Modules 8

Module Overview

An executing STREAMS module consists of a pair of initialized `queue` structures and a defined set of kernel-level procedures and data structures used to process data, status, and control information. A Stream may have zero or more modules. User processes push (insert) modules on a Stream using the `I_PUSH ioctl` and pop (remove) them using the `I_POP ioctl`. Pushing and popping of modules happens in a LIFO (Last-In-First-Out) fashion. Modules manipulate messages as they flow through the Stream.

Note that this differs from a module you write as a driver writer. A module you write consists of initialized `qinit` structures, where an executing module consists of initialized `queue` structures.

STREAMS Module Configuration

Like device drivers and all other kernel modules, STREAMS modules are dynamically linked and can be loaded and unloaded from the running kernel.

Note – The word *module* is used in two different ways when talking about drivers. There are STREAMS modules, which are pushable non-driver entities, and there are kernel-loadable modules, which are components of the kernel.

8

Making a module loadable involves providing appropriate module linkage information for the kernel in the form of initialized `modlstrmod`(9S) as well as several entry points, namely `_init`(9E), `_info`(9E), and `_fini`(9E).

STREAMS modules may be unloaded from the kernel when not pushed onto a Stream. A STREAMS module may "veto" unloading by having its `_fini()` routine return an error from `errno.h`. `EBUSY` is a good choice. STREAMS drivers accomplish unloading veto through the standard driver `detach` (9E) mechanism, which is unavailable to STREAMS modules.

Module Procedures

STREAMS module procedures (`open`, `close`, `put`, `service`) have already been described in the previous chapters. This section shows some examples and further describes attributes common to module `put` and `service` procedures.

A module's `put` procedure is called by the preceding module, driver, or Stream head, and always before that queue's `service` procedure. The `put` procedure should do any immediate processing (for example, high-priority messages), while the corresponding `service` procedure performs deferred processing.

The `service` procedure is used primarily for performing deferred processing, with a secondary task to implement flow control. Once the `service` procedure is enabled, it may start but not complete before running user-level code. The `put` and `service` procedures must not block because there is no thread synchronization being done.

Code Example 8-1 shows a STREAMS module read-side `put` procedure:

Code Example 8-1 Read- side `put` Procedure

```
static int
modrput(queue_t *q, mblk_t *mp)
{
    struct mod_prv *modptr;

    modptr = (struct mod_prv *) q->q_ptr;  /*state info*/

    if (mp->b_datap->db_type >= QPCTL){ /*proc pri msg*/
            putnext(q, mp);        /* and pass it on */
            return (0);
    }
```

Code Example 8-1 Read- side `put` Procedure

```
static int

    switch(mp->b_datap->db_type) {
    case M_DATA:                /* may process message data */
            putq(q, mp); /* queue msg for service procedure */
            return (0);

    case M_PROTO:           /* handle protocol control message */
            .
            .
            .

    default:
            putnext(q, mp);
            return (0);
    }
}
```

The preceding code does the following:

- A pointer to a queue defining an instance of the module and a pointer to a message are passed to the `put` procedure.
- The `put` procedure switches on the type of the message. For each message type, the `put` procedure either enqueues the message for further processing by the module `service` procedure, or passes the message to the next module in the Stream.
- High priority messages are typically processed immediately, but not required, by the `put` procedure and passed to the next module.
- Ordinary (or normal) messages are either queued or passed along the Stream.

Code Example 8-2 shows a module write-side `put` procedure:

Code Example 8-2 Write-side `put` Procedure

```
static int
modwput(queue_t *q, mblk_t *mp)
{
```

Code Example 8-2 Write-side `put` Procedure

```
    struct mod_prv *modptr;

    modptr = (struct mod_prv *) q->q_ptr; /*state info*/

    if (mp->b_datap->db_type >= QPCTL){ /* proc pri msg */
            putnext(q, mp);                  /* and pass it on */
            return (0);
    }

    switch(mp->b_datap->db_type) {
    case M_DATA:                  /* may process message data */
            putq(q, mp);/* queue msg for service procedure or */
                    /* pass message along with putnext(q,mp) */
            return (0);

    case M_PROTO:
            .
            .
            .

    case M_IOCTL:             /* if cmd in msg is recognized */
                              /* process message and send back
reply */
                              /* else pass message downstream */

    default:
            putnext(q, mp);
            return (0);
    }
}
```

The write-side `put` procedure, unlike the read side, may be passed `M_IOCTL` messages. It is up to the module to recognize and process the `ioctl` command, or pass the message downstream if it does not recognize the command.

