Overview of Modules and Drivers 7

Module and Driver Environment

Modules and drivers are processing elements in STREAMS. A Stream device driver is similar to a conventional device driver. It is opened like a character driver and is responsible for the system interface to the device.

STREAMS modules and drivers are structurally similar. The call interfaces to driver routines are identical to interfaces used for modules. Drivers and modules must declare `streamtab`, `qinit`, and `module_info` structures. Within the STREAMS mechanism drivers are required elements, but modules are optional.

There are three significant differences between modules and drivers:

- A driver must be able to handle interrupts from a device, so the driver will include an interrupt handler routine.
- A driver may have multiple Streams connected to it.
- Drivers exist within the file system name space; you use the system call open to open them. Modules don't process interrupts and can only be pushed onto an already opened Stream.

User context is not generally available to STREAMS module procedures and drivers.

7

Caution – STREAMS driver and module `put` procedures and `service` procedures have no user context. They cannot block.

Module and Driver Declarations

A module and driver will contain, at a minimum, declarations of the following form:

Code Example 7-1 Module and Driver Declarations

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/param.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info rminfo =
     { 0x08, "mod", 0, INFPSZ, 0, 0 };
static struct module_info wminfo =
     { 0x08, "mod", 0, INFPSZ, 0, 0 };
static int modopen (queue_t *, dev_t *, int, int, cred_t *);
static int modput (queue_t *, mblk_t *);
static int modclose (queue_t*, int, cred_t*);

static struct qinit rinit = {
    modput, NULL, modopen, modclose, NULL, &rminfo, NULL };
static struct qinit winit = {
    modput, NULL, NULL, NULL, NULL, &wminfo, NULL };

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

The contents of these declarations are constructed for the null module example in this section. This module performs no processing. Its only purpose is to show linkage of a module into the system. The descriptions in this section are general to all STREAMS modules and drivers unless they specifically reference the example. For information on the data structures discussed, see the man(9S) section of SunOS 5.3 Reference Manual.

The declarations shown are: the header set; the read and write queue (*rminfo* and *wminfo*) `module_info` structures; the module open, read-put, write-put, and close procedures; the read and write (*rinit*, and *winit*) `qinit` structures; and the `streamtab` structure.

The header files, types.h and stream.h, are always required for modules and drivers. The header file, param.h, contains definitions for NULL and other values for STREAMS modules and drivers. See also *Writing Device Drivers*.

The streamtab(9S) contains qinit(9S) values for the read and write queues. The qinit structures in turn point to a module_info (9S)and an optional module_stat structure. The two required structures are:

Code Example 7-2 qinit

```
struct qinit {
    int         (*qi_putp)();  /* put procedure */
    int         (*qi_srvp)();  /* service procedure */
    int         (*qi_qopen)(); /* called on each open or push */
    int         (*qi_qclose)();/* called on last close or pop */
    int         (*qi_qadmin)(); /* reserved for future use */
    struct module_info *qi_minfo;/* information structure */
    struct module_stat *qi_mstat;/* stats struct (opt) */
};

struct module_info {
    ushort      mi_idnum;       /* module ID number */
    char        *mi_idname;     /* module name *
    long        mi_minpsz;      /* min packet size,developeruse*/
    long        mi_maxpsz;      /* max packet size,developeruse*/
    ulong       mi_hiwat;       /* hi-water mark */
    ulong       mi_lowat;       /* lo-water mark */
 };
```

The qinit structure contains the queue procedures: put, service, open, and close. All modules and drivers with the same streamtab point to the same read side and write side structure(s). The structure is meant to be software read-only, as any changes to it affect all instantiations of that module in all Streams. Pointers to the open and close procedures must be contained in the read qinit structure. These fields are ignored on the write-side. Our example has no service procedure on the read-side or write-side.

