOS STREAMS can vary considerably, depending upon the communications protocol and upon the extent to which the application is performing protocol-related processing. An application must, however, issue the following OpenVOS STREAMS subroutine calls or equivalent C language function calls.

1. An application must call `s$streams_open` to establish a port connection with a STREAMS device and to open the device. The `s$streams_open` subroutine returns the OpenVOS port ID for the STREAMS device, which is used in subsequent subroutine calls for the opened Stream.

   The C runtime function `open` performs the equivalent operation, returning the file descriptor, which is used in subsequent C runtime function calls for the opened Stream. An application performs input/output on the Stream using `s$read_raw` and `s$write_raw`. Depending upon the protocol, the application may also have to send and receive protocol-related control information as well as data, using the subroutines s$putmsg (or s$putpmsg) and s$getmsg (or s$getpmsg).

The C runtime functions read, write, putmsg, putpmsg, getmsg, and getpmsg perform the equivalent operations.

2. In certain situations, an application may need to perform control operations on the open Stream, using the subroutines s$ioctl, s$putmsg, s$putpmsg, s$getmsg, or s$getpmsg.

   The C runtime functions ioctl, fnctl, putmsg, putpmsg, getmsg, and getpmsg perform the equivalent operations.

3. An application terminates communication with a STREAMS device by calling s$stream_close.

   The C runtime function close performs the equivalent operation.

The preceding OpenVOS STREAM subroutines and C functions are discussed further in Chapter 3.

Using Blocking Mode and Nonblocking Mode

An application can operate in either blocking mode or nonblocking mode. In *blocking mode* (the default), a call to an operating system subroutine does not return until the requested operation has completed or until a timeout occurs (if an I/O time limit has been established via the s$set_io_time_limit subroutine). If necessary, the subroutine blocks the application until it performs the operation. For example, s$read_raw does not return to the application until input is available or a timeout occurs.

In *nonblocking mode*, a call to an operating system subroutine does not block the application. If the requested operation cannot be completed immediately, the subroutine returns the error code EAGAIN. For example, s$read_raw returns this code when there is no data to receive. The application must retry the subroutine call later and, until then, can perform other work.

Unlike other OpenVOS applications, OpenVOS STREAMS applications **do not** use the OpenVOS subroutines s$set_wait_mode and s$set_no_wait_mode to enable blocking mode and nonblocking mode. Both of these subroutines, if called for a STREAMS device, return the error code e$invalid_io_operation (1040).

In OpenVOS STREAMS, blocking and nonblocking mode are enabled by using the following calls.

- s$streams_open
- s$ioctl/ioctl
- fcntl

For additional information, see the sections "Steps to Follow When Using Nonblocking Mode" and "Guidelines for Using Blocking and Nonblocking Modes".

Steps to Follow When Using Nonblocking Mode

The method of using nonblocking mode with STREAMS devices is as follows.

1. The application can call `s$streams_open` with `io_type` argument set to `STREAMS_ONDELAY` (see Chapter 3). This sets the variable `STREAMS_ONDELAY`, which places the application in nonblocking mode.

   Or,

   after the STREAMS device has been opened, `s$ioctl` or `ioctl` (see Chapter 3) can be called with the control command `I_SETDELAY` to switch between blocking mode and nonblocking mode.

   Or,

   a OpenVOS C application can call `fcntl` to set the file descriptor flags (see Chapter 3) after the STREAMS device has been opened to switch between blocking mode and nonblocking mode.

2. Read or write to the port until the error code `EAGAIN` (or `EAGAIN` for OpenVOS C applications) is returned.

3. Continue processing or call `s$poll` or the `poll` C runtime function. The `s$poll` subroutine returns an array of OpenVOS and/or STREAMS events that have occurred for the Streams. The `poll` function returns the STREAMS events, on a per-file descriptor basis, of the Streams for which an event has occurred. For further information, refer to the *OpenVOS Standard C Reference Manual* (R363) for a description of the OpenVOS C `poll` function and to the OpenVOS Subroutines manuals for a description of the `s$poll` subroutine.

Guidelines for Using Blocking and Nonblocking Modes

The choice of using blocking mode or nonblocking mode depends on the individual application. However, several general guidelines apply, as described below.

It is often convenient to open the STREAMS device in blocking mode and then switch to nonblocking mode for data transfer, since typically the application cannot perform any other processing until after the STREAMS device has been opened.

If application processing is driven by user input, blocking mode may be appropriate if no other processing is possible or necessary. If the application does not perform buffering, then blocking mode may be appropriate for write operations in order to prevent buffer overflows and loss of data by the receiver.

Multiplexing application programs may also require the use of nonblocking mode, regardless of the communications style. If multiplexing is achieved by the use of tasking, nonblocking mode must be used. If multiplexing is achieved through explicit programming, nonblocking mode is normally required. Tasking is discussed later in this chapter.

Flow Control and Buffering

STREAMS provides a mechanism for flow control. Each module in a Stream can use flow control to limit the number of bytes held in its read or write queues. The Stream head also uses a flow control mechanism to limit the number of bytes that can be received at the Stream head, and a STREAMS driver may also limit the number of bytes sent downstream from the Stream head.

High-priority messages are not subject to the flow control used for normal messages.

Communications protocol modules can also use end-to-end flow control between the local and remote end of the communications link. In this case, a transmit and receive window is used to indicate the number of bytes that can be sent or received.

Tasking

If an application must communicate with multiple STREAMS devices, tasking can be used to enhance resource sharing. When tasking is used with STREAMS devices, the tasking application **must** open the STREAMS devices in nonblocking mode. This differs from using tasking with other OpenVOS devices, where the main tasking application typically uses blocking (wait) mode and the subtasks spawned for each device use nonblocking (no-wait) mode. Refer to “Initializing a STREAMS Device Using `s$streams_open`” in “Initializing a STREAMS Device Using `s$streams_open`” in Chapter 3 for information on opening a STREAMS device in nonblocking mode.

When tasking is used for STREAMS devices, the application can use the polling support mechanism (as previously described in this chapter) to detect when data arrives for a device, or when a flow control condition has cleared.

Refer to the OpenVOS Transaction Processing Facility Manuals for information about using tasking in an application program.

Compiling, Binding, and Debugging

An OpenVOS STREAMS application must be compiled and bound to create an executable program. Refer to the appropriate OpenVOS programming language manual for information on compiling and binding an application for execution under OpenVOS.

If an application does not compile or execute correctly, the OpenVOS debugger (`debug` command) can be used to help diagnose the problem. For information on the `debug` command, refer to the *OpenVOS Commands Reference Manual* (R098).

Using Include Files

Applications written in OpenVOS C or OpenVOS PL/I can use include files that are supplied for use with OpenVOS STREAMS. These files include definitions for the various structures, subroutine options, and output values returned by the application interface to OpenVOS STREAMS.

Refer to “OpenVOS Include Files” in Chapter 3 for a list of the include files and a description of their contents.

Device Configuration Requirements

Before they can be used by applications, all STREAMS devices must be defined to OpenVOS in the device table configuration file, `devices.tin`. The device type value `streams` is used to define a STREAMS device in the `devices.tin` file. After STREAMS devices are defined in the device table, STREAMS drivers and modules can be added to the OpenVOS runtime system, using the `configure_comm_protocol` command. Refer to the manual *OpenVOS System Administration: Configuring a System* (R287) for general information on configuring devices.

After STREAMS devices and modules have been added to the system, an application can then open and perform I/O using the STREAMS device. An application opening a STREAMS device is actually opening a specific STREAMS protocol driver. After opening a STREAMS device, an application can, if necessary, push protocol-specific STREAMS modules below the Stream head and above the STREAMS driver.

Chapter 3: Application Program Interface

This chapter documents the application program interface to OpenVOS STREAMS. It contains the following sections.

Applications written in OpenVOS C can use either the OpenVOS subroutines or the OpenVOS Standard C language interface to OpenVOS STREAMS. Applications written in OpenVOS PL/I must use the OpenVOS subroutines as the interface to OpenVOS STREAMS.

OpenVOS Include Files

This chapter uses the constant names and structure declarations defined in the following include files, wherever appropriate. These include files are located in the `(master_disk)>system>include_library` directory.

- `stropts.h`—Contains all `s$ioctl` commands, command options, and control structure declarations.

- `poll.h`—Contains all STREAMS poll events used with the `s$poll` subroutine and the `poll` C function. This file also contains the `pollfd` and `strpoll` structure definitions.

- `types.h`—Contains data type definitions that are useful when porting STREAMS applications for use with OpenVOS STREAMS.

- `signal.h`—Contains signals, including `SIGPOLL`, which is used with the `s$ioctl` `I_SETSIG` command and the OpenVOS C `signal` function.

- `fcntl.h`—Contains all options that can be used when opening a STREAM using the `open` function. This file also contains the constant definitions for `F_GETFL` and `F_SETFL`, which are used by the `fcntl` C function.

STREAMS applications written in OpenVOS C can include these header files as part of the source program. OpenVOS PL/I applications can use the corresponding include files (which have a suffix `.incl.pl1` in place of the `.h` suffix), except for `fnctl.h` and `types.h`, which are used only with OpenVOS C. Note that for OpenVOS PL/I applications, the include file `errno.incl.pl1`, located in the `(master_disk)>system>include_library` directory, contains the STREAMS error code names and their corresponding integer values.

Table 3-1 lists each of the OpenVOS subroutines used with OpenVOS STREAMS and briefly describes their functions.

Table 3-1. Description of OpenVOS STREAMS Subroutines

Subroutine	Function
`s$streams_open`	Attaches and opens a STREAMS device.
`s$streams_close`	Closes and detaches a STREAMS device.
`s$read_raw`	Reads data from a STREAMS device.
`s$write_raw`	Writes data to a STREAMS device.
`s$putmsg/s$putpmsg`	Writes data and/or a control message to a Stream.
`s$getmsg/s$getpmsg`	Reads data and/or a control message from a Stream.
`s$ioctl`	Requests an I/O control operation on a Stream. Specific control operations (commands) are passed as parameters to the `s$ioctl` subroutine.
`s$poll`	Monitors I/O activity for multiple STREAMS devices.

Initializing a STREAMS Device Using `s$streams_open`

To use a STREAMS device, an application must first call the subroutine `s$streams_open` to open a specified STREAMS device and to obtain an OpenVOS port ID for the device. An OpenVOS *port ID* is an identifier, associated with an actual device or file, that is returned to the application. It is used in all subsequent subroutine calls to identify the STREAMS device.

The `s$streams_open` subroutine is declared as follows.

