STREAMS Processing Routines 4

Put and Service Procedures

The put(9E) and srv(9E) procedures in the queue are routines that process messages as they transit the queue. The processing is generally performed according to the message type and can result in a modified message, new message(s), or no message. A resultant message, if any, is generally sent in the same direction in which it was received by the queue, but may be sent in either direction. Each put(9E) procedure places messages on its queue as they arrive, for later processing by the service procedure.

A queue will always contain a put procedure and may also contain an associated service procedure. Having both a put(9E) and srv(9E) procedure in a queue enables STREAMS to provide the rapid response and the queuing required in multi-user systems.

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

Put Procedure

A `put` procedure is the queue routine that receives messages from the preceding queues in the Stream. Messages are passed between queues by a procedure in one queue calling the `put` procedure contained in the following queue. A call to the `put` procedure in the appropriate direction is generally the only way to pass messages between STREAMS components. There is usually a separate `put` procedure for the read and write queues because of the full-duplex operation of most Streams. However, there can be a single `put` procedure shared between both the read and write queues. The syntax for the put procedure is shown in the following example. For more information, see Appendix F, “Manual Pages" for more information.

```
int prefixrput(queue_t *q, mblk_t *mp);
int prefixwput(queue_t *q, mblk_t *mp);
```

where *q* is a pointer to the `queue`(9S) structure, and *mp* is a pointer to the message block. See `msgb`(9S).

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

The `put` procedure always executes before the `service` routine for each specific message. In the multithreaded environment, a `put` procedure and a scheduled `service` procedure may be running simultaneously in different threads.

Each STREAMS component accesses the adjacent `put` procedure indirectly using the routine `putnext` (9F).

Note – A module should never directly call other module routines, including `put` and `service` procedures.

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

Note – STREAMS modules do not know which modules they are connected to, so the `put` procedures cannot depend on a message being handled only by `put` procedures at the stream head or in the driver. Any other modules along the Stream may queue the message and process it with a `service` procedure.

Service Procedure

In addition to the `put` procedure, a `service` procedure may be contained in each queue to allow deferred message processing. If a queue has both a `put` and `service` procedure, message processing will generally be divided between the procedures. The `put` procedure is always called first, from a preceding queue. After completing its part of the message processing, it arranges for the `service` procedure to be called by passing the message to the `putq`(9F) procedure. `putq` does two things: it places the message on the message queue of the queue and links the queue to the end of the STREAMS scheduling queue. When `putq`() returns to the `put` procedure, the procedure can return or continue to process the message. Some time later, the `service` procedure will be automatically called by the STREAMS scheduler. The syntax for the `service` procedure looks like this:

```
int prefixrsrv(queue_t *q);
int prefixwsrv(queue_t *q);
```

The STREAMS scheduler is separate and distinct from the system process scheduler. It is concerned only with queues linked to the STREAMS scheduling queue. The scheduler calls each `service` procedure of the scheduled queues one at a time in a First-In-First-Out (FIFO) manner. There is a dedicated thread for `service` procedure scheduling. `Put` procedures may be called on any thread in the kernel.

Note – The scheduling of queue `service` procedures is machine dependent and implementation specific. However, given the multithreaded architecture, service routines may not have finished running before returning to user level.

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

A `service` procedure can use a `putbq`(9F) to put messages back on a queue due to flow control or other reasons. A high-priority message must *never* be put back on the queue. This may result in an infinite loop in calling the service procedure. High priority messages would usually not get queued in the first place.

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

Rules that should be observed in `put` and `service` procedures are listed in Chapter 7, “Overview of Modules and Drivers".

An Asynchronous Protocol Stream Example

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

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

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

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

These options are set with the following modules:

`CHARPROC`
: Provides input character processing functions, including dynamically setable (via control messages passed to the module) character echo and parity checking. The module's default settings are to echo characters and not check character parity.