The module_info contains identification and limit values. All queues associated with a certain driver/module share the same module_info structures. The module_info structures define the characteristics of that driver/module's queues. As with the qinit, this structure is intended to be software read-only. However, the four limit values (*q_minpsz*, *q_maxpsz*, *q_hiwat*, *q_lowat*) are copied to a queue structure where they are modifiable via

strqset(). In the example, the flow control high and low water marks are zero since there is no service procedure and messages are not queued in the module.

Three names are associated with a module: the character string in fmodsw, obtained from the name of the /kernel/strmod/modname file (or alternately /usr/kernel/strmod) used to configure the module; the prefix for streamtab, used in configuring the module; and the module name field in the module_info structure. The module name must be the same as that of /kernel/strmod/modname for autoconfiguration (for example /kernel/strmod/ldterm). Each module ID and module name should be unique in the system.

Minimum and maximum packet sizes are intended to limit the total number of characters contained in M_DATA messages passed to this queue. These limits are advisory except for the Stream head. For certain system calls that write to a Stream, the Stream head will observe the packet sizes set in the write queue of the module immediately below it. Otherwise, the use of packet size is developer dependent. In the example, INFPSZ indicates unlimited size on the read-side.

The module_stat is optional. Currently, there is no STREAMS support for per-module statistical information gathering. For STREAMS framework statistics, use netstat -m.

Null Module Example

The null module procedures are as follows:

Code Example 7-3 Null Module Example

```
static int modopen(
    queue_t*q,                  /* pointer to the read queue */
    dev_t*devp,                 /* ptr to major/minor device # */
    int flag,                   /* file flags */
    int sflag,                  /* stream open flags */
    cred_t*credp)               /* ptr to a credentials struct */
{
    qprocson(q);                /* enable put/srv routines */
    return (0);                 /* return success */
}

static int modput(              /* put procedure */
    queue_t*q,                  /* pointer to the queue */
```

```
    mblk_t*mp)                  /* message pointer */
{
    putnext(q, mp);             /* pass message through */
    return (0);
}

/* Note: we only need one put procedure that can be used
 * for both read-side and write-side.
 */

static int modclose(
    queue_t*q,              /* pointer to the read queue */
    int flag,               /* file flags */
    cred_t*credp)           /* ptr to a credentials structure */
{
    qprocsoff(q);
    return (0);
}
```

The form and arguments of these procedures are the same in all modules and all drivers. Modules and drivers can be used in multiple Streams and their procedures must be reentrant.

modopen illustrates the open call arguments and return value. The arguments are the read queue pointer (*q*), the pointer (*devp*) to the major/minor device number, the file flags (*flag*, defined in open(9E), the Stream open flag (*sflag*), and a pointer to a credentials structure (*credp*). The Stream open flag can take the following values:

Table 7-1 Stream `open` flag

sflag value	definition
MODOPEN	normal module open
CLONEOPEN	clone driver open
0	normal driver open

The return value from open is 0 for success and an error number for failure. If a driver is called with the `CLONEOPEN` flag and the driver supports the clone feature, the device number pointed to by the *devp* should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major and minor device number). The open procedure for a module is called on `I_PUSH` and on all subsequent `open` calls to the same

Stream. During a push, a nonzero return value causes the `I_PUSH` to fail and the module to be removed from the Stream. If an error is returned by a module during a push, the `ioctl` fails and the Stream remains intact.

In the next example, the module open fails if not opened by the super-user. Permission checks in module and driver open routines should be done with the `drv_priv`() routine.

```
error = drv_priv(credp);
 if (error) == EPERM /* not super-user */
     return EPERM;
```

In the null module example, *modopen* enables its `put` and `srv` routines and returns successfully. *modput* illustrates the common interface to `put` procedures. The arguments are the read or write `queue` pointer, as appropriate, and the message pointer. The `put` procedure in the appropriate side of the queue is called when a message is passed from upstream or downstream. The `put` procedure has no return value. In the example, no message processing is performed. All messages are forwarded using the `putnext` function (see Appendix C). `putnext` calls the `put` procedure of the next queue in the proper direction.