Usage

```
#include <system_io_constants.incl.c>

 short int          port_id;
 char_varying (256) path_name;
 short int          file_org;
 short int          max_len;
 short int          io_type;
 short int          lock_mode;
 short int          access_mode;
 char_varying (32)  index_name;
 short int          error_code;

 void s$streams_open ();

      s$streams_open (&port_id,
                      &path_name,
                      &file_org,
                      &max_len,
                      &io_type,
                      &lock_mode,
                      &access_mode,
                      &index_name,
                      &error_code);
```

Arguments

- `port_id` (output)
  Contains the port ID returned by `s$streams_open`. The returned `port_id` value is used in all subsequent OpenVOS subroutine calls for the opened Stream.

- `path_name` (input)
  The `path_name` parameter is the full path name of a STREAMS device. When opening a clone device, `s$streams_open` opens the next available clone device. The application can determine exactly which clone device was opened by using the `s$get_port_info` subroutine.

- `file_org` (input)
  This argument is not used and should be initialized to `0`.

- `max_len` (input)
  This argument is not used and should be initialized to `0`.

- `io_type` (input)
  For STREAMS devices, this argument can be set to `STREAMS_ONDELAY` to set nonblocking mode for the STREAMS device. For blocking mode, `io_type` should be set to `0`.

- `lock_mode` (input)
  This argument is not used and should be initialized to `0`.

- `access_mode` (input)
  This argument is not used and should be initialized to `0`.

▶ `index_name` (input)
This argument is not used.

▶ `error_code` (output)
A returned error code. A value of `0` indicates successful completion of the `s$streams_open` call. Possible error code values are listed below.

OpenVOS Error Code	`errno.h` **Equivalent**
`e$invalid_io_operation`	none
`e$device_not_found`	none
`ENOENT`	`ENOENT`
`ENOMEM`	`ENOMEM`
`e$clone_limit_exceeded`	none

Refer to the section "OpenVOS STREAMS Return Codes" at the end of this chapter for a description of each of these return codes.

Example

The example below opens a STREAMS device with the path name of `%sysA#dev.streams.01.01`, which is a nonclonable STREAMS device defined in the devices table. The device is opened for nonblocking mode by setting the `io_mode` parameter to `STREAMS_ONDELAY`. The parameters `file_org`, `max_len`, `lock_mode`, and `access_mode` (not used by `s$streams_open`) are initialized to `0`, and the `index_name` parameter, also unused, is set to the null string. If the call to `s$streams_open` returns a nonzero error code, an error processing routine is called.

```
path_name = "%sysA#dev.streams.01.01";
file_org = 0;
max_len = 0;
io_type = STREAMS_ONDELAY;
lock_mode = 0;
access_mode = 0;
index_name = "";

     s$streams_open(&port_id, &path_name, &file_org,
                    &max_len, &io_type, &lock_mode,
                    &access_mode, &index_name, &error_code);
     if (error_code)
          process_error(&error_code);
```

Termination Procedures Using `s$streams_close`

The `s$streams_close` subroutine performs termination procedures, which consist of closing the STREAMS device and detaching the OpenVOS port from the device. The `s$streams_close` subroutine is declared as follows.

Usage

```
short int port_id;
short int error_code;

void s$streams_close ();

     s$streams_close (&port_id,
                      &error_code);
```

Arguments

- `port_id` (input)
  Specifies the port ID returned by `s$streams_open`.

- `error_code` (output)
  A returned error code. A value of `0` indicates successful completion. Possible error code values are listed below.

OpenVOS Error Code	`errno.h` **Equivalent**
`e$invalid_io_operation`	none
`e$device_already_assigned`	none

Refer to the section "OpenVOS STREAMS Return Codes" at the end of this chapter for a description of each of these return codes.

Reading Data Using `s$read_raw`

The subroutine `s$read_raw` is used to read data from a STREAMS device.

When a STREAMS device is in nonblocking mode (i.e., when `STREAMS_ONDELAY` is set), a call to `s$read_raw` always returns immediately with any data present at the Stream head. Otherwise, `s$read_raw` returns immediately with `EAGAIN` in the `error_code` parameter. In this case, the `EAGAIN` return code is synonymous with `e$caller_must_wait`. The application **cannot** use the OpenVOS subroutine `s$wait_event` to wait for data. Instead, the application must use either the `s$poll` subroutine or the `I_SETSIG s$ioctl` control operation in order to be notified when data or a message is available.

For a STREAMS device in blocking mode, the call to `s$read_raw` blocks the application (waits) until data arrives.

Refer to Chapter 2 for more information on blocking mode and nonblocking mode.

Refer to the OpenVOS Subroutines manuals for a complete description of `s$poll` and of `s$read_raw` and its parameters.

If `s$read_raw` fails, the `error_code` parameter can contain the following error code values (a value of `0` indicates successful completion).

OpenVOS Error Code	`errno.h` **Equivalent**
`EAGAIN`	`EAGAIN`
`EBADMSG`	`EBADMSG`
`e$invalid_io_operation`	none

Refer to the section "OpenVOS STREAMS Return Codes" at the end of this chapter for a description of each of these return codes.

Writing Data Using s$write_raw

The subroutine s$write_raw is used to write data to a STREAMS device.

If the application is in blocking mode, the call to s$write_raw blocks until all data has been written. If the application is in nonblocking mode, EAGAIN is returned if the data could not be written immediately. Refer to Chapter 2 for more information on blocking mode and nonblocking mode.

Refer to the OpenVOS Subroutines manuals for a complete description of s$write_raw and its parameters.

If s$write_raw fails, the error_code parameter can contain the following error code values (a value of 0 indicates successful completion).

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
ERANGE	ERANGE
e$invalid_io_operation	none

Refer to the section "OpenVOS STREAMS Return Codes" at the end of this chapter for a description of each of these return codes.

Message Passing

The subroutines used to receive and send data and/or STREAMS control messages are `s$getmsg`, `s$getpmsg`, `s$putmsg`, and `s$putpmsg`, which the following sections describe.

- "Receiving Messages Using `s$getmsg/s$getpmsg`"
- "Sending Messages Using `s$putmsg/s$putpmsg`"

Receiving Messages Using `s$getmsg/s$getpmsg`

The subroutines `s$getmsg` and `s$getpmsg` are used to receive a STREAMS data message and/or control message from the Stream head. The subroutine `s$getpmsg` is almost identical to `s$getmsg`, but has an additional argument to specify the priority band at which to receive the message.

Unless otherwise specified in the following discussion, information presented on `s$getmsg` also applies to `s$getpmsg`.

Note that according to the STREAMS standard, an application that receives a message containing only control information can also receive either a zero-length data buffer or no data buffer (a null buffer pointer). OpenVOS STREAMS supports this feature, but in a restricted manner, as described below.

The declarations of both `s$getmsg` and `s$getpmsg` are shown below.

Usage

```
#include <stropts.h>

 short int port_id;
 long  int ctl_maxlen;
 long  int ctl_len;
 char      *ctl_bufp;
 long  int data_maxlen;
 long  int data_len;
 char      *data_bufp;
 long  int *bandp
 long  int *flagsp;
 long  int rval;
 short int error_code;

 void s$getmsg ();
 void s$getpmsg ();

      s$getmsg (&port_id,
                &ctl_maxlen,
                &ctl_len,
                ctl_bufp,
                &data_maxlen,
                &data_len,
                data_bufp,
                flagsp,
                &rval,
                &error_code);

      s$getpmsg (&port_id,
                 &ctl_maxlen,
                 &ctl_len,
                 ctl_bufp,
                 &data_maxlen,
                 &data_len,
                 data_bufp,
                 bandp,
                 flagsp,
                 &rval,
                 &error_code);
```

Arguments

- ▶ `port_id` (input)
  Specifies the port ID of a Stream that was previously opened by the application.

- ▶ `ctl_maxlen` (input)
  Specifies the size of the output character array `ctl_bufp[]` for the control portion of the message.

- ▶ `ctl_len` (output)
  Contains the actual number of bytes returned in `ctl_bufp`.

- ctl_bufp (output)
  Contains the control portion of the message. If ctl_len is set to 0 and ctl_bufp is a valid buffer pointer, a zero-length control buffer is returned. If ctl_len is 0 and ctl_bufp is set to OS_NULL_PTR, no control portion of the message is returned.

- data_maxlen (input)
  Specifies the size of the output character array data_bufp[] for the data portion of the message.

- data_len (output)
  Contains the actual number of bytes returned in data_bufp.

- data_bufp (output)
  Contains the data portion of the message. If data_len is set to 0 and data_bufp is a valid buffer pointer, a zero-length data buffer is returned. If data_len is 0 and data_bufp is set to OS_NULL_PTR, no data portion of the message is returned.

- bandp (input/output)
  For use with s$getpmsg only, specifies the priority band of the message to be received. The bandp parameter for s$getpmsg is used in conjunction with the flagsp parameter, as described below.

- flagsp (input/output)
  Specifies the priority band of the message to be received. It is used by s$getmsg and s$getpmsg differently, as described below.

- rval (output)
  Contains an indication of whether or not there are more messages still at the Stream head. A value of 0 indicates that a complete message has been returned. A value of MORECTL indicates that a control message is queued at the Stream head. A value of MOREDATA indicates that a data message is queued at the Stream head. A value of MORECTL | MOREDATA indicates that both message types are queued at the Stream head.

- error_code (output)
  Contains an indication of the outcome of the operation. A nonzero value indicates failure. Error codes are listed later in this section.

Explanation

The bandp parameter for s$getpmsg is used in conjunction with the flagsp parameter, as described below.

For s$getmsg, the flagsp parameter can be used to retrieve only high-priority messages by setting flagsp to RS_HIPRI. By default, s$getpmsg returns the first available message queued at the Stream head. If flagsp is set to 0, s$getmsg returns any message queued at the Stream head and, if a high-priority message is returned, sets flagsp to RS_HIPRI.

For s$getpmsg, the flagsp parameter points to a bit mask with the following mutually exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like s$getmsg, s$getpmsg returns the first available message queued at the Stream head. By setting flagsp to MSG_HIPRI and bandp to 0, the user can request to receive only a high-priority message, in which case, s$getpmsg will return the next available message only if it is of high priority.

To receive a message of any priority band, the user sets flagsp to MSG_BAND and sets bandp to a specific priority band. In this case, s$getpmsg will return the next available message only if its priority is equal to or greater than the priority band pointed to by bandp, or if it is a high-priority message.

