Messages 5

Message Overview

Messages are the means of communication within a Stream. All input and output under STREAMS is based on messages. The objects passed between Streams components are pointers to messages. All messages in STREAMS use two data structures to refer to the data in the message. These data structures describe the type of the message and contain pointers to the data of the message, as well as other information. Messages are sent through a Stream by successive calls to the `put`(9E) routine of each queue in the Stream. Messages may be generated by a driver, a module, or by the Stream head.

Message Types

There are several different STREAMS messages (see Appendix B, “Message Types"). The messages differ in their intended purpose and their queueing priority. The contents of certain message types can be transferred between a process and a Stream by use of system calls.

The message types are briefly described and classified according to their queueing priority.

Ordinary Messages (also called `normal messages`):

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

High Priority Messages:

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

Note – Transparent `ioctls`, among other things, support applications developed before the introduction of STREAMS.

Expedited Data

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

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

Expedited data is mainly for exceptional cases and transmission of control signals. These are emergency data that are processed immediately, ahead of normal data. These messages are placed ahead of normal data in the queue, but after STREAMS high-priority messages and after any expedited data already in the queue.

Expedited data flow control is unaffected by the flow control constraints of normal data transfer. Expedited data have their own flow control because they can easily use all the system buffers if their flow is unrestricted.

Drivers and modules define separate high and low water marks for priority band data flow. (Watermarks are defined for each queue and they indicate the upper and lower limit of bytes that can be contained in the queue; see `M_SETOPTS` in “Ordinary Messages” on page 331). The default water marks for priority band data and normal data is the same. The Stream head also ensures that incoming priority band data is not blocked by normal data already in the queue. This is accomplished by associating a priority with the messages. This priority implies a certain ordering of the messages in the queue. (Message queues and priorities are discussed in the section “Message Queues and Message Priority” on page 66.)

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

5

Message Structure

All messages are composed of one or more message blocks. A message block is a linked list of triples, each consisting of two structures and a data buffer. The structures are a message block (`msgb`(9S)) and a data block (`datab`(9S)). The data buffer is a location in memory where the data of the message are stored.

```
struct msgb {
    struct msgb        *b_next;    /*next msg on queue*/
    struct msgb        *b_prev;    /*previous msg on queue*/
    struct msgb        *b_cont;    /*next msg block of message*/
    unsigned char      *b_rptr;    /*1st unread byte in bufr*/
    unsigned char      *b_wptr;    /*1st unwritten byte in bufr*/
    struct datab       *b_datap;   /*data block*/
    unsigned char      b_band;     /*message priority*/
    unsigned short     b_flag;     /*message flags*/
};

typedef struct msgb mblk_t;

struct datab {
    unsigned char *db_base;        /* first byte of buffer */
    unsigned char*db_lim;          /* last byte+1 of buffer */
    unsigned char db_ref;          /* msg count ptg to this blk */
    unsigned char db_type;         /* msg type */
};

typedef struct datab dblk_t;
```

The STREAMS framework uses the *b_next* and *b_prev* fields to link messages into queues. Drivers and modules may read, but not directly modify these fields. *b_rptr* and *b_wptr* specify the current read and write pointers respectively, in the data buffer pointed to by *b_datap*. *b_rptpr* and *b_wptr* are maintained by drivers and modules.

The field *b_band* determines where the message is placed when it is queued using the STREAMS utility routines. This field has no meaning for high priority messages and is set to zero for these messages. When a message is allocated via `allocb`(), the *b_band* field will be initially set to zero. Modules and drivers may set this field, if so desired, to a value from 0 to 255 depending

on the number of priority bands needed. Lower numbers are lower priority. The kernel will incur overhead in maintaining bands if non-zero numbers are used.

The `datab` structure specifies the data buffers' fixed limits (`db_base` and `db_lim`), a reference count field (`db_ref`), and the message field (`db_type`).

Note – SunOS has `b_band` in the `msgb` struct. Some other STREAMS implementations place `b_band` in the `datab` structure. The SunOS implementation is more flexible because each message is independent. For shared data blocks, the `b_band` may be different in the SunOS implementation, but not in other implementations.

A message consists of one or more linked message blocks. Multiple message blocks in a message can occur, for example, because of buffer-size limitations, or as the result of processing that expands the message. When a message is composed of multiple message blocks, the type associated with the first message block determines the overall message type, regardless of the types of the attached message blocks. Figure 5-1 illustrates two messages, each with multiple message blocks, to demonstrate the relationship of these data structures.