The module close routine is only called on an `I_POP ioctl` or on the last `close` call of the Stream. The arguments are the read queue pointer, the file flags as in *modopen*, and a pointer to a credentials structure. The return value is 0 on success and *errno* on failure.

Module and Driver ioctls

STREAMS is a special type of character device driver that is different from the historical character input/output (I/O) mechanism. In this section, the phrases *character I/O mechanism* and *I/O mechanism* refer only to that part of the mechanism that existed before STREAMS.

The character I/O mechanism handles all `ioctl`(2) system calls transparently. That is, the kernel expects all `ioctl`s to be handled by the device driver associated with the character special file on which the call is sent. All `ioctl` calls are sent to the driver, which is expected to perform all validation and processing other than file descriptor validity checking. The operation of any specific `ioctl` is dependent on the device driver. If the driver requires data to

be transferred in from user space, it will use the kernel `ddi_copyin()` function. It may also use `ddi_copyout()` to transfer any data results to user space.

With STREAMS, there are a number of differences from the character I/O mechanism that impart `ioctl` processing.

First, there are a set of generic STREAMS `iotcl` command values (see `ioctl(2))` recognized and processed by the Stream head. These are described in `streamio(7)`. The operation of the generic STREAMS `iotcls` are generally independent of the presence of any specific module or driver on the Stream.

The second difference is the absense of user context in a module and driver when the information associated with the `ioctl` is received. This prevents use of `ddi_copyin()` or `ddi_copyout()` by the module. This also prevents the module and driver from associating any kernel data with the currently running process. (It is likely that by the time the module or driver receives the `ioctl`, the process generating it may no longer be running.)

A third difference is that for the character I/O mechanism, all ioctls are handled by the single driver associated with the file. In STREAMS, there can be multiple modules on a Stream and each one can have its own set of `ioctls`. That is, the `ioctls` that can be used on a Stream can change as modules are pushed and popped.

STREAMS provides the capability for user processes to perform control functions on specific modules and drivers in a Stream with `ioctl` calls. Most `streamio`(7) `ioctl` commands go no further than the Stream head. They are fully processed there and no related messages are sent downstream. However, certain commands and all unrecognized commands cause the Stream head to create an `M_IOCTL` message which includes the `ioctl` arguments and send the message downstream to be received and processed by a specific module or driver. The `M_IOCTL` message is the initial message type which carries `ioctl` information to modules. Other message types are used to complete the `ioctl` processing in the Stream. In general, each module must uniquely recognize and act on specific `M_IOCTL` messages.

STREAMS `ioctl` handling is equivalent to the transparent processing of the character I/O mechanism. STREAMS modules and drivers can process `ioctls` generated by applications that are implementd for a non-STREAMS environment.

7

General ioctl Processing

STREAMS blocks a user process that issues an `ioctl` and causes the Stream head to generate an `M_IOCTL` message. The process remains blocked until one of the following occurs:

- A module or a driver responds with an `M_IOCACK` (ack, positive acknowledgment) message or an `M_IOCNAK` (nak, negative acknowledgment) message
- No message is received and the request “times out”
- Rhe `ioctl` is interrupted by the user process
- An error condition occurs. For the `ioctl I_STR`, the timeout period can be a user specified interval or a default. For the other `ioctl`s, the default value (infinite) is used.

For an `I_STR`, the STREAMS module or driver that generates a positive acknowledgment message can also return data to the process in that message. An alternate means to return data is provided with transparent `ioctl`s. If the Stream head does not receive a positive or negative acknowledgment message in the specified time, the `ioctl` call fails.

A module that receives an unrecognized `M_IOCTL` message must pass it on unchanged. A driver that receives an unrecognized `M_IOCTL` must produce a negative acknowledgment.