Code Example 8-3 shows a general scenario employed by the module's `service` procedure:

Code Example 8-3 Service Procedure

```
static int
modrsrv(queue_t *q)
{
    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
            if (!(mp->b_datap->db_type >= QPCTL) &&
               !canputnext(q)) {      /* flow control check */
                putbq(q, mp);         /* return message */
                return (0);
            }
            /* process the message */
            switch(mp->b_datap->db_type) {
                .
                .
                .
                putnext(q, mp); /* pass the result */
            }
        return (0);
}
```

The steps are:

- Retrieve the first message from the queue using `getq`().
- If the message is high priority, process it immediately, and pass it along the Stream.
- Otherwise, the `service` procedure should use the `canputnext`() utility to determine if the next module or driver that enqueues messages is within acceptable flow-control limits. The `canputnext`() procedure searches the Stream for the next module, driver, or the Stream head with a `service` procedure. When it reaches one, it looks at the total message space currently being allocated at that queue for enqueued messages. If the amount of space currently used at that queue reaches the high watermark, the `canputnext`() procedure returns false (zero). If the next queue with a `service` procedure is within acceptable flow-control limits, `canputnext`() returns true (nonzero).

- If `canputnext`() returns false, the `service` procedure should return the message to its own queue using the `putbq`() procedure. The `service` procedure can do no further processing at this time, and it should return.
- If `canputnext`() returns true, the `service` procedure should complete any processing of the message. This may involve retrieving more messages from the queue,allocating and deallocating header and trailer information, and performing control function, for the module.
- When the `service` procedure is finished processing the message, it calls the `putnext`() procedure to pass the resulting message to the next queue.

These steps are repeated until there are no messages left in the queue (that is, `getq`() returns `NULL`) or `canputnext`() returns false.

Filter Module Example

The module shown next, *crmod* in Code Example 8-4,is an asymmetric filter. On the write side, newline is converted to carriage return followed by newline. On the read side, no conversion is done. The declarations of this module are essentially the same as those of the null module presented in Chapter 7, “Overview of Modules and Drivers":

Code Example 8-4 crmod

```
/* Simple filter
 * converts newline -> carriage return, newline
 */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo =
    { 0x09, "crmod", 0, INFPSZ, 512, 128 };

static int modopen (queue_t*, dev_t*, int, int, cred_t*);
static int modrput (queue_t*, mblk_t*);
static int modwput (queue_t*, mblk_t*);
static int modwsrv (queue_t*);
static int modclose (queue_t*, int, cred_t*);
```

Code Example 8-4 crmod

```
static struct qinit rinit = {
    modrput, NULL, modopen, modclose, NULL, &minfo, NULL};

static struct qinit winit = {
    modwput, modwsrv, NULL, NULL, NULL, &minfo, NULL};

struct streamtab crmdinfo={ &rinit, &winit, NULL, NULL};
```

The procedure for configuring *crmod* is shown in Appendix E, “Configuration". `stropts.h` includes definitions of flush message options common to user level modules and drivers. *modopen* and *modclose* are unchanged from the null module example shown in Chapter 7, “Overview of Modules and Drivers". *modrput* is like *modput* from the null module.

Note that, in contrast to the null module example, a single `module_info` structure is shared by the read side and write side. The `module_info` includes the flow control high and low watermarks (512 and 128) for the write queue. (Though the same `module_info` is used on the read queue side, the read side has no `service` procedure so flow control is not used.) The `qinit` contains the `service` procedure pointer.