5

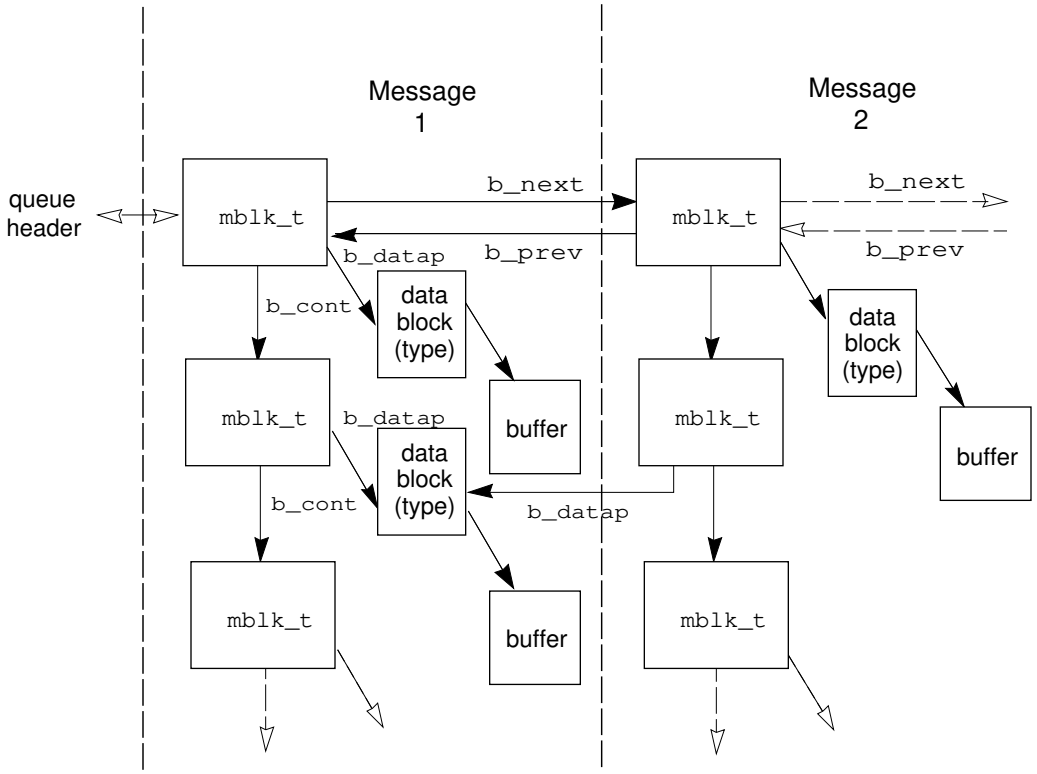

Figure 5-1 Message Form and Linkage

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

Note that a data block in message 1 is shared between message 1 and another message. Multiple message blocks can point to the same data block to conserve storage and to avoid copying overhead. For example, the same data block, with its associated buffer, may be referenced in two messages, from separate modules that implement separate protocol levels. (Figure 5-1 illustrates the concept, but data blocks would not typically be shared by messages on the same queue.) The buffer can be retransmitted, if required because of errors or

timeouts, from either protocol level without replicating the data. Data block sharing is accomplished by means of a utility routine (see `dupmsg`(9F) in “Utility Descriptions” on page 352). STREAMS maintains a count of the message blocks sharing a data block in the *db_ref* field.

STREAMS provides utility routines and macros, specified in Appendix C, “STREAMS Utilities", to assist in managing messages and message queues, and to assist in other areas of module and driver development. Utility routines should always be used when operating on a message queue or accessing the message storage pool. If messages are manipulated in the queue without using the STREAMS utilities, the message ordering may become confused and lead to inconsistent results.

Note – Modules or drivers may not modify `b_next` and `b_prev`. These fields are modified by utility routines such as `putq`(9F) and `getq`(9F).

Caution – Not adhering to the DDI/DKI can result in panics and system crashes.

Sending/Receiving Messages

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

Modules and drivers do not interact directly with any system calls except `open`(2) and `close`(2). The Stream head handles all message translation and passing between user processes and STREAMS components. Message transfer between processes and the Stream head can occur in different forms. For example, `M_DATA` and `M_PROTO` messages can be transferred in their direct form by the `getmsg`(2) and `putmsg`(2) system calls. Alternatively, `write`(2) causes one or more `M_DATA` messages to be created from the data buffer

supplied in the call. M_DATA messages received at the Stream head will be consumed by read(2) and copied into the user buffer. As another example, M_SIG causes the Stream head to send a signal to a process.

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

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

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

The format of putmsg is as follows:

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

fd identifies the Stream to which the message will be passed, ctlptr and dataptr identify the control and data parts of the message, and flags may be used to specify that a high-priority message (M_PCPROTO) should be sent.

When a control part is present, setting *flags* to 0 generates an M_PROTO message. If *flags* is set to RS_HIPRI, an M_PCPROTO message is generated. Note that a *ctlptr* is translated to M_PCPROTO and a *dataptr* is translated to M_DATA.

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

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

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

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

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

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

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

5

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

putpmsg() has the following interface:

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

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

The call

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

is equivalent to the system call

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

and the call

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

is equivalent to the system call

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

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

getpmsg() has the following format:

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

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

The call

```
int band = 0;
int flags = MSG_ANY;
getpmsg(fd, ctlptr, dataptr, &band, &flags);
```

is equivalent to

```
int flags = 0;
getmsg(fd, ctlptr, dataptr, &flags);
```

If MSG_HIPRI is set and *bandp is non-zero, getpmsg() fails with EINVAL.

5

Control of Stream Head Processing

The M_SETOPTS message allows a driver or module to exercise control over certain Stream head processing. An M_SETOPTS message can be sent upstream at any time. The Stream head responds to the message by altering the processing associated with certain system calls. The options to be modified are specified by the contents of the stroptions structure (see Appendix A, "STREAMS Data Structures") contained in the message. For more information on the options available in *so_flags*, see Appendix B, "Message Types".

Six Stream head characteristics can be modified. Four characteristics correspond to fields contained in queue (min/max packet sizes and high/low watermarks). The other two are discussed here.

Read Options

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

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

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

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

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

The second set of bits, RPROTMASK, deals with the treatment of protocol messages by the read(2) system call:

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

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

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

Write Offset

If the SO_WROFF flag of *so_flags* is turned on, the framework uses the value for write offset (*so_wroff*) as a hook to allow more efficient data handling. It works as follows: In every data message generated by a write(2) system call and in the first M_DATA block of the data portion of every message generated by a putmsg(2) call, the Stream head will leave *so_wroff* bytes of space at the beginning of the message block. Expressed as a C language construct:

```
bp->b_rptr = bp->b_datap->db_base + so_wroff
```

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

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

5

Message Queues and Message Priority

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

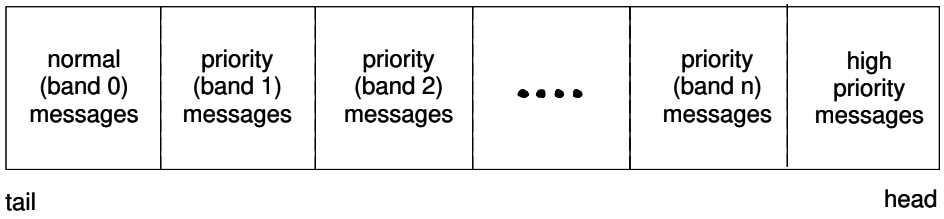

Figure 5-2 Message Ordering in a Queue

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

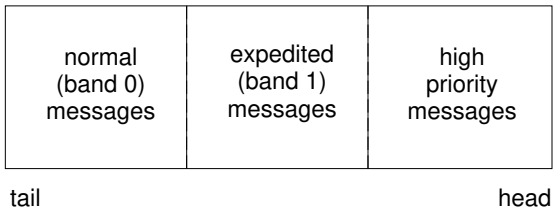

Figure 5-3 Message Ordering with One Priority Band

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

For example, a module may want to take a message off its queue, duplicate it, and put the original message back on its queue. It may then pass the new message on to the next module. If the priority band of the new message is changed somewhere else on the Stream, the original message will be out of order in the queue. Therefore, if the reference count of the message is greater than one, it is recommended that the module copy the message via `copymsg`(), free the duplicated message, and then change the priority of the copied message. The location of `b_band` is important relating to `copymsg`(). If `b_band` is in the `msgb` structure, then copying isn't necessary. If `b_band` is in `dblk`, then copying is necessary.