To receive the first message queued at the Stream head, the user sets flagsp to MSG_ANY and sets bandp to 0. On return, flagsp is set to MSG_HIPRI and bandp to 0 if the message returned is a high-priority message. Otherwise, on return, flags is set to MSG_BAND and bandp is set to the priority band of the returned message.

The following return codes can be returned by s$getmsg and s$getpmsg.

OpenVOS Error Code	errno.h **Equivalent**
EINVAL	EINVAL
EAGAIN	EAGAIN
EBADMSG	EBADMSG
e$invalid_io_operation	none

Sending Messages Using s$putmsg/s$putpmsg

The subroutines s$putmsg and s$putpmsg send a STREAMS data or control message from application buffers downstream to a STREAMS device or module. The subroutine s$putpmsg is almost identical to s$putmsg, but has an additional argument to specify the priority band at which to send the message.

Unless otherwise specified in the following discussion, information presented on s$putmsg also applies to s$putpmsg.

STREAMS messages sent using s$putmsg can contain control information, data, or both. The application passes data and control information to s$putmsg using separate buffers for each. The ability to send control information along with data distinguishes s$putmsg from s$write_raw. The control information is typically directed to a specific module or driver in the Stream.

Note that according to the STREAMS standard, an application that is sending only control information can pass either a zero-length data buffer or no data buffer (a null buffer pointer). OpenVOS STREAMS supports this feature, but in a restricted manner, as described below.

The declarations of both s$putmsg and s$putpmsg are shown below.

Usage

```
#include <stropts.h>

short int port_id;
long  int ctl_len;
char      *ctl_bufp;
long  int data_len;
char      *data_bufp;
long  int *bandp;
long  int *flagsp;
short int error_code;

void s$putmsg ();
void s$putpmsg ();

     s$putmsg (&port_id,
               &ctl_len,
               ctl_bufp,
               &data_len,
               data_bufp,
               flagsp,
               &error_code);

     s$putpmsg (&port_id,
                &ctl_len,
                ctl_bufp,
                &data_len,
                data_bufp,
                bandp,
                flagsp,
                &error_code);
```

Arguments

- `port_id` (input)
  Specifies the port ID of the Stream that was previously opened by the application.

- `ctl_len` (input)
  Specifies the length of the array `ctl_bufp[]`. Note that `ctl_len` must be set to a value of `0` or greater; it cannot be set to `-1`.

- `ctl_bufp` (input)
  Contains the control portion of the message to be sent downstream. If `ctl_len` is set to `0` and `ctl_bufp` is a valid buffer pointer, a zero-length control buffer is sent. If `ctl_len` is `0` and `ctl_bufp` is set to `OS_NULL_PTR`, no control portion of the message is sent.

- `data_len` (input)
  Specifies the length of the array `data_bufp[]`. Note that `data_len` must be set to a value of `0` or greater; it cannot be set to `-1`.

- `data_bufp` (input)
  Contains the data portion of the message to be sent downstream. If `data_len` is set to `0` and `data_bufp` is a valid buffer pointer, a zero-length data buffer is sent. If

data_len is 0 and data_bufp is set to OS_NULL_PTR, no data portion of the message is sent.

- ▶ bandp (input)
  For use with s$putpmsg only, specifies the priority band of the message to be sent downstream. The bandp parameter for s$putpmsg is used in conjunction with the flagsp parameter, as described below.

- ▶ flagsp (input)
  Specifies the priority band of the message to be sent downstream. It is used by s$putmsg and s$putpmsg differently, as described in the “Explanation” section.

- ▶ error_code (output)
  Contains an indication of the outcome of the operation. A nonzero value indicates failure. Error codes are listed later in this section.

Explanation

The same blocking/nonblocking rules for s$write_raw apply to s$putmsg and s$putpmsg.

s$putmsg and s$putpmsg use the flagsp parameter with various flags, as follows:

> **Note**: On OpenVOS systems, the flags MSG_HIPRI and RS_HIPRI are interchangeable. However, in standard use, RS_HIPRI is for use with I_PEEK ioctl, and MSG_HIPRI is for use with getmsg, getpmsg, putmsg, and putpmsg. An OpenVOS application that uses MSG_HIPRI and RS_HIPRI interchangeably is not portable.

- For s$putmsg, the flagsp parameter sends priority messages to a STREAMS device. The flag 0 indicates a normal priority message, and the flag MSG_HIPRI indicates a high-priority message. See below for information on the flag s$PUTMSG_PARTIAL.

- For s$putpmsg, the flagsp parameter points to a bit mask with the mutually exclusive flags defined, MSG_HIPRI and MSG_BAND. If the integer pointed to by flagsp is set to 0, s$putpmsg fails and returns EINVAL.

  If a control part is defined and flagsp is set to MSG_HIPRI and bandp is set to 0, a high-priority message is sent. If flagsp is set to MSG_HIPRI and either no control part is defined or bandp is set to a nonzero value, s$putpmsg fails and returns EINVAL.

  If flagsp is set to MSG_BAND, then a message is sent at the priority band specified by bandp. If a control part and data part are not specified and flagsp is set to MSG_BAND, no message is sent and 0 is returned. (MSG_BAND is valid only for s$putpmsg.)

The s$PUTMSG_PARTIAL flag (in stropts.incl) can be passed to s$putmsg and s$putpmsg. This flag can be used to inform the driver that the application will be sending more data. How the driver responds to this flag is driver-specific. The driver is not required to transport the data until the application uses s$putmsg without the s$PUTMSG_PARTIAL flag. For example, the TCP driver defers transmitting the last segment of partial data and also adjusts the tcp protocol push bit to inform the peer to avoid waking up the receiving process for partial data.

The following error codes values can be returned by `s$putmsg` and `s$putpmsg`.

OpenVOS Error Code	`errno.h` Equivalent
`EINVAL`	`EINVAL`
`ERANGE`	`ERANGE`
`EAGAIN`	`EAGAIN`
`e$invalid_io_operation`	none

Control Operations Using `s$ioctl`

STREAMS supports a variety of control functions (commands) that can be used on an opened Stream. This section describes how the OpenVOS subroutine `s$ioctl` is used to perform these control operations on an opened Stream.

The declaration of `s$ioctl` is shown below.

Usage

```
#include <stropts.h>

short int port_id;
long  int opcode;
short int (short int)control_structure;
long  int rval;
short int error_code;

void s$ioctl ();

     s$ioctl (&port_id,
              &opcode,
              &control_structure,
              &rval
              &error_code)
```

Arguments

▶ `port_id` (input)
Specifies the port ID of a Stream that was previously opened by the application.

▶ `opcode` (input)
Specifies the `s$ioctl` control operation. Each control operation (`s$ioctl` command) shown in Table 3-2 is documented in the sections that follow.

▶ `control_structure` (input/output)
Contains the parameter(s) specific to the control operation (`opcode`) being used. The use of `control_structure` is documented for each control operation in the sections that follow.

- ▶ rval (output)
  Contains a return value or output parameter from the s$ioctl control operation, where applicable.

- ▶ error_code (output)
  Contains an indication of the outcome of the s$ioctl operation. A nonzero value indicates failure. Error codes are listed for each control operation in the sections that follow.

Explanation

Each s$ioctl control operation (command) shown in Table 3-2 is described in the following sections.

The s$ioctl commands, defined constants, structures, and return values are contained within the stropts.h include file.

After calling s$ioctl, application processing is suspended until the command request has been serviced, even if the application is in nonblocking mode.

Table 3-2. Control Operations for s$ioctl

Control Operation	Function
I_ATMARK	Checks if the current message on the Stream head read queue is "marked" by a downstream module.
I_CANPUT	Checks if a specified priority band is writable (not flow controlled).
I_CKBAND	Checks if a message of a specified priority band exists on the Stream head read queue.
I_FDINSERT	Allows an application to send a message to the Stream head that includes queue information in the message.
I_FIND	Checks if a specified module is present in a Stream.
I_FLUSH	Flushes all input or output queues in a Stream.
I_FLUSHBAND	Flushes all messages of a specified priority band.
I_GETBAND	Returns the priority band of the first message on the Stream head read queue.
I_GETCLTIME	Returns the delay close time for a Stream.
s$I_GET_MAX_CTL	Returns the maximum length that the s$putmsg subroutine allows for the control buffer.
s$I_GET_MAX_DATA	Returns the maximum length that the s$putmsg subroutine allows for the data buffer.
I_GETSIG	Returns the signals on which the application is waiting.

Table 3-2. Control Operations for `s$ioctl` (Continued)

Control Operation	Function
I_GRDOPT	Returns the read options previously set by I_SRDOPT.
I_LINK	Links a STREAMS driver above another STREAMS driver.
I_LIST	Returns a list of all module names on a Stream.
I_NREAD	Returns the number of bytes in the first message on the Stream head read queue.
I_PEEK	Returns information about the first message on the Stream head read queue without removing the message from the queue.
I_PLINK	Links a STREAMS driver above another STREAMS driver, creating a persistent link.
I_POP	Removes the topmost STREAMS module from a Stream.
I_PUNLINK	Unlinks two STREAMS drivers previously linked by I_PLINK.
I_PUSH	Pushes a STREAMS module onto a Stream.
I_RECVFD	Receives a port ID passed by another application via I_SENDFD.
I_SENDFD	Passes a port ID to another application.
I_SETCLTIME	Sets the closing time delay for a Stream.
I_SETDELAY	Places a STREAMS device in blocking mode or nonblocking mode.
I_SETSIG	Notifies the Stream head of events for which the application will receive SIGPOLL signals.
I_SRDOPT	Sets the read options of a Stream.
I_STR	Enables an application to exchange internal I/O commands and messages with downstream modules.
I_UNLINK	Unlinks two STREAMS drivers previously linked via I_LINK.

I_ATMARK

Purpose

The I_ATMARK command checks if the current message on the Stream head read queue is "marked" by some module downstream.

Explanation

The control_structure for I_ATMARK is a four-byte integer declared as follows.

```
long int options;
```

The options argument determines how the checking is to be done when there may be multiple marked messages on the Stream head read queue. It can be set to one of the following values.

Value	Meaning
ANYMARK	Check if the message is marked.
LASTMARK	Check if the message is the last one marked on the queue.

The range of rval output values and their meanings are as follows.

Value	Meaning
0	The specified mark condition is false.
1	The specified mark condition is true.
-1	An error has occurred.

Example

The example below checks if the current message on the Stream head read queue is marked by a downstream module.

```
long int options;
options = ANYMARK;
opcode = I_ATMARK;

s$ioctl (&port_id, &opcode, &options, &rval, &error_code);
```

The `I_ATMARK` command can return the following error codes.