The write side `put` procedure, the beginning of the `service` procedure, and an example of flushing a queue are shown next:

```
static int
modwput(queue_t *q, mblk_t *mp)
{
    if (mp->b_datap->db_type >= QPCTL &&
      mp->b_datap->db_type != M_FLUSH)
            putnext(q, mp);
    else
            putq(q, mp); /* Put it on the queue */
    return (0);
}
static int
modwsrv(queue_t *q)
{
    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
            switch (mp->b_datap->db_type) {
```

```
                default:
                    if (canputnext(q)) {
                            putnext(q, mp);
                            break;
                    } else {
                            putbq(q, mp);
                            return (0);
                     }

                case M_FLUSH:
                    if (*mp->b_rptr & FLUSHW)
                            flushq(q, FLUSHDATA);
                    putnext(q, mp);
                    break;
```

modwput, the write `put` procedure, switches on the message type. High priority messages that are not type `M_FLUSH` are `putnext` to avoid scheduling. The others are queued for the `service` procedure. An `M_FLUSH` message is a request to remove messages on one or both queues. It can be processed in the `put` or `service` procedure.

modwsrv is the write `service` procedure. It takes a single argument, a pointer to the write `queue`. *modwsrv* processes only one high priority message, `M_FLUSH`. No other high priority messages should reach *modwsrv*.

For an `M_FLUSH` message, *modwsrv* checks the first data byte. If `FLUSHW` is set, the write queue is flushed by use of the `flushq`() utility (see Appendix C, “STREAMS Utilities"). `flushq`() takes two arguments, the queue pointer and a flag. The flag indicates what should be flushed, data messages (FLUSHDATA) or everything (FLUSHALL). Data includes `M_DATA`, `M_DELAY`, `M_PROTO`, and `M_PCPROTO` messages. The choice of what types of messages to flush is module specific.

Ordinary messages will be returned to the queue if `canputnext`(q) returns false, indicating the downstream path is blocked. The example continues with the remaining part of *modwsrv* processing `M_DATA` messages:

```
        case M_DATA: {
            mblk_t *nbp = NULL;
            mblk_t *next;
            if (!canputnext(q)) {
                    putbq(q, mp);
                    return (0);
```

```
            }
            /* Filter data, appending to queue */
            for (; mp != NULL; mp = next) {
                    while (mp->b_rptr < mp->b_wptr) {
                            if (*mp->b_rptr == '\n')
                                if (!bappend(&nbp, '\r'))
                                        goto push;
                            if (!bappend(&nbp, *mp->b_rptr))
                                goto push;
                            mp->b_rptr++;
                            continue;
                    push:
                            if (nbp)
                                putnext(q, nbp);
                            nbp = NULL;
                            if (!canputnext(q)) {
                                if (mp->b_rptr>=mp->b_wptr){
                                        next = mp->b_cont;
                                        freeb(mp);
                                        mp=next;
                                }
                                if (mp)
                                        putbq(q, mp);
                                return (0);
                            }
                    } /* while */
                    next = mp->b_cont;
                    freeb(mp);
                    if (nbp)
                        putnext(q, nbp);
                }
            }
        }
    }
    return (0);
}
```

The differences in M_DATA processing between this and the example in Chapter 5, “Messages" in the section “Message Allocation and Freeing" relate to the manner in which the new messages are forwarded and flow controlled. For the purpose of demonstrating alternative means of processing messages, this version creates individual new messages rather than a single message containing multiple message blocks. When a new message block is full, it is immediately forwarded with the putnext() procedure rather than being

linked into a single, large message (as was done in Chapter 5, “Messages"). This alternative may not be desirable because message boundaries will be altered and because of the additional overhead of handling and scheduling multiple messages.

When the filter processing is performed (following push), flow control is checked (with `canputnext`()) after, rather than before, each new message is forwarded. This is done because there is no provision to hold the new message until the queue becomes unblocked. If the downstream path is blocked, the remaining part of the original message is returned to the queue. Otherwise, processing continues.

Flow Control

To support the STREAMS flow control mechanism, modules that use `service` procedures must invoke `canputnext`() before calling `putnext`(), and use appropriate values for the high and low watermarks. If your module has a service procedure, it is your responsibility to manage the flow control. If you don’t have a service procedure, then there is no need to do anything.

The queue hiwat and lowat values limit the amount of data that can be placed on a queue. It prevents depletion of buffers in the buffer pool. Flow control is advisory in nature and it can be bypassed. It is managed by high and low watermarks and regulated by utility routines such as `qenable`(). Module flow control is implemented by using the `canputnext`(), `getq`(), `putq`(), `putbq`(), `insq`(), `rmvq`(), and `canputnext`() procedures.