Note – A service procedure should never queue a high-priority message on its own queue, or else an infinite loop will result. The enqueuing will trigger the queue to be immediately scheduled again.

Several routines are provided to aid you in controlling each priority band of data flow. These routines are

- `flushband`(9F)
- `bcanputnext`(9F)
- `strqget`(9F)
- `strqset`(9F)

The `flushband`() routine is discussed in "Flush Handling" on page 153, the `bcanputnext`() routine is discussed in "Flow Control" on page 76, and the other two routines are described below. Appendix C, "STREAMS Utilities" also has a description of these routines.

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

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

The information is returned in the `long` referenced by *valp*. The fields that can be obtained are defined by the following (defined in `<sys/stream.h>`):

```
    QLOWAT                  /* q_lowat or qb_lowat */
    QMAXPSZ                 /* q_maxpsz */
    QMINPSZ                 /* q_minpsz */
    QCOUNT                  /* q_count or qb_count */
    QFIRST                  /* q_first or qb_first */
    QLAST                   /* q_last or qb_last */
    QFLAG                   /* q_flag or qb_flag */
```

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

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

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

The updated information is provided by *val*. strqset() returns 0 on success and an error number on failure. If the field is intended to be read-only, then the error EPERM is returned and the field is left unchanged. The following fields are read-only: QCOUNT, QFIRST, QLAST, and QFLAG. The use of strqget and strqset routines must be enclosed by freezestr() and unfreezestr().

The ioctls I_FLUSHBAND, I_CKBAND, I_GETBAND, I_CANPUT, and I_ATMARK support multiple bands of data flow. The ioctl I_FLUSHBAND allows a user to flush a particular band of messages. It is discussed in more detail in “Flush Handling” on page 153.

The ioctlI_CKBAND allows a user to check if a message of a given priority exists on the Stream head read queue. Its interface is:

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

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

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

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

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

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

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

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

The field `b_flag` of the `msgb` structure can have a flag `MSGMARK` that allows a module or driver to `mark` a message. This is used to support TCP's (Transmission Control Protocol) ability to indicate to the user the last byte of out-of-band data. Once `marked`, a message sent to the Stream head causes the Stream head to remember the message. A user may check to see if the message on the front of its Stream head read queue is `marked` or not with the `I_ATMARK ioctl`. If a user is reading data from the Stream head and there are multiple messages on the read queue, and one of those messages is `marked`, the `read`(2) terminates when it reaches the `marked` message and returns the data only up to that `marked` message. The rest of the data may be obtained with successive reads.

The `ioctl I_ATMARK` has the following format:

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

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

The `queue` Structure

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

The operation of a queue revolves around the `queue` structure as described in queue (9S):

```
struct qinit              *q_qinfo; /* procs and limits for queue */
struct msgb               *q_first; /* msg que head for this queue */
struct msgb               *q_last;  /* msg queue tail for this queue */
struct queue              *q_next;  /* next queue in Stream */
struct queue              *q_link   /* to next Q for scheduling */
void                      *q_ptr;   /* to module private data */
ulong                      q_count; /* number of bytes in queue */
ulong                      q_flag;  /* queue state */
long                       q_minpsz;/* min packet size accepted */
long                       q_maxpsz;/* max packet size accepted */
ulong                      q_hiwat; /* queue high watermark */
ulong                      q_lowat; /* queue low watermark */
```

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

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

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

q_count is used in flow control calculations and is the number of bytes in messages in the queue.

Using queue Information

Modules and drives can change *q_ptr* directly. Modules and drivers can read but should not change *q_qinfo*, and *q_next*. The `strqset`(9F) utility can be used to change *q_hiwat*, *q_lowat*, *q_maxpsz*, and *q_minpsz*. Modules and drivers should use `strgget`(9F) to read *q_hiwat*, *1_lowat*, *q_mazpsz*, *q_count*, *q_first*, *q_last*, or *q_flag*.

All other accesses to fields in the queue(9S) structure should be made through STREAMS utility routines (see Appendix C, “STREAMS Utilities"). Modules and drivers should not change any fields not explicitly listed above. Also modules should lock their private data structures. See Chapter 13, “Multi-Threaded STREAMS" for more information on locking.

Queue Flags

Programmers using the STREAMS mechanism should be aware of the following queue flags. See queue(9S).

QENAB	queue is enabled to run service procedure (on the run)
QWANTR	to read from the queue
QWANTW	to write to the queue
QFULL	queue is full
QREADR	set for all read queues
QUSE	queue has been allocated
QNOENB	do not enable the queue when data is placed on it

The qband Structure

The queue flow information for each band, other than band 0, is contained in a qband structure. This structure is not visible to other modules. For accessible information see strqget and strqset. qband is described in qband(9S) and is defined as follows:

```
struct qband *qb_next;              /* next band's info */
ulong        qb_count;              /* number of bytes in band */
struct msgb *qb_first;              /* beginning of band's data */
struct msgb *qb_last;               /* end of band's data */
ulong        qb_hiwat;              /* high watermark for band */
ulong        qb_lowat;              /* low watermark for band */
ulong        qb_flag;               /* flag, QB_FULL, denotes that a */
                                    /* band of data flow is flow */
                                    /* controlled */
```

This structure contains pointers to the linked list of messages in the queue. These pointers, *qb_first* and *qb_last*, denote the beginning and end of messages for the particular band. The *qb_count* field is analogous to the queue's *q_count*

field. However, *qb_count* only applies to the messages in the queue in the band of data flow represented by the corresponding `qband` structure. In contrast, *q_count* only contains information regarding normal and high-priority messages.

Each band has a separate high and low water mark, *qb_hiwat* and *qb_lowat*. These are initially set to the `queue`'s *q_hiwat* and *q_lowat* respectively. Modules and drivers may change these values if desired through the `strqset`(9F) function. Two flags, `QB_FULL` and `QB_WANTW`, are defined for *qb_flag*. `QB_FULL` denotes that the particular band is full. `QB_WANTW` indicates that someone attempted to write to the band that was flow controlled.

The `qband` structures are not preallocated per queue. Rather, they are allocated when a message with a priority greater than zero is placed in the queue via `putq`(9F), `putbq`(9F), or `insq`(9F). Since band allocation can fail, these routines return 0 on failure and 1 on success. Once a `qband` structure is allocated, it remains associated with the queue until the queue is freed. `strqset`() and `strqget`() will cause `qband` allocation to occur. Sending a message to a band will cause all bands up to and including that one to be created.

qband Flags

Programmers using the STREAMS mechanism should be aware of the following qband flags.