OpenVOS Error Code	`errno.h` **Equivalent**
`EINVAL`	`EINVAL`
`e$invalid_io_operation`	none

I_CANPUT

Purpose

The I_CANPUT command checks if a specified priority band is writable (not flow controlled).

Explanation

The control_structure for I_CANPUT is declared as follows.

```
long int  priority;
```

The range of rval output values and their meanings are as follows.

Value	Meaning
0	The specified priority band is flow controlled.
1	The specified priority band is writable.
-1	An error has occurred.

Example

The example below checks if priority band 2 is writable.

```
long int priority;
priority = 2;
opcode = I_CANPUT;

s$ioctl (&port_id, &opcode, &priority, &rval, &error_code);
```

The I_CANPUT command can return the following error code.

OpenVOS Error Code	errno.h Equivalent
EINVAL	EINVAL

I_CKBAND

Purpose

The `I_CKBAND` command checks if the message of a given priority band exists on the Stream head read queue.

Explanation

The `control_structure` for `I_CKBAND` is declared as follows.

```
long int  priority;
```

The range of `rval` values and their meanings are as follows.

Value	Meaning
`1`	The Stream head read queue contains a message of the specified priority.
`0`	No message of the specified priority exists on the Stream head read queue.
`-1`	An error has occurred.

Example

The example below checks if a priority 2 message exists on the Stream head read queue.

```
long int priority;
priority = 2;
opcode = I_CKBAND;

s$ioctl (&port_id, &opcode, &priority, &rval, &error_code);
```

The `I_CKBAND` command can return the following error code.

OpenVOS Error Code	`errno.h` Equivalent
`EINVAL`	`EINVAL`

I_FDINSERT

Purpose

The I_FDINSERT command is used with STREAMS multiplexers. It allows an application to send a message to the Stream head and have OpenVOS STREAMS include queue information in the message. This message is then sent downstream. The queue information included is for the file descriptor specified as part of the I_FDINSERT command.

Explanation

The control_structure for I_FDINSERT is shown below.

```
struct strfdinsert
{
     struct  strbuf  ctlbuf;
     struct  strbuf  databuf;
     long            flags;
     int             fildes;
     int             offset;
};
```

The strbuf structure is declared as follows.

```
struct strbuf
{
     long int maxlen;
     long int len;
     char     *buf;
};
```

The application is responsible for supplying the message contained in the strbuf structures ctlbuf and databuf. Note that these structures are the same as those used for s$putmsg.

The flags field specifies whether or not the message is a priority message. If flags is set to RS_HIPRI, then it is a priority message; if flags is set to 0, then it is a nonpriority message.

The fildes parameter specifies the port ID (or file descriptor, if the OpenVOS C ioctl call is used) of the target module to receive the message.

The offset parameter must be set by the application to a word-aligned value that specifies the number of bytes from the beginning of the control buffer (ctlbuf) where the Stream head will store a pointer to the fildes Stream's driver read queue structure. How STREAMS modules (or more likely, STREAMS drivers) use this information is specific to the module (or driver). An application must know how the information will be used before using I_FDINSERT.

Example

The example below shows programming statements for using the I_FDINSERT command.

```
strfdinsert.ctlbuf.maxlen = max_ctl_len;
strfdinsert.ctlbuf.len    = ctl_len + (ctl_len % 2) +
                            sizeof(long *);
strfdinsert.ctlbuf.buf    = tmp;  /* control buffer */
strfdinsert.databuf.maxlen= max_data_len;
strfdinsert.databuf.len   = -1;   /* No data portion */
strfdinsert.databuf.buf   = writep;
strfdinsert.offset        = ctl_len + (ctl_len % 2);
strfdinsert.flags         = 0; /* nonpriority message */
strfdinsert.fildes        = read_fd;/* file id of another Stream */
opcode = I_FDINSERT;

s$ioctl (&write_fd, &opcode, &strfdinsert, &rval, &error_code);
```

The I_FDINSERT command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
EFAULT	EFAULT
EINVAL	EINVAL
ENXIO	ENXIO
ERANGE	ERANGE
e$invalid_io_operation	none

I_FIND

Purpose

The I_FIND command determines if a specified STREAMS module is present in a Stream.

Explanation

The control_structure for I_FIND is declared as follows.

```
char name [FMNAMESZ];
```

If an error_code of 0 is returned, the module exists in the Stream referenced by port_id.

Note that in the s$ioctl call, name should be passed by value (as name), and not by reference (as &name).

Example

The example below determines if the STREAMS module x25 is present in the Stream.

```
strcpy (name, "x25");
opcode = I_FIND;

s$ioctl (&port_id, &opcode, name, &rval, &error_code);
```

The I_FIND command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EFAULT	EFAULT
EINVAL	EINVAL
e$invalid_io_operation	none

I_FLUSH

Purpose

The `I_FLUSH` command flushes all input or output queues in a Stream.

Explanation

The `control_structure` for `I_FLUSH` is a four-byte integer declared as follows.

```
long int flush_options;
```

The range of `flush_options` values and their meanings are as follows.

Value	Meaning
`FLUSHR`	Flush read queues.
`FLUSHW`	Flush write queues.
`FLUSHRW`	Flush read and write queues.

Example

The example below flushes all read and write queues in a Stream.

```
long int flush_options;
flush_options = FLUSHRW
opcode = I_FLUSH;

s$ioctl (&port_id, &opcode, &flush_options, &rval, &error_code);
```

The `I_FLUSH` command can return the following error codes.

OpenVOS Error Code	`errno.h` Equivalent
`EAGAIN`	`EAGAIN`
`EINVAL`	`EINVAL`
`ENXIO`	`ENXIO`
`e$invalid_io_operation`	none

I_FLUSHBAND

Purpose

The I_FLUSHBAND command flushes messages of a particular priority band.

Explanation

The control_structure for I_FLUSHBAND is declared as follows.

```
struct bandinfo
{
     unsigned char bi_pri;
     char          bi_pad [3];
     long int      bi_flag;
};
```

The element bi_pri is the priority band of messages to be flushed, specified as an unsigned character.

The element bi_flag can be set to one of the following values.

Value	**Meaning**
FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush both read and write queues.

Example

The example below flushes all priority 2 messages from a Stream.

```
short int priority;
struct bandinfo binfo;
priority = 2;
binfo.bi_pri = (unsigned char) priority;
binfo.bi_flag = FLUSHRW;
opcode = I_FLUSHBAND;

s$ioctl (&port_id, &opcode, &binfo, &rval, &error_code);
```

The I_FLUSHBAND command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
EINVAL	EINVAL
ENXIO	ENXIO
e$invalid_io_operation	none

I_GETBAND

Purpose

The `I_GETBAND` command obtains the priority band of the first message on the Stream head read queue.

Explanation

The `control_structure` for `I_GETBAND` is declared as follows.

```
long int priority;
```

Example

The example below shows programming statements for using the `I_GETBAND` command.

```
long int priority;
opcode = I_GETBAND;

s$ioctl (&port_id, &opcode, &priority, &rval, &error_code);
```

The `I_GETBAND` command can return the following error code.

OpenVOS Error Code	`errno.h` Equivalent
ENODATA	ENODATA

I_GETCLTIME

Purpose

The I_GETCLTIME command obtains the delay close time for a Stream.

Explanation

The control_structure for I_GETCLTIME is a four-byte integer declared as follows.

```
long int options;
```

Example

The example below shows programming statements for using the I_GETCLTIME command.

```
long int options;
opcode = I_GETCLTIME;

s$ioctl (&port_id, &opcode, &options, &rval, &error_code);
```

s$I_GET_MAX_CTL

Purpose

The `s$I_GET_MAX_CTL` command obtains the maximum length that the `s$send_message` subroutine allows for the control buffer.

Explanation

The `control_structure` for `s$I_GET_MAX_CTL` is declared as follows.

```
size_t s;
r = ioctl(fd, s$I_GET_MAX_CTL, &s);
```

Example

The example below shows programming statements for using the `s$I_GET_MAX_CTL` command.

```
short int port_id;
long int opcode = s$I_GET_MAX_CTL;
size_t max_ctl_buf;
long int rval;
short int error_code;

s$ioctl (&port_id, &opcode, &max_ctl_buf, &rval, &error_code);
```

If the `s$I_GET_MAX_CTL` command fails, assume 1024 as the maximum control buffer size.

The `s$I_GET_MAX_CTL` command can return the following error code.

OpenVOS Error Code	`errno.h` **Equivalent**
`EINVAL`	`EINVAL`

s$I_GET_MAX_DATA

Purpose

The `s$I_GET_MAX_DATA` command obtains the maximum length that the `s$send_message` subroutine allows for the data buffer.

Explanation

The `control_structure` for `s$I_GET_MAX_DATA` is declared as follows.

```
size_t s;
r = ioctl(fd, s$I_GET_MAX_DATA, &s);
```

Example

The example below shows programming statements for using the `s$I_GET_MAX_DATA` command.

```
short int port_id;
long int opcode = s$I_GET_MAX_DATA;
size_t max_data_buf;
long int rval;
short int error_code;

s$ioctl (&port_id, &opcode, &max_data_buf, &rval, &error_code);
```

If the s$I_GET_MAX_DATA command fails, assume `0x6E00` (or less) as the maximum data buffer size.

The s$I_GET_MAX_DATA command can return the following error code.

OpenVOS Error Code	`errno.h` **Equivalent**
`EINVAL`	`EINVAL`

I_GETSIG

Purpose

The I_GETSIG command returns the signals the application is waiting on.

Explanation

The control_structure for I_GETSIG is a four-byte integer declared as follows.

```
long int signal_flags;
```

On return, the signal_flags argument contains the signals the application is waiting on. The signal_flags argument should be initialized to 0.

Example

The example below shows programming statements for using the I_GETSIG command.

```
long  signal_flags;
opcode = I_GETSIG;
signal_flags = 0;

s$ioctl (&port_id, &opcode, &signal_flags, &rval, &error_code);
```

The I_GETSIG command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EFAULT	EFAULT
EINVAL	EINVAL
e$invalid_io_operation	none

I_GRDOPT

Purpose

The I_GRDOPT command retrieves the read options previously set by the use of the I_SRDOPT command.

Explanation

The control_structure for I_GRDOPT is a four-byte integer declared as follows.

```
long int options;
```

Note that the options parameter is used for output only. On return, it can be set to one of the following values.