The form of an `M_IOCTL` message is a single `M_IOCTL` message block followed by zero or more `M_DATA` blocks (see Figure B-1 in Appendix B, “Message Types"). The `M_IOCTL` message block contains an `iocblk`(9S) structure.

```
struct iocblk {
    int         ioc_cmd;              /* ioctls command type */
    cred_t      *ioc_cr;              /* full credentials */
    uint        ioc_id;               /* ioctl id */
    uint        ioc_count;            /* byte cnt in data field */
    int         ioc_error;            /* error code */
    int         ioc_rval;             /* return value */
};
```

For an `I_STR ioctl`, *ioc_cmd* contains the command supplied by the user in the `strioctl` structure defined in `streamio`(7). For others, it is the value of the `cmd` argument in the call to `ioctl()`.

If a module or driver determines an M_IOCTL message is in error for any reason, it must produce the negative acknowledgment message. This is done by setting the message type to M_IOCNAK and sending the message upstream. No data or a return value can be sent to a user in this case. If *ioc_error* is set to 0, the Stream head will cause the ioctl call to fail with EINVAL. The driver has the option of setting *ioc_error* to an alternate error number if desired.

Note – *ioc_error* can be set to a nonzero value in both M_IOCACK and M_IOCNAK. This will cause that value to be returned as an error number to the process that sent the ioctl.

If a module looks at what ioctls of other modules are doing, the module should not search for a specific M_IOCTL on the write-side but look for M_IOCACK or M_IOCNAK on the read-side. For example, the module sees TCSETA (see termio(7)) failing and searches for what is being set. The module should look at it and save away the answer but not use it. The read-side processing knows that the module is waiting for an answer for the ioctl. When the read-side processing sees an "ack" or "nak" next time, it checks if it is the same ioctl (here TCSETA) and if it is, the module may use the answer previously saved.

The two STREAMS ioctl mechanisms, I_STR and transparent, are described next. (Here, I_STR means the streamio(7) I_STR command and implies the related STREAMS processing unless noted otherwise.) I_STR has a restricted format and restricted addressing for transferring ioctl-related data between user and kernel space. It requires only a single pair of messages to complete ioctl processing. The transparent mechanism is more general and has almost no restrictions on ioctl data format and addressing. The transparent mechanism generally requires that multiple pairs of messages be exchanged between the Stream head and module to complete the processing.

This is a rather simplistic view. There is nothing preventing a given ioctl from being issued either directly (transparent) or by means of I_STR. Furthermore, ioctls issued through I_STR potentially can require further processing of the form typically associated with transparent ioctls.

7

I_STR ioctl Processing

The `I_STR ioctl` provides a capability for user applications to perform module and driver control functions on STREAMS files. `I_STR` allows an application to specify the `ioctl` timeout. It encourages all user `ioctl` data (to be received by the destination module) be placed in a single block that is pointed to from the user `strioctl` structure. The module can also return data to this block.

If the module is looking at, for example, the `TCSETA/TCGETA` group of `ioctl` calls as they pass up or down a Stream, it must never assume that because `TCSETA` comes down that it actually has a data buffer attached to it. The user may have formed `TCSETA` as an `I_STR` call and accidentally given a null data buffer pointer. One must always check *b_cont* to see if it is `NULL` before using it as an index to the data block that goes with `M_IOCTL` messages.

The `TCGETA` call, if formed as an `I_STR` call with a data buffer pointer set to a value by the user, will always have a data buffer attached to *b_cont* from the main message block. If one assumes that the data block is not there and allocates a new buffer and assigns *b_cont* to point at it, the original buffer will be lost. Thus, before assuming that the `ioctl` message does not have a buffer attached, one should check first.