`QB_FULL`	band is considered full
`QB_WANTW`	to write to the queue

Using `qband` Information

The STREAMS utility routines should be used when manipulating the fields in the `queue` and `qband` structures. The routines `strqset`(9F) and `strqget`(9F) should be used to access band information.

Drivers and modules are allowed to change the *qb_hiwat* and *qb_lowat* fields of the `qband` structure.

Drivers and modules may only read the *qb_count*, *qb_first*, *qb_last*, and *qb_flag* fields of the `qband` structure.

Only the fields listed previously may be referenced at all. There are fields in the structure that are reserved and are thus not documented.

Figure 5-4 shows a queue with two extra bands of flow.

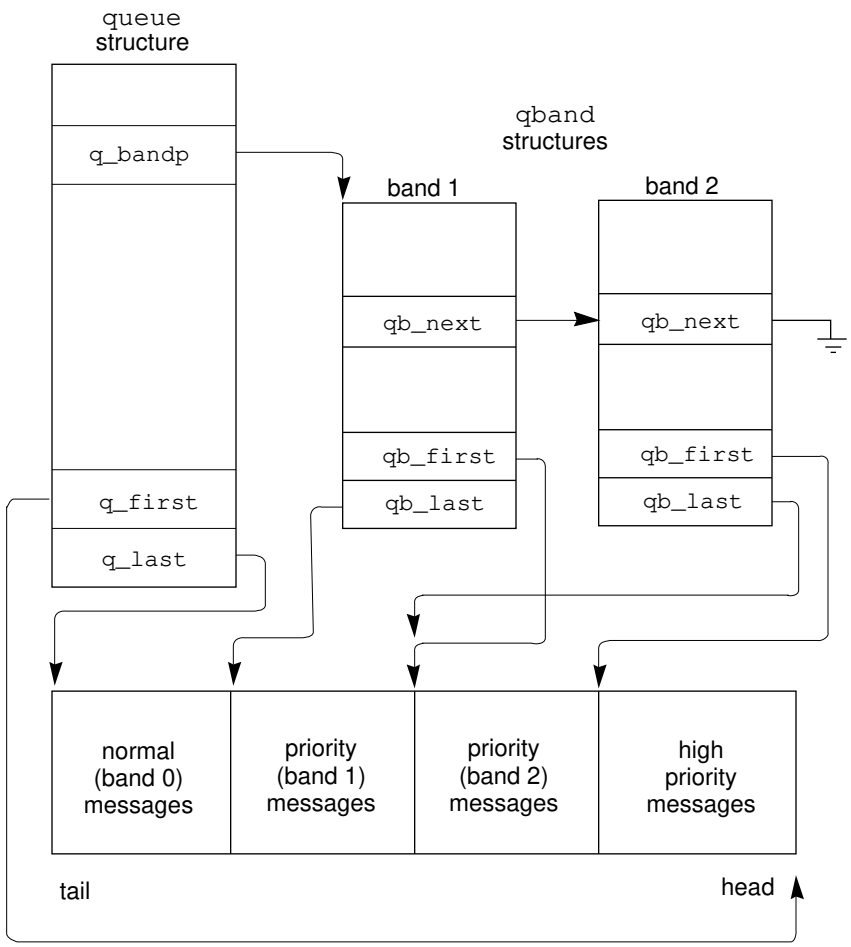

Figure 5-4 Data Structure Linkage

Message Processing

`put` procedures are generally required in pushable modules. `service` procedures are optional. If the `put` routine queues messages, there must exist a corresponding `service` routine that handles the queued messages. If the `put` routine does not queue messages, the `service` routine need not exist.

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

1. A message is received by the `put` procedure associated with queue, where some processing may be performed on the message.
2. The `put` procedure places the message in the queue by use of the `putq`() utility routine for the `service` procedure to perform further processing later.
3. `putq`() places the message in the queue based on its priority.
4. Then, `putq`() makes the queue ready for execution by the STREAMS scheduler following all other queues currently scheduled.
5. When the system goes from kernel mode to user mode, the STREAMS scheduler calls the `service` procedure.
6. The `service` procedure gets the first message (*q_first*) from the message queue by using the `getq`() utility.
7. The `service` procedure processes the message and passes it to the `put` procedure of the next queue with `putnext`().
8. The `service` procedure gets the next message and processes it. This processing continues until the queue is empty or flow control blocks further processing. The `service` procedure returns to the caller.

Caution – `A service` or `put` procedure must never block since it has no user context. It must always return to its caller.

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

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

More typically, `put` procedures will, at a minimum, process high priority messages to avoid queueing them.

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

Note – The presence of a `service` procedure is mandatory if the flow control mechanism is to be utilized by the queue. If you don't implement flow control, it is possible to overflow queues and hang the system.

Flow Control

The STREAMS flow control mechanism is voluntary and operates between the two nearest queues in a Stream containing `service` procedures (see Figure 5-5). Messages are generally held on a queue only if a `service` procedure is present in the associated queue.

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

Flow control operates as follows:

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

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

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

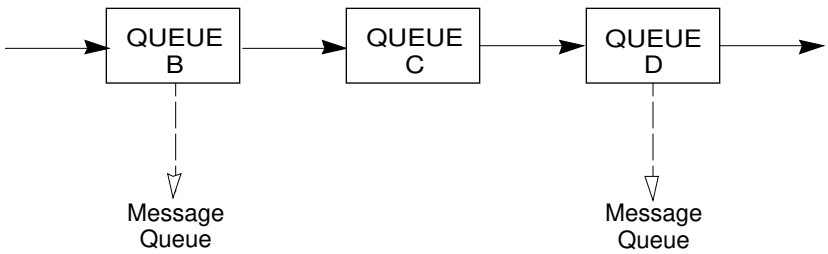

Figure 5-5 Flow Control

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

5. While B is blocked, any messages except high priority messages arriving at B will accumulate on its message queue (recall that high priority messages are not subject to flow control). Eventually, B may reach a full state and the full condition will propagate back to the preceding module in the Stream.

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

Modules and drivers need to observe the message priority. High priority messages, determined by the type of the first block in the message,

```
    (mp)->b_datap->db_type> = QPCTL
```

are not subject to flow control. They should be processed immediately and forwarded, as appropriate.