Value	**Meaning**
RNORM	Byte-stream mode
RMSGD	Message discard mode.
RMSGN	Message nondiscard mode

Example

The example below shows programming statements for using the I_GRDOPT command.

```
long int options;
opcode = I_GRDOPT;

s$ioctl (&port_id, &opcode, &options, &rval, &error_code);
```

The I_GRDOPT command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EFAULT	EFAULT
e$invalid_io_operation	none

I_LINK

Purpose

The I_LINK command links one STREAMS driver above another. Typically, this command links a multiplexing driver above another STREAMS driver. The rval output argument returns the multiplexer ID number.

Explanation

The control_structure for I_LINK is the port ID of the Stream to link.

Example

The example below shows programming statements for using the I_LINK command. In this example, the driver associated with portid1 is linked on top of the driver associated with portid2.

```
opcode = I_LINK;

s$ioctl (&portid1, &opcode, &portid2, &rval, &error_code);
```

The I_LINK command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
ENXIO	ENXIO
ETIME	ETIME
EAGAIN	EAGAIN
EBADF	EBADF
EINVAL	EINVAL
e$invalid_io_operation	none

I_LIST

Purpose

The I_LIST command returns a list of all the module names on the Stream, up to and including the topmost driver name. If called with the control_structure argument str_list set to NULL, it returns (in rval) the number of modules, including the driver, that are on the Stream associated with port_id. This allows the application to allocate sufficient space for the module names on a subsequent call.

Explanation

The control_structure for I_LIST is declared as follows and is defined in the include file stropts.incl.c (stropts.h).

```
struct str_list
{
     long int          sl_nmods;
     struct str_mlist *sl_modlist
};
```

The str_mlist structure is declared as follows.

```
struct str_mlist
{
     char l_name[FMNAMESZ + 1];
};
```

The sl_nmods member indicates the number of entries the application has allocated in the array and on return; sl_modlist contains the list of module names. The return value rval contains the number of entries that have been filled in.

Example

The example below shows programming statements for using the I_LIST command.

```
char   list_space[sizeof(int) + sizeof (void *) +
                      20 * (FMNAMESZ)];
struct str_list *  listp;
listp = (struct str_list *)&list_space;
listp->sl_nmods = 20;
listp->sl_modlist = (struct str_mlist *)&list_space +
                      sizeof (int) + sizeof (void *);
opcode = I_LIST;

s$ioctl (&port_id, &opcode, listp, &rval, &error_code);
```

The I_LIST command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
EINVAL	EINVAL

I_NREAD

Purpose

The I_NREAD command returns a count of the number of data bytes in the first message on the Stream head read queue. Only the number of data bytes is returned, and no indication is returned of how many control bytes may exist in the first message.

Explanation

The control_structure for I_NREAD is a two-byte integer declared as follows.

```
int nread;
```

Upon return, the nread argument contains the data byte count of the first message on the Stream head read queue.

Example

The example below shows programming statements for using the I_NREAD command.

```
int nread;
opcode = I_NREAD;

s$ioctl (&port_id, &opcode, &nread, &rval, &error_code);
```

The I_NREAD command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EFAULT	EFAULT
e$invalid_io_operation	none

I_PEEK

Purpose

The I_PEEK command retrieves information about the first message on the Stream head read queue without removing the message from the queue.

Explanation

The control_structure for I_PEEK is shown below.

```
struct strpeek
{
     struct strbuf  ctlbuf;
     struct strbuf  databuf;
     long           flags;
};
```

The strbuf structure is declared as follows.

```
struct strbuf
{
     int    maxlen;
     int    len;
     char   *buf;
};
```

Example

The example below shows programming statements for using the I_PEEK command.

```
strpeek.ctlbuf.maxlen = MAX_BUF_SIZE;
strpeek.databuf.maxlen = MAX_BUF_SIZE;
strpeek.ctlbuf.buf = &control_buffer[0];
strpeek.databuf.buf = &data_buffer[0];
opcode = I_PEEK;

s$ioctl (&port_id, &opcode, &peek, &rval, &error_code);
```

The I_PEEK command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EFAULT	EFAULT
e$invalid_io_operation	none

I_PLINK

Purpose

The I_PLINK command links one STREAMS driver above another. Typically, this command links a multiplexing driver above another STREAMS driver. The rval output argument returns the multiplexer ID number if successful, or a value of -1 if unsuccessful.

The difference between the I_PLINK command and the I_LINK command is that I_PLINK creates a persistent link below the multiplexing driver, which can exist even if the upper Stream to the multiplexing driver is closed.

Explanation

The control_structure for I_PLINK is the port ID of the Stream to link.

Example

The example below links the driver associated with portid1 on top of the driver associated with portid2.

```
opcode = I_PLINK;

s$ioctl (&portid1, &opcode, &portid2, &rval, &error_code);
```

The I_PLINK command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
ENXIO	ENXIO
ETIME	ETIME
EAGAIN	EAGAIN
EBADF	EBADF
EINVAL	EINVAL
e$invalid_io_operation	none

I_POP

Purpose

The I_POP command removes the top STREAMS module from a Stream.

Explanation

The control_structure for I_PUSH is a dummy (unused) two-byte integer.

Example

The example below shows programming statements for using the I_POP command.

```
opcode = I_POP;
short int  dummy;

s$ioctl (&port_id, &opcode, &dummy, &rval, &error_code);
```

The I_POP command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EINVAL	EINVAL
ENXIO	ENXIO
e$invalid_io_operation	none

I_PUNLINK

Purpose

The I_PUNLINK command disconnects two Streams previously linked by I_PLINK.

Explanation

The control_structure for I_PUNLINK is the four-byte multiplexer ID number previously returned in the rval parameter of the corresponding I_PLINK command.

Example

The example below shows programming statements for using the I_PUNLINK command.

```
long int id_number;
id_number = mux_id_n;     /* an existing multiplexer ID */
opcode = I_PUNLINK;

s$ioctl (&port_id, &opcode, &id_number, &rval, &error_code);
```

The I_PUNLINK command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
ENXIO	ENXIO
ETIME	ETIME
EAGAIN	EAGAIN
EINVAL	EINVAL
e$invalid_io_operation	none

I_PUSH

Purpose

The I_PUSH command pushes a STREAMS module onto an open Stream directly below the Stream head.

Explanation

The order in which modules are pushed is important in building the desired STREAMS stack. The most recently pushed module is the highest module on the stack, excluding the Stream head.

The control_structure for I_PUSH is declared as follows.

```
char name [FMNAMESZ];
```

The name field contains the name of the module to be pushed.

Note that in the s$ioctl call, name should be passed by value (as name), and not by reference (not as &name).

Example

The example below pushes the tcp_ip module onto the Stream directly below the Stream head.

```
strcpy (name, "tcp_ip");
opcode = I_PUSH;

s$ioctl (&port_id, &opcode, name, &rval, &error_code);
```

The I_PUSH control operation can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EFAULT	EFAULT
EINVAL	EINVAL
ENXIO	ENXIO
e$invalid_io_operation	none

I_RECVFD

Purpose

The `I_RECVFD` command receives the port ID passed by other applications via `I_SENDFD`.

Explanation

The `control_structure` for `I_RECVFD` is declared as follows.

```
struct LONGMAP strrecvfd
{
     int              fd;
     unsigned short   legacy_field1;
     unsigned short   legacy_field2;
     int              uid;
     int              gid;
     int              process_id;
     unsigned int     privilege_mask;

#define strrecvfd_privilege_mask_privileged 0x80000000
#define strrecvfd_privilege_mask_reserved_mask 0x7FFFFFFF
#define strrecvfd_privilege_mask_reserved_shift 0

     unsigned short           flags;

#define strrecvfd_flags_backup          0x8000
#define strrecvfd_flags_audit           0x4000
#define strrecvfd_flags_mbz_mask        0x3FFF
#define strrecvfd_flags_mbz_shift       0

     short                    character_set;
     int                      filler2 [25];
     char                     person_name [256];
     char                     group_name [256];
     char                     language [256];
} strrecvfd;
```

The `fd` element is the returned port ID. All of the remaining fields define properties of the sending process and/or the current user associated with that process (for additional information on process properties, see the description of the `s$get_process_info` subroutine in the OpenVOS Subroutines manuals):

- `uid` is the numeric user ID.
- `gid` is the numeric group ID.
- `process_id` is the process identifier.

- privilege_mask.privileged is the privileged bit for the process.
- flags.backup is the backup bit for the process.
- flags.audit is the audit bit for the process.
- character_set is the default graphic character set for the process.
- filler2 is reserved for future use.
- person_name is the text name associated with the user ID.
- group_name is the text name associated with the group ID.
- language is the national language environment for the process.

Example

The example below shows how the I_RECVFD command is used to receive a port ID and assign it to a local portid2 variable.

```
short int port_id;
struct strrecvfd recvfd;
short int portid2;
long int rval;
short int error_code;
long int opcode = I_RECVFD;

s$ioctl (&port_id, &opcode, &recvfd, &rval, &error_code);
port_id2 = recvfd.fd;
```

The I_RECVFD command can return the following error codes.

Note: Individual drivers may return additional error codes.

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
EBADMSG	EBADMSG
EFAULT	EFAULT
EMFILE	EMFILE
ENXIO	ENXIO
e$invalid_io_operation	none

I_SENDFD

Purpose

The `I_SENDFD` command passes a port ID to another application.

Explanation

If an application opens a port, it can then pass that port to another application by passing the port ID to an opened Stream using an `I_SENDFD` command.

In order to pass the port to the other application process, there first must be an open Stream between the two processes. Not all Streams support `I_SENDFD`: the Stream in question must be looped back by the driver in order to support `I_SENDFD`. It is driver specific whether or not there is a way to loop back open streams.

Not all kinds of ports can be passed through a stream using `I_SENDFD`. Currently this is supported for other Streams, simple file types, pipes, sockets, directories, `window_term` ports, and the null device.

The `control_structure` for `I_SENDFD` is a two-byte integer declared as follows.

```
short int portid_to_send;
```

Example

The example below shows programming statements for using the `I_SENDFD` command.

```
short int port_id;
long int opcode = I_SENDFD;
short int portid_to_send;
long int rval;
short int error_code;

s$ioctl (&port_id, &opcode, &portid_to_send, &rval, &error_code);
```

The I_SENDFD command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EAGAIN	EAGAIN
EBADF	EBADF
EINVAL	EINVAL
ENXIO	ENXIO
e$invalid_io_operation	none

I_SETCLTIME

Purpose

The `I_SETCLTIME` command sets the amount of time (in milliseconds) the Stream head will delay closing each module and driver when there is data on the write queues.

Explanation

Before closing each module and driver, the Stream head will delay for the specified number of milliseconds to allow the data to drain. After the specified delay, any data remaining on the queues is flushed.