The following scenario takes place normally in flow control:

A driver sends data to a module using the `putnext`() procedure, and the module’s `put` procedure queues data using `putq`(). As a result of `putq()`, the service procedure is enabled and will execute at some indeterminate time in the future. When the `service` procedure runs, it retrieves the data by calling the `getq`() utility.

If the module cannot process data at the rate at which the driver is sending the data, the following happens:

When the message is queued, putq increments the value of q_count by the size of the message and compares the result against the modules high water limit (`q_hiwat`) value for that write queue or read queue. If the count reaches `q_hiwat`, `putq` will set the internal `FULL` indicator for the queue.

This will cause messages from upstream in the case of a write side queue or downstream in the case of a read side queue to be halted (`canputnext`() returns `FALSE`) until the queue count drops below `q_lowat`. `getq` decrements the queue count. If the resulting count is below `q_lowat`, `getq` will back-enable and cause the service procedure to be called for any queue which had been blocked.

Note – Flow control does not prevent reaching `q_hiwat` on any given queue. Flow control may exceed its maximum value before `canputnext` detects `QFULL` and flow is stopped.

The next two examples show a line discipline module's flow control. Code Example 8-5 is a read-side line discipline module and the second shows a write side line discipline module. Note that the read side is the same as the write side but without the `M_IOCTL` processing.

Code Example 8-5 Read-side Line Discipline Module

```
/* read side line discipline module flow control */
static mblk_t *read_canon(mblk_t *);

static int
ld_read_srv(
    queue_t *q)                         /* pointer to read queue */
{
    mblk_t *mp;                         /* original message */
    mblk_t *bp;                         /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
            switch (mp->b_datap->db_type) { /* type of msg */
            case M_DATA: /* data message */
                if (canputnext(q)) {
                        bp = read_canon(mp);
                        putnext(q, bp);
                } else {
                        putbq(q, mp); /* put messagebackinqueue */
                        return (0);
                }
                break;

            default:
                if (mp->b_datap->db_type >= QPCTL)
                        putnext(q, mp); /* high priority message */
```

Code Example 8-5 Read-side Line Discipline Module

```
                else { /* ordinary message */
                        if (canputnext(q))
                                putnext(q, mp);
                        else {
                                putbq(q, mp);
                                return (0);
                        }
                }
                break;
            }
    }
return (0);
}

/* write side line discipline module flow control */
static int
ld_write_srv(
    queue_t *q)                     /* pointer to write queue */
{
    mblk_t *mp;                     /* original message */
    mblk_t *bp;                     /* canonicalized message */

    while ((mp = getq(q)) != NULL) {
            switch (mp->b_datap->db_type) { /* type of msg */
            case M_DATA: /* data message */
                if (canputnext(q)) {
                        bp = write_canon(mp);
                        putnext(q, bp);
                } else {
                        putbq(q, mp);
                        return (0);
                }
                break;

            case M_IOCTL:
                ld_ioctl(q, mp);
                break;

            default:
                if (mp->b_datap->db_type >= QPCTL)
                        putnext(q, mp);/* high priority message */
                else {                  /* ordinary message */
                        if (canputnext(q))
```

Code Example 8-5 Read-side Line Discipline Module

```
                                putnext(q, mp);
                        else {
                                putbq(q, mp);
                                return (0);
                        }
                }
                break;
        }
    }
return (0);
}
```

Design Guidelines

Module developers should follow these guidelines:

- If a module does not understand the message types, the message types must be passed to the next module.
- The module that acts on an `M_IOCTL` message should send an `M_IOCACK` or `M_IOCNAK` message in response to the `ioctl`. If the module does not understand the `ioctl`, it should pass the `M_IOCTL` message to the next module.
- Modules should be designed in such way that they don't pertain to any particular driver but can be used by all drivers.
- In general, modules should not require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. This makes it easier to arbitrarily push modules on top of each other in a sensible fashion. Not following this rule may limit module reusability.
- Filter modules pushed between a service user and a service provider may not alter the contents of the `M_PROTO` or `M_PCPROTO` block in messages. The contents of the data blocks may be manipulated, but the message boundaries must be preserved.

Also see "Design Guidelines" on page 160 of Chapter 7, "Overview of Modules and Drivers".

8