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

This is the general flow control processing of ordinary messages:

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

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

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

```
            putbq()
            return
    }
```

Expedited data have their own flow control with the same general processing as that of ordinary messages. `bcanputnext`(9F) is used to provide modules and drivers with a way to test flow control in the given priority band. It returns 1 if a message of the given priority can be placed in the queue. It returns 0 if the priority band is flow controlled. If the band does not yet exist in the queue in question, the routine returns 1.

If the band is flow controlled, the higher bands are not affected. However, the same is not true for lower bands. The lower bands are also stopped from sending messages. If this didn't take place, the possibility would exist where lower priority messages would be passed along ahead of the flow controlled higher priority ones.

The call `bcanputnext(q, 0);` is equivalent to the call `canputnext(q);`.

Note – A `service` procedure must process all messages in its queue unless flow control prevents this.

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

Note – High-priority messages are discarded only if there is already a high-priority message on the Stream head read queue. That is, there can be only one high priority message (PC_PROTO) present on the Stream head read queue at any time.

putbq() replaces messages at the beginning of the appropriate section of the message queue in accordance with their priority. This might not be the same position at which the message was retrieved by the preceding getq(). A subsequent getq() might return a different message.

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

This also applies to putbq() and insq(). If an attempt is made to insert a message out of order in a queue via insq(), the message is not inserted and the routine fails.

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

The service procedure is written using the following algorithm:

```
while ((bp = getq(q)) != NULL) {
    if (queclass (bp) == QPCTL)
            /* Process the message */
            putnext(q, bp);
    } else if (bcanputnext(q, bp->b_band)) {
            /* Process the message */
            putnext(q, bp);
    } else {
            putbq(q, bp);
            return;
    }
}
```

If the module or driver is unconcerned with priority bands, the algorithm is the same as described in the previous paragraphs, except that `canputnext(q)` is substituted for the `bcanputnex()` call.

Driver upstream flow control is explained next as an example. Although device drivers typically discard input when unable to send it to a user process, STREAMS allows driver read-side flow control, possibly for handling temporary upstream blockages. This is done through a driver-read `service` procedure which is disabled during the driver open with `noenable`(). If the driver input interrupt routine determines messages can be sent upstream (from canputnext), it sends the message with putnext(). Otherwise, it calls `putq`() to queue the message. The message waits in the message queue (possibly with queue length checked when new messages are queued by the interrupt routine) until the upstream queue becomes clear. When the blockage abates, STREAMS back-enables the driver read `service` procedure. The `service` procedure sends the messages upstream using `getq`() and `canputnext`(), as described previously. This is similar to `looprsrv`() (See “Loop-Around Driver" section of Chapter 9, “Drivers" where the `service` procedure is present only for flow control.

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

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

Note – STREAMS routines are called at different priority levels. Interrupt routines are called at the interrupt priority of the interrupting device. `Service` routines are called with interrupts enabled (hence `service` routines for STREAMS drivers can be interrupted by their own interrupt routines).

5

Service Interfaces

STREAMS provides the means to implement a service interface between any two components in a Stream, and between a user process and the topmost module in the Stream. A service interface is defined at the boundary between a service user and a service provider (see Figure 5-7). A *service interface* is a set of primitives and the rules that define a service and the allowable state transitions that result as these primitives are passed between the user and the provider. These rules are typically represented by a state machine. In STREAMS, the service user and provider are implemented in a module, driver, or user process. The primitives are carried bidirectionally between a service user and provider in `M_PROTO` and `M_PCPROTO` messages.

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

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

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

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

This section describes these system calls in the context of a service interface example. First, a brief overview of STREAMS service interfaces is presented.

Service Interface Benefits

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

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

Each STREAMS module provides a set of processing functions, or services, and an interface to those services. The service interface of a module defines the interaction between that module and any neighboring modules, and is a necessary component for providing module substitution. By creating a well-defined service interface, applications and STREAMS modules can interact with any module that supports that interface. Figure 5-6 demonstrates this.

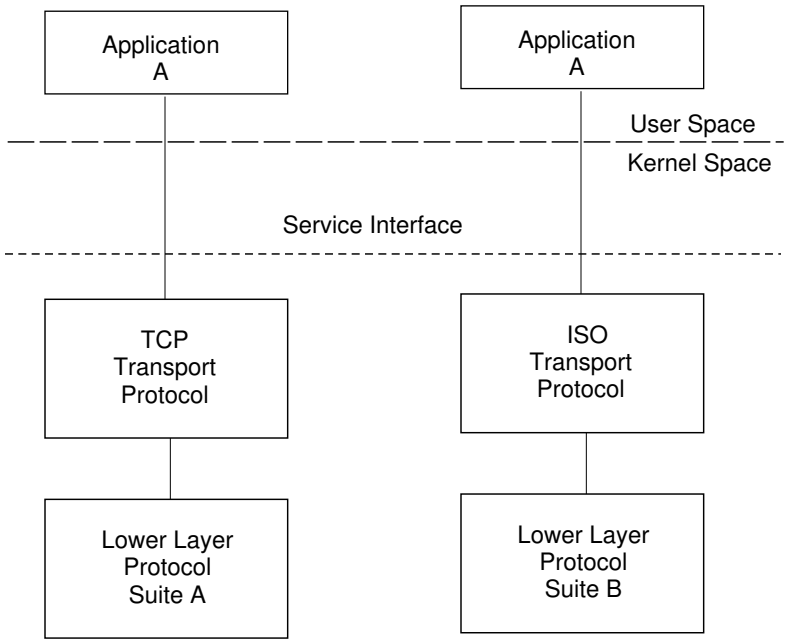

Figure 5-6 Protocol Substitution

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

The three components of any service interface are the service user, the service provider, and the service interface itself, as seen in the following figure.