The `control_structure` for `I_SETCLTIME` is declared as follows.

```
long int options;
```

The `options` argument specifies the number of milliseconds to delay. The default delay time is 15 seconds (15,000 milliseconds).

Example

The example below sets the delay time to 20,000 milliseconds.

```
long int options;
options = 20000;
opcode = I_SETCLTIME;

s$ioctl (&port_id, &opcode, &options, &rval, &error_code);
```

The `I_SETCLTIME` command can return the following error codes.

OpenVOS Error Code	`errno.h` **Equivalent**
`EINVAL`	`EINVAL`
`e$invalid_io_operation`	none

I_SETDELAY

Purpose

The I_SETDELAY command places a STREAMS device in blocking mode or nonblocking mode.

Explanation

The control_structure for I_SETDELAY is a two-byte integer declared as follows.

```
int delay_arg;
```

If delay_arg is set to STREAMS_ONDELAY, the STREAMS device is placed in nonblocking mode. If delay_arg is set to 0, the STREAMS device is placed in blocking mode.

Example

The example below places a STREAMS device in nonblocking mode.

```
int delay_arg;
delay_arg = STREAMS_ONDELAY;
opcode = I_SETDELAY;

s$ioctl (&port_id, &opcode, &delay_arg, &rval, &error_code);
```

The I_SETDELAY command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EINVAL	EINVAL
EAGAIN	EAGAIN
e$invalid_io_operation	none

I_SETSIG

Purpose

The I_SETSIG command is used to inform the Stream head of the events for which the application will receive SIGPOLL signals from the kernel when the events occur on a specified Stream.

Explanation

To receive SIGPOLL signals, an application must first register a handler for SIGPOLL. Only the C signal function can be used to register the handler. There is no corresponding OpenVOS subroutine available to perform this function. The signal function in STREAMS is an alternative for the poll function or the s$poll subroutine.

For a list of valid STREAMS events and their meanings, refer to the chapter on polling and signalling in the manual *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS*.

Example

The example below specifies four events for which the application will receive SIGPOLL signals.

```
long signal_flags;
opcode = I_SETSIG;
signal_flags = S_HIPRI | S_MSG | S_INPUT | S_OUTPUT;

s$ioctl (&port_id, &opcode, &signal_flags, &rval, &error_code);
```

The I_SETSIG command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
EINVAL	EINVAL
EAGAIN	EAGAIN
e$invalid_io_operation	none

I_SRDOPT

Purpose

The I_SRDOPT command sets the read options of a Stream.

Explanation

The control_structure for I_SRDOPT is a four-byte integer declared as follows.

```
int options;
```

The legal values for the options parameter are as follows.

Value	Meaning
RNORM	Byte-stream mode, this is the default. Read or s$read_raw will read as many bytes from the Stream head as the caller requested.
RMSGD	Message discard mode. Read or s$read_raw will read as many bytes as the user requested, or until a message boundary is reached. Any data remaining is discarded by the Stream head.
RMSGN	Message nondiscard mode. Same as RMSGD, except that any extra data is not discarded and can be retrieved by subsequent calls to read or s$read_raw.

Example

The example below sets the read options to message nondiscard mode.

```
int options;
options = RMSGN;  /* Do not discard incomplete messages.*/
opcode = I_SRDOPT;

s$ioctl (&port_id, &opcode, &options, &rval, &error_code);
```

The I_SRDOPT command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EINVAL	EINVAL
e$invalid_io_operation	none

I_STR

Purpose

STREAMS modules and drivers can define their own internal I/O control commands, which applications can access using the `I_STR` command. For details concerning this `s$ioctl` command, refer to the manual *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS*.

Explanation

This internal command mechanism is provided to enable an application to specify timeouts and variable amounts of data when sending an `s$ioctl` request to downstream modules and drivers. This command allows information to be sent with `s$ioctl`, and returns to the user any information sent upstream by the downstream recipient.

The `I_STR` command blocks application processing until the system responds with either a positive or negative acknowledgement message, or until the request times out, regardless of whether or not the application is in nonblocking mode.

The `control_structure` for the `I_STR` command is declared as follows.

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

Example

The example below sends the internal command `I_TP_SEND_DATAGRAM` to a downstream module or driver. Note that the memory pointed to by `datagram_ptr` must be contiguous memory; it cannot contain pointers to other memory addresses.

```
struct strioctl  i_str;
i_str.ic_cmd = I_TP_SEND_DATAGRAM;
i_str.ic_timout = -1; /* Wait forever to complete request.*/
i_str.ic_len = sizeof (struct datagram);
i_str.ic_dp = datagram_ptr;
opcode = I_STR;

s$ioctl (&port_id, &opcode, &i_str, &rval, &error_code);
```

The I_STR command can return the following error codes.

OpenVOS Error Code	errno.h Equivalent
EAGAIN	EAGAIN
EFAULT	EFAULT
EINVAL	EINVAL
ENXIO	ENXIO
ETIME	ETIME
e$invalid_io_operation	none

I_UNLINK

Purpose

The I_UNLINK command unlinks two previously linked drivers.

Explanation

The control_structure for the I_UNLINK command is identical to the I_LINK command, except that a four-byte integer id_number argument is used in place of port_id2. The id_number argument should contain the multiplexer ID number previously returned by an I_LINK command.

Example

The example below unlinks the drivers associated the multiplexer ID number contained in ip_portid.

```
int id_number;
id_number = ip_portid;
opcode = I_UNLINK;

s$ioctl (&portid, &opcode, &id_number, &rval, &error_code);
```

The I_UNLINK command can return the following error codes.

OpenVOS Error Code	errno.h **Equivalent**
ENXIO	ENXIO
ETIME	ETIME
EAGAIN	EAGAIN
EINVAL	EINVAL
e$invalid_io_operation	none

Streams Polling Using `s$poll`

For information on the `s$poll` subroutine, see the OpenVOS Subroutines manuals.

C Language Interface

Table 3-3 lists the OpenVOS Standard C language functions and the equivalent OpenVOS subroutine that can be used with OpenVOS STREAMS.

Table 3-3. OpenVOS C Functions and Equivalent OpenVOS Subroutines

OpenVOS C Function	Equivalent OpenVOS Subroutine
`close`	`s$streams_close`
`fcntl`	`s$ioctl`
`getmsg/getpmsg`	`s$getmsg/s$getpmsg`
`ioctl`	`s$ioctl`
`open`	`s$streams_open`
`poll*`	`s$poll`
`putmsg/putpmsg`	`s$putmsg/s$putpmsg`
`read` and `readv`	`s$read_raw`
`select` and `select_with_events`	`s$poll`
`signal`	None
`write` and `writev`	`s$write_raw`

* `poll` provides only a subset of `s$poll` functionality.

For a description of the `poll` function, see the *OpenVOS Standard C Reference Manual* (R363). The include files `stropts.h`, `fcntl.h`, and `poll.h` contain function prototype definitions for the C functions that are used exclusively with OpenVOS STREAMS. For additional information on OpenVOS Standard C, refer to the *OpenVOS C Subroutines Manual* (R068) and the *OpenVOS Standard C Reference Manual* (R363).

close

The `close` function for OpenVOS STREAMS operates identically to the existing OpenVOS C `close` function.

fcntl

Purpose

The `fcntl` function places a STREAMS device in blocking mode or nonblocking mode, depending on the setting of the appropriate arguments. It can also be used to return the current mode of the STREAMS device.

Usage

```
#include <fcntl.h>

int fcntl(int, int, ...);

     fnctl(fd, cmd, arg);
```

Arguments

- `fd` (input)
  Specifies the file descriptor of an opened Stream.

- `cmd` (input)
  Specifies the command to be executed by `fcntl`. Currently, `cmd` can be set to `F_SETFL` to set the file descriptor flag `O_NDELAY`, or to `F_GETFL` to return the current flag settings.

- `arg` (input)
  Specifies the arguments to the specified `cmd`. Currently, `arg` can be set to the following file descriptor flag values only for the `F_SETFL` command.

Value	**Meaning**
`0`	Blocking I/O for the Stream
`O_NDELAY`	Nonblocking I/O for the Stream

Note that `arg` is ignored for the `F_GETFL` command.

Explanation

The only commands currently supported by `fnctl` are `F_SETFL` and `F_GETFL`, which allow the caller to set or obtain the value of the `O_NDELAY` file descriptor flag.

If `fcntl` is called with the command `F_SETFL`, it returns a value of `0` on success and a value of `-1` on failure (and sets `errno`). If `fcntl` is called with the command `F_GETFL`, it returns the value of the flag (`0` or `O_NDELAY`), or `-1` on failure (and sets `errno`).

getmsg/getpmsg

Purpose

The functions `getmsg` and `getpmsg` are used to receive a STREAMS data message and/or control message from the Stream head. Unless otherwise specified, information presented on `getmsg` also applies to `getpmsg`.

The declarations of both `getmsg` and `getpmsg` are shown below.

Usage

```
#include <stropts.h>

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

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

     getmsg(fd, *ctlptr, *dataptr, *flagsp);
     getpmsg(fd, *ctlptr, *dataptr, *bandp, *flagsp);
```

Arguments

- `fd` (input)
  Specifies the file descriptor of an opened Stream.

- `*ctlptr` (output)
  A pointer to a `strbuf` structure in which the control message, if any, is returned.

- `*dataptr` (output)
  A pointer to a `strbuf` structure in which the data message, if any, is returned.

- `*bandp` (input/output)
  For use with `getpmsg` only, specifies the priority band of the message to be received. The `*bandp` parameter for `getpmsg` is used in conjunction with the `*flagsp` parameter, as described below.

- `*flagsp` (input/output)
  Specifies the priority band of the message to be received. It is used by `getmsg` and `getpmsg` differently, as described below.

Explanation

On success, getmsg and getpmsg return a value of 0 to indicate that a complete message has been returned. A return value of MORECTL indicates that a control message is queued at the Stream head. A return value of MOREDATA indicates that a data message is queued at the Stream head. A value of MORECTL|MOREDATA indicates that both message types are queued at the Stream head. These constants are defined in the include file stropts.h.

On failure, getmsg and getpmsg return -1 (and set errno).

For getmsg, if the *flagsp parameter is set to RS_HIPRI, the function returns only high-priority messages. By default, getmsg returns the first available message queued at the Stream head. If *flagsp is set to 0, getmsg returns any message queued at the Stream head and, if a high-priority message is returned, getmsg sets *flagsp to RS_HIPRI.

For getpmsg, the *flagsp parameter points to a bit mask with the following mutually exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like getmsg, getpmsg returns the first available message queued at the Stream head. By setting *flagsp to MSG_HIPRI and *bandp to 0, the user requests to receive only a high-priority message, in which case, getpmsg will return the next available message only if it is of high priority.