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

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

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

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

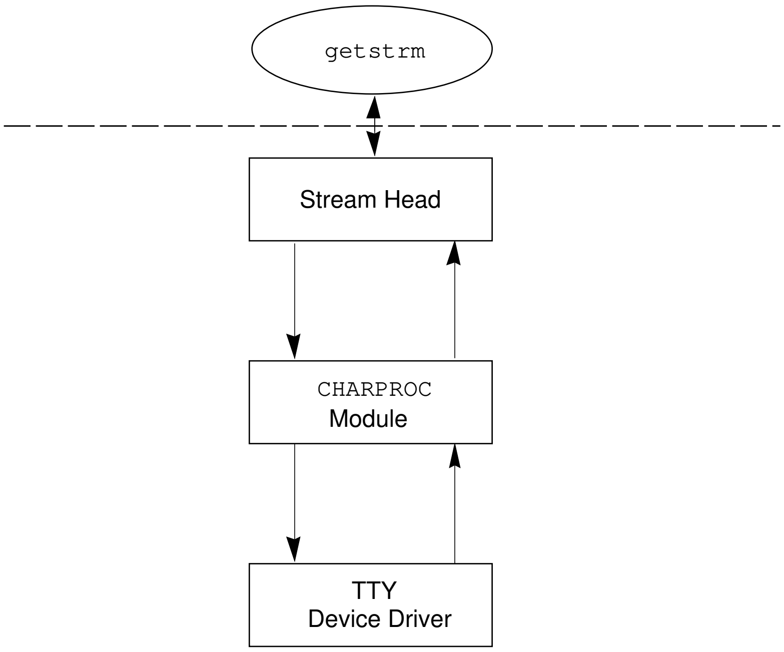

Figure 4-1 Idle Stream Configuration Example

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

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

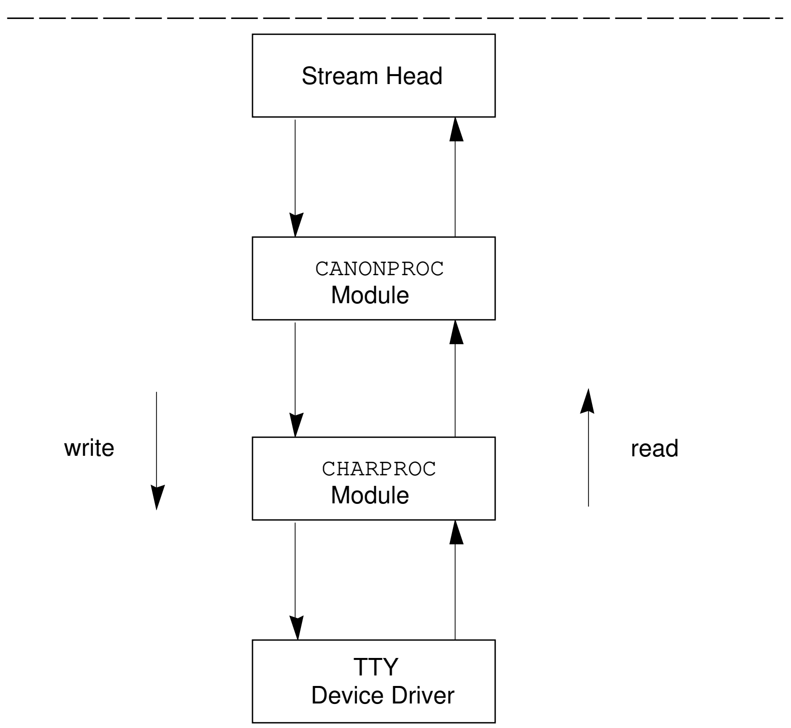

Figure 4-2 Operational Stream for Example

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

Read-Side Processing

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

Driver Processing

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

CHARPROC

`CHARPROC` has both `put` and `service` procedures on its read-side. In the example, the other queues in the modules also have both procedures:

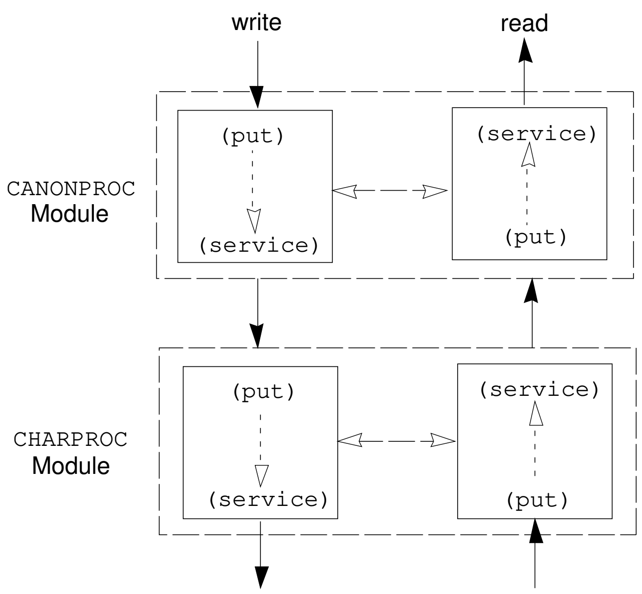

Figure 4-3 Module Put and Service Procedures

When the driver calls `CHARPROC`'s read queue `put` procedure, the procedure checks private data flags in the queue. In this case, the flags indicate that echoing is to be performed (recall that echoing is optional and that you are working with port hardware that cannot automatically echo). `CHARPROC` causes the echo to be transmitted back to the terminal by first making a copy of the message with a STREAMS utility routine. Then, CHARPROC uses another utility routine to obtain the address of its own write queue. Finally, the CHARPROC read put procedure calls its write put procedure and passes it the message copy. The write procedure sends the message to the driver to effect the echo and then returns to the read procedure.

This part of read-side processing is implemented with put procedures so that the entire processing sequence occurs as an extension of the driver input character interrupt. The CHARPROC read and write put procedures appear as subroutines (nested in the case of the write procedure) to the driver. This manner of processing is intended to produce the character echo quickly.

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

CANONPROC

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

Write-Side Processing

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

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

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

Analysis

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