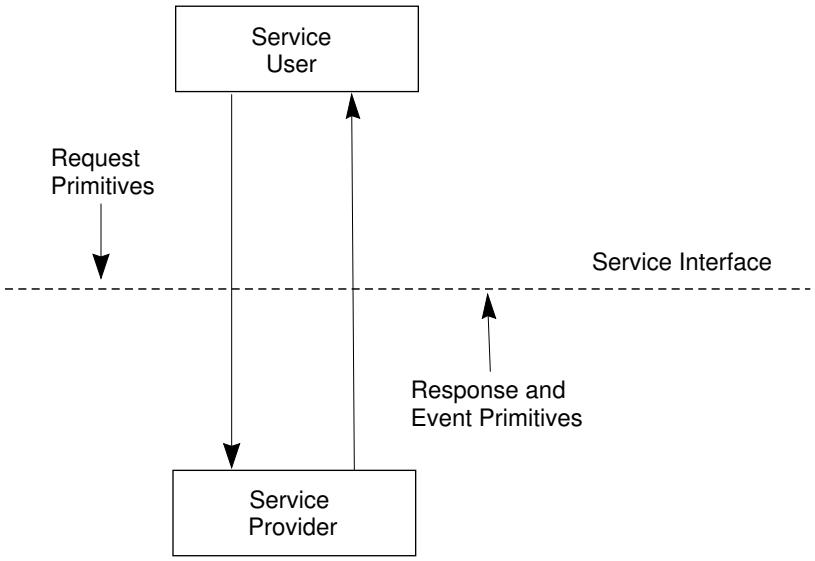

Figure 5-7 Service Interface

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

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

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

5

Service Interface Library Example

The service interface library example presented here includes four functions that enable a user to do the following:

- establish a Stream to the service provider and bind a protocol address to the Stream,
- send data to a remote user,
- receive data from a remote user, and
- close the Stream connected to the provider

First, the structure and constant definitions required by the library are shown in the following example. These typically will reside in a header file associated with the service interface.

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

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

/*
 * The following structure definitions define the format
 * of the control part of the service interface message
 * of the above primitives.
 */
struct bind_req {                       /* bind request */
    long        PRIM_type;              /* always BIND_REQ */
    long        BIND_addr;              /* addr to bind */
};
 struct unitdata_req {                  /* unitdata request */
    long        PRIM_type;              /* always UNITDATA_REQ */
    long        DEST_addr;              /* destination addr */
};

struct ok_ack {                         /* positiv acknowledgment*/
    long        PRIM_type;              /* always OK_ACK */
```

```
};

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

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

/* union of all primitives */
union primitives {
    long                                type;
    struct bind_req                     bind_req;
    struct unitdata_req                  unitdata_req;
    struct ok_ack                       ok_ack;
    struct error_ack                    error_ack;
    struct unitdata_ind                 unitdata_ind;
};

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

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

`BIND_REQ` This request asks the provider to bind a specified protocol address. It requires an acknowledgment from the provider to verify that the contents of the request were syntactically correct.

`UNITDATA_REQ` This request asks the provider to send data to the specified destination address. It does not require an acknowledgment from the provider.

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

`OK_ACK` This primitive informs the user that a previous bind request was received successfully by the service provider.

ERROR_ACK	This primitive informs the user that a non-fatal error was found in the previous bind request. It indicates that no action was taken with the primitive that caused the error.
UNITDATA_IND	This primitive indicates that data destined for the user have arrived.

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

Accessing the Service Provider

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

Code Example 5-1 inter_open

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

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

    /* send bind request msg down stream */

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

    if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {
```

Code Example 5-1 inter_open

```
            close(fd);
            return(-1);
    }
```

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

After *inter_open* sends the bind request, it must wait for an acknowledgment from the service provider, as follows:

Code Example 5-2 Service Provider Example

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

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

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

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

 /* switch on type (first long in rcvbuf) */
    switch(rcvbuf.type) {
```

Code Example 5-2 Service Provider Example

```
    default:
            close(fd);
            errno = EPROTO;
            return(-1);

    case OK_ACK:
            return(fd);

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

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

The acknowledgment primitives are defined as high priority messages. Messages are queued in a first-in-first-out manner within their priority at the Stream head; high priority messages are placed at the front of the Stream head queue followed by priority band messages and ordinary messages. The STREAMS mechanism allows only one high priority message per Stream at the Stream head at one time. Any additional high priority messages will be discarded upon reaching the Stream head. (There can be only one high priority message present on the Stream head read queue at any time.) High priority messages are particularly suitable for acknowledging service requests when the acknowledgment should be placed ahead of any other messages at the Stream head.

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

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

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

Closing the Service Provider

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

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

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

Sending Data to Service Provider

The third routine, *inter_snd*, passes data to the service provider for transmission to the user at the address specified in *addr*. The data to be transmitted are contained in the buffer pointed to by *buf* and contains *len* bytes.

On successful completion, this routine returns the number of bytes of data passed to the service provider; on failure, it returns -1 and sets *errno* to an appropriate UNIX system error value.

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

    unitdata_req.PRIM_type = UNITDATA_REQ;
    unitdata_req.DEST_addr = addr;

    ctlbuf.len = sizeof(struct unitdata_req);
    ctlbuf.buf = (char *)&unitdata_req;
    databuf.len = len;
    databuf.buf = buf;

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

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

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

Receiving Data

The final routine in this example, *inter_rcv*, retrieves the next available data. *buf* points to a buffer where the data should be stored, *len* indicates the size of that buffer, and *addr* points to a long integer where the source address of the data

will be placed. On successful completion, *inter_rcv* returns the number of bytes of retrieved data; on failure, it returns -1 and an appropriate UNIX system error value.

Figure 5-8 Receiving Data

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

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

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

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

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

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

The user's control and data buffers should be large enough to hold any incoming data. If both buffers are large enough, `getmsg` will process the data indication and return 0, indicating that a full message was retrieved successfully. However, if neither buffer is large enough, `getmsg` will only retrieve the part of the message that fits into each user buffer. The remainder of the message is saved for subsequent retrieval (if in message non-discard mode), and a positive, non-zero value is returned to the user. A return value of `MORECTL` indicates that more control information is waiting for retrieval. A return value of `MOREDATA` indicates that more data is waiting for retrieval. A return value of (`MORECTL | MOREDATA`) indicates that data from both parts of the message remain. In the example, if the user buffers are not large enough (that is, `getmsg` returns a positive, non-zero value), the function will set *errno* to EIO and fail.

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

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

Module Service Interface Example

The following example is part of a module that illustrates the concept of a service interface. The module implements a simple service interface and mirrors the service interface library example. The following rules pertain to service interfaces:

- Modules and drivers that support a service interface must act upon all `PROTO` messages and not pass them through.

- Modules may be inserted between a service user and a service provider to manipulate the data part as it passes between them. However, these modules may not alter the contents of the control part (PROTO block, first message block) nor alter the boundaries of the control or data parts. That is, the message blocks comprising the data part may be changed, but the message may not be split into separate messages nor combined with other messages.

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

Declarations

The service interface primitives are defined in the declarations:

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

 /* Primitives initiated by the service user */

#define BIND_REQ                1       /* bind request */
#define UNITDATA_REQ            2       /* unitdata request */

 /* Primitives initiated by the service provider */

#define OK_ACK                  3   /* bind acknowledgment */
#define ERROR_ACK               4   /* error acknowledgment */
#define UNITDATA_IND            5   /* unitdata indication */
/*
 * The following structures define the format of the
 * stream message block of the above primitives.

 */
struct bind_req {                   /* bind request */
    long        PRIM_type;          /* always BIND_REQ */
    long        BIND_addr;          /* addr to bind*/
};
struct unitdata_req {               /* unitdata request */
    long        PRIM_type;          /* always UNITDATA_REQ */
    long        DEST_addr;          /* dest addr */
};
struct ok_ack {                     /* ok acknowledgment */
```

```
    long        PRIM_type;              /* always OK_ACK */
};
struct error_ack {                      /* error acknowledgment */
    long        PRIM_type;              /* always ERROR_ACK */
    long        UNIX_error;             /* UNIX system error code*/
};
struct unitdata_ind {                   /* unitdata indication */
    long        PRIM_type;              /* always UNITDATA_IND */
    long        SRC_addr;               /* source addr */
};

union primitives {                      /* union ofallprimitives */
    long                                type;
    struct bind_req                     bind_req;
    struct unitdata_req                 unitdata_req;
    struct ok_ack                       ok_ack;
    struct error_ack                    error_ack;
    struct unitdata_ind                 unitdata_ind;
};
struct dgproto {                        /*structure minor device */
    short state;                        /*current provider state */
    long addr;                          /* net address */
};

/* Provider states */
#define IDLE 0
#define BOUND 1
```