To receive a message of any priority band, the user sets *flagsp to MSG_BAND and sets *bandp to a specific priority band. In this case, getpmsg will return the next available message only if its priority is equal to or greater than the priority band pointed to by *bandp, or if it is a high-priority message.

To receive the first message queued at the Stream head, the user sets *flagsp to MSG_ANY and sets *bandp to 0. On return, *flagsp is set to MSG_HIPRI and *bandp to 0 if the message returned is a high-priority message. Otherwise, on return, *flagsp is set to MSG_BAND, and *bandp is set to the priority band of the returned message.

For further information on these functions, refer to the manual *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS*.

ioctl

Purpose

The `ioctl` function performs a control operation on an opened Stream. It returns an error if it is called for a non-STREAMS device.

Usage

```
#include <stropts.h>

int ioctl (int fd, int request, void *arg);

     ioctl (fd, request, *arg);
```

Arguments

- `fd` (input)
  Specifies the file descriptor of a Stream that was previously opened by the application.

- `request` (input)
  Specifies the `ioctl` control operation. Refer to the sections on `s$ioctl` control operations presented previously in this chapter for a description of each control operation. Also, refer to the manual *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS* for further information on `ioctl` control operations.

- `*arg` (input/output)
  A pointer to the parameter(s) specific to the control operation (`request`) being used.

Explanation

Each `ioctl` control operation (command) is shown in Table 3-2 and is described in the sections following the table.

The `ioctl` commands, defined constants, structures, and return values are contained within the `stropts.h` include file. Note that the command `I_SETDELAY` is not supported by `ioctl` (it can only be used with `s$ioctl`). For specific information on the commands, arguments, and return values, refer to the manual *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS*.

After calling `ioctl`, application processing is suspended until the command request has been serviced, even if the application is in nonblocking mode.

open

The `open` function is the normal OpenVOS Standard C operation to open a device, and it is used to open a STREAMS device.

The `open` function returns a file descriptor, which is used to identify the STREAMS device in subsequent I/O calls for the device.

See the *OpenVOS Standard C Reference Manual* (R363) for a detailed explanation of the `open` function.

poll

For information on the `poll` function, see the *OpenVOS Standard C Reference Manual* (R363).

putmsg/putpmsg

Purpose

The functions `putmsg` and `putpmsg` send a STREAMS data and/or control message from application buffers downstream to a STREAMS device or module. The function `putpmsg` is almost identical to `putmsg`, but has an additional argument to specify the priority band at which to send the message.

Unless otherwise specified, information presented on `putmsg` also applies to `putpmsg`.

STREAMS messages sent using `putmsg` can contain control information, data, or both. The application passes a data and control message to `putmsg` using separate buffers for each. The ability to send control information along with data distinguishes `putmsg` from the `write` function. The control message is typically directed to a specific module or driver in the Stream.

Note that according to the STREAMS standard, an application that is sending only a control message can pass either a zero-length data buffer or no data buffer (a null buffer pointer). OpenVOS STREAMS supports this feature, but in a restricted manner, as described below.

The declarations of both `putmsg` and `putpmsg` are shown below.

Usage

```
#include <stropts.h>;

int putmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int flagsp);

int putpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int bandp,
            int flagsp);

     putmsg(fd, ctlptr, dataptr, flagsp);
     putpmsg(fd, ctlptr, dataptr, bandp, flagsp);
```

Arguments

- `fd` (input)
  Specifies the file descriptor of an opened Stream.

- `*ctlptr` (input)
  A pointer to a `strbuf` structure (defined in `stropts.h`) in which the control message, if any, is contained. No control part is sent if either `ctlptr` is `NULL` or the `len` field of `ctlptr` is set to `-1`. Note that if the control part is sent, the Stream head ensures that it is at least 64 bytes in length.

- ▶ `*dataptr` (input)
  A pointer to a `strbuf` structure in which the data message, if any, is contained. To send a data message, `dataptr` must not be `NULL` and the `len` field of `dataptr` must have a value of `0` or greater. No data message is sent if either `dataptr` is `NULL` or the `len` field of `dataptr` is set to `-1`.

- ▶ `bandp` (input)
  For use with `putpmsg` only, specifies the priority band of the message to be sent. The `bandp` parameter for `putpmsg` is used in conjunction with the `flagsp` parameter, as described below.

- ▶ `flagsp` (input/output)
  Specifies the priority band of the message to be received. It is used by `getmsg` and `getpmsg` differently, as described below.

Explanation

On success, `putmsg` and `putpmsg` return a value of `0`. On failure, they return a value of `-1` (and set `errno`).

`s$putmsg` and `s$putpmsg` use the `flagsp` parameter with three flags.

`The s$PUTMSG_PARTIAL` flag (in `stropts.incl`) can be passed to `s$putmsg` and `s$putpmsg`. This flag can be used to inform the driver that the application will be sending more data. How the driver responds to this flag is driver-specific. The driver is not required to transport the data until the application uses `s$putmsg` without the `s$PUTMSG_PARTIAL` flag.

Note the following additional information about `s$putmsg` and `s$putpmsg`:

- For `s$putmsg`, the `flagsp` parameter sends priority messages to a STREAMS device. In addition to `s$PUTMSG_PARTIAL`, the flags available to an application for `s$putmsg` are `0`, indicating a normal priority message, and `RS_HIPRI`, indicating a high-priority message.

- For `s$putpmsg`, the `flagsp` parameter points to a bit mask with the mutually exclusive flags defined, `MSG_HIPRI` and `MSG_BAND`. If the integer pointed to by `flagsp` is set to `0`, `s$putpmsg` fails and returns `EINVAL`.

  If a control part is defined and `flagsp` is set to `MSG_HIPRI` and `bandp` is set to `0`, a high-priority message is sent. If `flagsp` is set to `MSG_HIPRI` and either no control part is defined or `bandp` is set to a nonzero value, `s$putpmsg` fails and returns `EINVAL`.

  `MSG_BAND` is valid only for `s$putpmsg`. If `flagsp` is set to `MSG_BAND`, then a message is sent at the priority band specified by `bandp`. If a control part and data part are not specified and `flagsp` is set to `MSG_BAND`, no message is sent and `0` is returned.

See *AT&T UNIX System 5 Release 4 Programmer's Guide: STREAMS* for more information on using the `putmsg` and `putpmsg` functions.

read

The `read` function is the normal OpenVOS Standard C operation to read from a device, and it is used to read from a STREAMS device. For more information, see *OpenVOS Standard C Reference Manual* (R363).

readv

The `readv` function is the normal OpenVOS Standard C operation to read from multiple devices, and it is used to read from a STREAMS device into multiple input buffers. For more information, see *OpenVOS Standard C Reference Manual* (R363).

select

The `select` function indicates which of the specified file descriptors can be read, written, or has an error condition pending. For more information, see *OpenVOS Standard C Reference Manual* (R363).

select_with_events

The select_with_events function indicates which of the specified file descriptors can be read, written, or has an error condition pending, and indicates operating-system events. For more information, see *OpenVOS Standard C Reference Manual* (R363).

signal

The `signal` function signal `SIGPOLL` is used by the Stream head to indicate that a Stream action is required. It is supported by OpenVOS C. For further information on the use of the `signal` function on a Stream, refer to *AT&T UNIX System 5 Release 4 Programmer's Guide: STREAMS*.

write

The `write` function is the normal OpenVOS Standard C operation to write to a device, and it is used to write to a STREAMS device. For more information, see *OpenVOS Standard C Reference Manual* (R363).

writev

The `writev` function is the normal OpenVOS Standard C operation to write to multiple devices, and it is used to write to a STREAMS device, combining multiple output buffers to form one message. For more information, see *OpenVOS Standard C Reference Manual* (R363).

OpenVOS STREAMS Return Codes

All OpenVOS STREAMS return codes referenced in this chapter are summarized below in Table 3-4.

Table 3-4. OpenVOS STREAMS Return Codes

OpenVOS Return Code	errno.h	Explanation
EAGAIN	EAGAIN	This error generally means that a requested resource was not available to complete a given operation. The application can reissue the request.
EBADF	EBADF	An invalid or unopened port ID was used in an operation.
EBADMSG	EBADMSG	The message at the Stream head (usually for the read operation) does not contain a valid message.
EFAULT	EFAULT	An address fault occurred during request processing.
EINVAL	EINVAL	An illegal parameter was used in an operation.
ENODATA	ENODATA	There is no message at the Stream head.
ENOENT	ENOENT	This error usually means a search of an item, such as a driver, module, or STREAMS device has failed.
ENOMEM	ENOMEM	The STREAMS environment has insufficient memory to finish an operation.
ENXIO	ENXIO	A hangup has been received by a given Stream referenced through a port ID.
ERANGE	ERANGE	A parameter contained a value out of its legal range.
ETIME	ETIME	A time limit has been exceeded for a request.

The following OpenVOS return codes can also be returned to an OpenVOS STREAMS application.

`e$device_already_assigned` (1155) – The device is already assigned to another process.

`e$invalid_io_operation` (1040) – Invalid I/O operation for current port state or attachment.

`e$device_not_found` (1220) – The specified device name is not known to the system.

`e$clone_limit_exceeded` (7136) – The maximum number of clones for the device, as defined in the `devices.tin` file, has been exceeded.