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

The module recognizes two commands:

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

`UNITDATA_REQ` Send data to the specified address.

The module generates three messages:

`OK_ACK` A positive acknowledgment (ack) of `BIND_REQ`.

`ERROR_ACK` A negative acknowledgment (nak) of `BIND_REQ`.

`UNITDATA_IND` Data from the network have been received.

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

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

state	current state of the service provider IDLE or BOUND
addr	network address that has been bound to this Stream

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

Service Interface Procedure

The write put procedure is:

```
static int protowput(queue_t *q, mblk_t *mp)
{
    union primitives *proto;
    struct dgproto *dgproto;
    int err;
    dgproto = (struct dgproto *) q->q_ptr;  /* priv data struct */
    switch (mp->b_datap->db_type) {
    default:
            /* don't understand it */
            mp->b_datap->db_type = M_ERROR;
            mp->b_rptr=mp->b_wptr=mp->b_datap->db_base;
            *mp->b_wptr++ = EPROTO;
            qreply(q, mp);
            break;
    case M_FLUSH: /* standard flush handling goes here ... */
            break;
    case M_PROTO:
            /* Protocol message -> user request */
            proto = (union primitives *) mp->b_rptr;
            switch (proto->type) {
            default:
                mp->b_datap->db_type = M_ERROR;
                mp->b_rptr=mp->b_wptr=mp->b_datap->db_base;
                *mp->b_wptr++ = EPROTO;
                qreply(q, mp);
                return;
```

```
            case BIND_REQ:
                if (dgproto->state != IDLE) {
                        err = EINVAL;
                        goto error_ack;
                }
                if (mp->b_wptr - mp->b_rptr !=
                 sizeof(struct bind_req)) {
                        err = EINVAL;
                        goto error_ack;
                }
                if (err = chkaddr(proto->bind_req.BIND_addr))
                        goto error_ack;
                dgproto->state = BOUND;
                dgproto->addr = proto->bind_req.BIND_addr;
                mp->b_datap->db_type = M_PCPROTO;
                proto->type = OK_ACK;
                mp->b_wptr=mp->b_rptr+sizeof(structok_ack);
                qreply(q, mp);
                break;
            error_ack:
                mp->b_datap->db_type = M_PCPROTO;
                proto->type = ERROR_ACK;
                proto->error_ack.UNIX_error = err;
                mp->b_wptr=mp->b_rptr+sizeof(structerror_ack);
                qreply(q, mp);
                break;
            case UNITDATA_REQ:
                if (dgproto->state != BOUND)
                        goto bad;
                if (mp->b_wptr - mp->b_rptr !=
                     sizeof(struct unitdata_req))
                        goto bad;
                if(err=chkaddr(proto->unitdata_req.DEST_addr))
                        goto bad;
                putq(q, mp);
                /* start device or mux output ... */
                break;
            bad:
                freemsg(mp);
                break;
            }
        }
return(0);
}
```

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

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

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

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

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

Message Allocation and Freeing

The allocb(9F) utility routine is used to allocate a message and the space to hold the data for the message. allocb() returns a pointer to a message block containing a data buffer of at least the size requested, providing there is enough memory available. It returns null on failure. Note that allocb() always returns a message of type M_DATA. The type may then be changed if required. *b_rptr* and *b_wptr* are set to *db_base* (see msgb and datab), which is the start of the memory location for the data.

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

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

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

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

In the following example, allocb() is used by the bappend subroutine that appends a character to a message block:

```
/*
 * Append a character to a message block.
 * If (*bpp) is null, it will allocate a new block
 * Returns 0 when the message block is full, 1 otherwise
 */
#define MODBLKSZ      128        /* size of message blocks */

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

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

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

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

The next example subroutine modwput processes all the message blocks in any downstream data (type M_DATA) messages. freemsg() frees messages.

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

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

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

                        bp->b_rptr++;
                        continue;

                newblk:
                        if (nmp == NULL)
                                nmp = nbp;
                        else { /* link msg blk to tail of nmp */
                                linkb(nmp, nbp);
                                nbp = NULL;
                        }
                }
```

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

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

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

Recovering From No Buffers

The `bufcall`(9F) utility can be used to recover from an `allocb`() failure. The call syntax is as follows:

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

`bufcall`() calls (**func*)(*arg*) when a buffer of *size* bytes is available. When *func* is called, it has no user context and must return without blocking. Also, there is no guarantee that when *func* is called, a buffer will actually still be available.

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

Caution – Care must be taken to avoid deadlock when holding resources while waiting for `bufcall`() to call (**func*)(*arg*). `bufcall`() should be used sparingly.

Two examples are provided. The first is a device-receive-interrupt handler:

Code Example 5-3 Device Interrupt handler

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
int id;                /* hold id val for unbufcall */

dev_rintr(dev)
{
    /* process incoming message ... */
    /* allocate new buffer for device */
    dev_re_load(dev);
}

/*
 * Reload device with a new receive buffer
 */
dev_re_load(dev)
{
    mblk_t *bp;
    id = 0;                /* begin with no waiting for buffers */
    if ((bp = allocb(DEVBLKSZ, BPRI_MED)) == NULL) {
            cmn_err(CE_WARN,"dev:allocbfailure(size%d)\n",
                 DEVBLKSZ);
            /*
             * Allocation failed. Use bufcall to
             * schedule a call to ourselves.
             */
            id = bufcall(DEVBLKSZ,BPRI_MED,dev_re_load,dev);
            return;
    }

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

dev_rintr is called when the device has posted a receive interrupt. The code retrieves the data from the device (not shown). *dev_rintr* must then give the device another buffer to fill by a call to *dev_re_load*, which calls `allocb`(). If `allocb`() fails, *dev_re_load* uses `bufcall`() to call itself when STREAMS determines a buffer is available. *id* is saved as the return value from `bufcall`() to be used later by `unbufcall`() prior to closing the driver. This is important to be aware of as a system crash due to a callback that still has a `bufcall`() request pending is very difficult to track down. See Chapter 13, "Multi-Threaded STREAMS" for more information on the uses of `unbufcall`(). These references are protected by MT locks.

Note – Since `bufcall`() may fail, there is still a chance that the device may hang. A better strategy, in the event `bufcall`() fails, would be to discard the current input message and resubmit that buffer to the device. Losing input data is generally better than getting hung.

The second example is a write `service` procedure, *mod_wsrv*, which needs to prefix each output message with a header.

```
static int mod_wsrv(queue_t *q)
{
    extern int qenable();
    mblk_t *mp, *bp;
    while (mp = getq(q)) {
            /* check for priority messages and canput ... */

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

In this previous example, *mod_wsrv* illustrates a case for potential deadlock. If `allocb`() fails, *mod_wsrv* tends to recover without loss of data and calls `bufcall`(). In this case, the routine passed to `bufcall`() is `qenable`(). When a buffer is available, the `service` procedure will be automatically re-enabled. Before exiting, the current message is put back in the queue.This example deals with `bufcall`() failure by resorting to the `timeout`() operating system utility routine. `timeout`() will schedule the given function to be run with the given argument in the given number of clock cycles(there are 1,000,000 microseconds per second). In this example, if `bufcall`() fails, the system will run `qenable`() after two seconds have passed.

Releasing Callback Requests

When `allocb`() fails and a call is made to `bufcall`(), a callback is pending until a buffer is actually returned. Since this callback is an asynchronous process, it must be released before all processing is complete. To release this queued event, use `unbufcall`().

Pass the *id* returned from `bufcall`() to the `unbufcall`() routine. Then you can close the driver in the normal way. If this sequence of `unbufcall`() and `xxclose`() is not followed, then a situation exists where the callback can occur and the driver will be closed. This is one of the most difficult types of bugs to track down during the debugging stage.

Extended STREAMS Buffers

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

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

In a typical network arrangement, data is received from the network by the I/O card. The controller reads the block of data into the card's internal buffer. It interrupts the host computer to denote that data have arrived. The

STREAMS driver gives the controller the kernel address where the data block is to go and the number of bytes to transfer. After the controller has read the data into its buffer and verified the checksum, it copies the data into main memory to the address specified by the DMA (direct memory access) memory address. Once in the kernel space, the data is packaged into message blocks and processed in the usual manner.

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

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

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

- **Allocation** - If the drivers are to use DARAM without wasting STREAMS resources and without being dependent on upstream modules, a data and message block can be allocated without an allocated data buffer. The routine to use is called `esballoc`(9F). This returns a message block and data block without an associated STREAMS buffer. Rather, the buffer used is the one supplied by the caller in the buffer passed in.
- **Freeing** - Each driver using non-STREAMS resources in a STREAMS environment must fully manage those resources, including freeing them. However, to make this as transparent as possible, a driver-dependent routine is executed in the event `freeb`() is called to free a message and data block with an attached non-STREAMS buffer.

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

The free routine should not reference any dynamically allocated data structures that become freed when the driver is closed, as messages can exist in a Stream after the driver is closed. For example, when a Stream is closed, the driver close routine is called and its private data structure may be deallocated. If the driver sends a message created by `esballoc` upstream, that message may still be on the Stream head read queue. When the Stream head read queue is flushed, the message is freed and a call is made to the driver's free routine after the driver has been closed.

The format of the `free_rtn(9S)` structure is as follows:

```
void (*free_func)(); /*driver dependent free routine*/
char *free_arg;     /* argument for free_rtn */
```

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

The method by which `free_func` is called is implementation-specific. Do not assume that *free_func* will or will not be called directly from STREAMS utility routines like `freeb`(). The `free_func` function must not call another module's `put` procedure nor attempt to acquire a private module lock that may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and/or deadlock exists.

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

esballoc Example

This skeletal example (which will not compile) shows how extended buffers are managed in the multithreaded environment. The driver maintains a pool of special memory which is esballoc'ed. The allocator free routine uses the queue struct assigned to the driver, or some other queue private data, so the allocator and the close routine need to coordinate to ensure that no outstanding esballoc'ed memory blocks remain after the close. The special memory blocks are of type `ebm_t`, the counter is `ebm`, the mutex `mp` and the condition variable `cvp` are used to implement the coordination:

Code Example 5-4 esballoc Example

```
ebm_t *
special_new()
{
    mutex_enter(&mp);
    /*
     * allocate some special memory
     */
    esballoc();
    /*
     * increment counter
     */
    ebm++;
    mutex_exit(&mp);
}

void
special_free()
{
    mutex_enter(&mp);
    /*
     * de-allocate some special memory
     */
    freeb();

    /*
     * decrement counter
     */
    ebm--;
    if (ebm == 0)
            cv_broadcast(&cvp);
```

Code Example 5-4 esballoc Example

```
    mutex_exit(&mp);
}

open_close(q, .....)
    ....
{
    /*
     * do some stuff
     */
    /*
     * Time to decomission the special allocator.  Are there
     * any outstanding allocations from it?
     */
    mutex_enter(&mp);
    while (ebm > 0)
            cv_wait(&cvp, &mp);

    mutex_exit(&mp);}
```

5