Appendix A:
PL/I Usage

s$streams_open

```
%include  'errno.incl.pl1';
%include  'system_io_constants.incl.pl1';
declare   port_id                fixed bin (15);
declare   path_name              char (256) varying;
declare   file_org               fixed bin (15);
declare   max_len                fixed bin (15);
declare   io_type                fixed bin (15);
declare   lock_mode              fixed bin (15);
declare   access_mode            fixed bin (15);
declare   index_name             char (32) varying;
declare   error_code             fixed bin (15);

declare   s$streams_open entry ( fixed bin (15),
                                 char (256) varying,
                                 fixed bin (15),
                                 fixed bin (15),
                                 fixed bin (15),
                                 fixed bin (15),
                                 fixed bin (15),
                                 char (32) varying,
                                 fixed bin (15));

          call s$streams_open (  port_id,
                                 path_name,
                                 file_org,
                                 max_len,
                                 io_type,
                                 lock_mode,
                                 access_mode,
                                 index_name,
                                 error_code);
```

s$streams_close

```
declare   port_id                   fixed bin (15);
declare   error_code                fixed bin (15);

declare   s$streams_close entry (  fixed bin (15),
                                   fixed bin (15));

          call s$streams_close (   port_id,
                                   error_code);
```

s$getmsg/s$getpmsg

```
%include  'errno.incl.pl1';
declare   port_id           fixed bin (15);
declare   ctl_maxlen        fixed bin (31);
declare   ctl_len           fixed bin (31);
declare   ctl_bufp          pointer;
declare   data_maxlen       fixed bin (31);
declare   data_len          fixed bin (31);
declare   data_bufp         pointer;
declare   bandp             pointer;              /* s$getpmsg only */
declare   flagsp            pointer;
declare   rval              fixed bin (31);
declare   error_code        fixed bin (15);

declare   s$getmsg entry ( fixed bin (15),
                           fixed bin (31),
                           fixed bin (31),
                           pointer,
                           fixed bin (31),
                           fixed bin (31),
                           pointer,
                           pointer,
                           fixed bin (31),
                           fixed bin (15));

declare   s$getpmsg entry ( fixed bin (15),
                            fixed bin (31),
                            fixed bin (31),
                            pointer,
                            fixed bin (31),
                            fixed bin (31),
                            pointer,
                            pointer,
                            pointer,
                            fixed bin (31),
                            fixed bin (15));

          call s$getmsg (  port_id,
                           ctl_maxlen,
                           ctl_len,
                           ctl_bufp,
                           data_maxlen,
                           data_len,
                           data_bufp,
                           flagsp,
                           rval,
                           error_code);
```

(Continued on next page)

```
call s$getpmsg (  port_id,
                  ctl_maxlen,
                  ctl_len,
                  ctl_bufp,
                  data_maxlen,
                  data_len,
                  data_bufp,
                  bandp,
                  flagsp,
                  rval,
                  error_code);
```

s$putmsg/s$putpmsg

```
%include  'errno.incl.pl1';
declare   port_id          fixed bin (15);
declare   ctl_len          fixed bin (31);
declare   ctl_bufp         pointer;
declare   data_len         fixed bin (31);
declare   data_bufp        pointer;
declare   bandp            pointer;             /* s$putpmsg only */
declare   flagsp           pointer;
declare   error_code       fixed bin (15);

declare   s$putmsg entry ( fixed bin (15),
                           fixed bin (31),
                           pointer,
                           fixed bin (31),
                           pointer,
                           pointer,
                           fixed bin (15));

declare   s$putpmsg entry ( fixed bin (15),
                            fixed bin (31),
                            pointer,
                            fixed bin (31),
                            pointer,
                            pointer,
                            pointer,
                            fixed bin (15));

          call s$putmsg (  port_id,
                           ctl_len,
                           ctl_bufp,
                           data_len,
                           data_bufp,
                           flagsp,
                           error_code);

          call s$putpmsg (  port_id,
                            ctl_len,
                            ctl_bufp,
                            data_len,
                            data_bufp,
                            bandp,
                            flagsp,
                            error_code);
```

s$ioctl

```
%include  'errno.incl.pl1';
declare   port_id           fixed bin (15);
declare   opcode            fixed bin (15);
declare   control_structure fixed bin (15);
declare   rval              fixed bin (31);
declare   error_code        fixed bin (15);

declare   s$ioctl entry (   fixed bin (15),
                            fixed bin (15),
                            fixed bin (15),
                            fixed bin (31),
                            fixed bin (15));

          call s$ioctl (    port_id,
                            opcode,
                            control_structure,
                            rval,
                            error_code);
```

s$poll

For information on `s$poll`, see the OpenVOS Subroutines manuals.

Glossary

acknowledgment

The transmission, by the receiver, of characters that serve as positive confirmation to the sender.

adapter

See **I/O adapter**.

American Standard Code for Information Interchange (ASCII)

A standard 7-bit character representation code that the operating system stores in an 8-bit byte.

ASCII

In the OpenVOS internal character coding system, the half of the 8-bit code page with code values in the range 00-7F (hexadecimal), representing the American Standard Code for Information Interchange.

See also **American Standard Code for Information Interchange**.

attach

To associate a port with a file or device, creating the port if necessary, and generating a port ID. The port ID can then be used to refer to the file or device.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding compacts the code and resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the `bind` command.

binder control file

A text file containing directives for the binder.

blocked

In STREAMS, a state in which a queue's service procedure cannot be enabled or executed due to flow control.

clone device

A STREAMS device that returns an unused major/minor device when initially opened, rather than requiring the minor device to be specified by name in the `open` call.

configuration table

One of the table files that the operating system uses to identify the elements of a system or a network. For example, the file `devices.table` contains information about each device present in the system, the file `disks.table` contains information about each of the disks present in a system, and the file `modules.table` contains information about each module in the system.

debug

To correct errors (bugs) in a program.

debugger

An OpenVOS tool used as an aid in finding program errors.

detach

To disassociate a port from a file or device.

device configuration table

The file `devices.table`, which contains information about each device present in a system. See also **configuration table**.

device driver

A Stream component whose principle functions are handling an associated physical device and transforming data and information between the external interface and the Stream.

device name

The name of a device. Device names are specified by the system administrator in the device configuration table. The path name of a device has two components: (1) the name of the system, prefixed by a percent sign, and (2) the name of the device, prefixed by a number sign (for example, `%s1#sales_printer`).

devices table

A file with the name `devices.table` that contains information about each device present in a system. See also **configuration table**.

`devices.tin` file

The table input file that contains definitions of all the devices, except disks, attached to a system. These devices can be I/O adapters, line adapters, terminals, printers, or tape drives. The OpenVOS operating system uses this file to create the devices table. See also **devices table**.

downstream

In STREAMS, the direction of data flow going from the stream head toward the driver. Also called *write side* or *output side*.

driver

In STREAMS, a module that forms the Stream end. It can be a device driver or a pseudo-device driver. It is a required component of a Stream. It typically handles data transfer between the kernel and a device and does little or no processing of data.

flow control

In STREAMS, a mechanism that regulates the rate of message transfer within a Stream and from user space into a Stream.

input side

In STREAMS, the direction of data flow going from the driver toward the stream head. Also called *read side* or *upstream*.

include file

1. A file that the compiler includes in the source module used by the compilation process. The name of the include file must be specified in a language-specific directive within the source module.

2. A text file containing language statements, compile-time statements, or both, that the compiler inserts into the source module in place of an `include` compile-time statement. For example, OpenVOS PL/I include files have the suffix `.incl.pl1`.

kernel

The part of the operating system that performs all privileged system operations. The command processor and the debugger are included in the kernel.

message

In STREAMS, one or more linked message blocks. A message is referenced by its first message block, and its type is defined by the message type of that block.

message block

In STREAMS, a triplet consisting of a data buffer and associated control structures. It carries data or control information, as identified by its message type, in a Stream.

message queue

In STREAMS, a linked list of zero or more messages connected together.

message type

In STREAMS, a defined set of values identifying the contents of a message.

module

In STREAMS, a defined set of kernel-level routines and data structures used to process data, status, and control information on a Stream. There can be zero or more modules in one Stream. Each module provides a pair of message queues (read queue and write queue) and communicates to other components in a Stream by passing messages on these queues.

multiplexer

A STREAMS mechanism that allows messages to be routed among multiple Streams in the kernel. A multiplexing configuration includes at least one multiplexing pseudo-device driver connected to one or more upper Streams and one or more lower Streams.

object module

A file produced by a compiler that contains the machine-code version of one or more procedures; it usually contains symbolic references to external variables and programs. To execute the program, an object module must be processed by the binder to produce a program module, and then loaded by the loader.

output side

In STREAMS, the direction of data flow going from the stream head toward the driver. Also called *write-side* or *downstream*.

pop

In STREAMS, to remove the module immediately below the stream head. See also **push**.

port attachment

The creation of a port for the purpose of accessing a file or device.

port detachment

The disassociation of a port from a file or device.

port ID

A two-byte integer used to identify a port.

pseudo-device driver

In STREAMS, a software driver, not directly associated with a physical device, that performs functions internal to a Stream. An example of a pseudo-device driver is a multiplexer or log driver.

push

In STREAMS, to insert a module in a Stream immediately below the stream head. See also **pop**.

put procedure

In STREAMS, a routine, in a module or a driver associated with a queue, that receives messages from the preceding queue. It is the single entry point into a queue from a preceding queue. It may perform processing on a message and will then generally either queue the message for subsequent processing by the queue's service procedure or pass the message to the put procedure of the following queue.

queue

In STREAMS, a data structure that contains status information, a pointer to routines processing messages, and pointers for administering the Stream. It typically contains pointers to a put and a service procedure, a message queue, and private (internal) data.

read queue

In STREAMS, a message queue in a module or driver containing messages moving upstream. Associated with the `read` system call and input from a driver.

read side

In STREAMS, a direction of data flow going from a driver toward the stream head. Also called *upstream* and *input side*.

service interface

In STREAMS, a set of primitives that define a service at the boundary between a service user and a service provider, and the rules for allowable sequences of primitives across the boundary. At a Stream/user boundary, the primitives are typically contained in the control part of a message; within a Stream, they are contained in `M_PROTO` or `M_PCPROTO` message blocks.

service procedure

In STREAMS, a routine in a module or driver associated with a queue which receives messages queued for it by the put procedure of that queue. The procedure is called by the STREAMS scheduler. It may perform processing on the message and, generally passes the message to the put procedure of the following queue.

service provider

An entity in a service interface that responds to request primitives from the service user with response and event primitives.

service user

An entity in a service interface that generates request primitives for the service provider and receives response and event primitives.

status code

A two-byte integer, with an associated name, indicating the success, failure, or other status of an operation. A zero status code generally indicates a successful operation.

In the operating system, there are four types of status codes:

- error codes, whose names begin with the prefix `e$`
- message codes, whose names begin with the prefix `m$`
- query codes, whose names begin with the prefix `q$`
- response codes, whose names begin with the prefix `r$`.

Stream

A kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are the Stream head, the driver, and zero or more pushable modules between the Stream head and driver.

stream end

The Stream component furthest from the user process, containing a driver.

stream head

The Stream component closest to the user process. It provides the interface between the Stream and the user process.

STREAMS

A kernel mechanism that provides the framework for network services and data communications. It defines interface standards for character input and output within the kernel and between the kernel and user levels.

upstream

In STREAMS, a direction of data flow going from a driver towards the stream head. Also called *read side* and *input side*.

write queue

In STREAMS, a message queue in a module or driver containing messages moving downstream. Associated with the `write` system call and output from a user process.

write side

In STREAMS, a direction of data flow going from the Stream head toward the driver. Also called *downstream* and *output side*.

Index

T

U

W

Z