The following example, Code Example 7-4, illustrates processing associated with an `I_STR ioctl`. *lpdoioctl* is called to process trapped `M_IOCTL` messages:

Code Example 7-4 I_STR ioctl

```
static void
lpdoioctl(
    struct lp *lp,
    mblk_t *mp)
{
    struct iocblk *iocp;
    queue_t *q;

    q = lp->qptr;

    /* 1st block contains iocblk structure */
    iocp = (struct iocblk *)mp->b_rptr;

    switch (iocp->ioc_cmd) {
    case SET_OPTIONS:
```

```
            /* Count should be exactly one short's worth
             * (for this example)
             */
            if (iocp->ioc_count != sizeof(short))
                goto iocnak;
            if (mp->b_cont == NULL)
                goto lognak; /* not shown in this example */
            /* Actual data is in 2nd message block */
            lpsetopt(lp, *(short *)mp->b_cont->b_rptr);

            /* ACK the ioctl */
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
    default:
    iocnak:
            /* NAK the ioctl */
            mp->b_datap->db_type = M_IOCNAK;
            qreply(q, mp);
    }
}
```

lpdoioctl illustrates driver `M_IOCTL` processing which also applies to modules. However, at case *default*, a module would not "nak" an unrecognized command, but would pass the message on. In this example, only one command is recognized, `SET_OPTIONS`. *ioc_count* contains the number of user-supplied data bytes. For this example, it must equal the size of a short. The user data is sent directly to the printer interface using *lpsetopt*. Next, the `M_IOCTL` message is changed to type `M_IOCACK` and the *ioc_count* field is set to zero to indicate that no data is to be returned to the user. Finally, the message is sent upstream using `qreply`(). If *ioc_count* was left nonzero, the Stream head would copy that many bytes from the second - Nth message blocks into the user buffer. You must set *ioc* count if you want to pass any data back to the user.

Transparent ioctl Processing

The transparent STREAMS `ioctl` mechanism allows application programs to perform module and driver control functions with `ioctl`s other than `I_STR`. It is intended to transparently support applications developed prior to the introduction of STREAMS. It alleviates the need to recode and recompile the

user level software to run over STREAMS files. More importantly, it relieves applications of the burden of packaging their ioctl requests into the form demanded by `I_STR`.

The mechanism extends the data transfer capability for STREAMS `ioctl` calls beyond that provided in the `I_STR` form. Modules and drivers can transfer data between their kernel space and user space in any `ioctl` which has a value of the *command* argument not defined in `streamio`(7). These `ioctl`s are known as transparent `ioctl`s to differentiate them from the `I_STR` form. Transparent processing support is necessary when existing user level applications perform `ioctl`s on a non-STREAMS character device and the device driver is converted to STREAMS. The `ioctl` data can be in any format mutually understood by the user application and module.

The transparent mechanism also supports STREAMS applications that send `ioctl` data to a driver or module in a single call, where the data may not be in a form readily embedded in a single user block. For example, the data may be contained in nested structures, and different user space buffers, for instance.

This mechanism is needed because user context does not exist in modules and drivers when `ioctl` processing occurs. This prevents them from using the kernel `ddi_copyin`()/`ddi_copyout`() functions. For example, consider the following `ioctl` call:

```
    ioctl (stream_filedes, user_command, &ioctl_struct);
```

where *ioctl_struct* is a structure whose members are:

```
struct ioctl_struct {
    int         stringlen;
    char        *string;
    struct other_struct*other1;
};
```

To read (or write) the elements of *ioctl_struct*, a module would have to cause a series of `ddi_copyin`()/`ddi_copyout`() calls at the stream head, using pointer information from a prior `ddi_copyin`() to transfer additional data. A non-STREAMS character driver could directly execute these copy functions because user context exists during all system calls to the driver. However, in STREAMS, user context is only available to modules and drivers in their open and close routines.

The transparent mechanism enables modules and drivers to request that the Stream head perform a ddi_copyin() or ddi_copyout() on their behalf to transfer ioctl data between their kernel space and various user space locations. The related data is sent in message pairs exchanged between the Stream head and the module. A pair of messages is required so that each transfer can be acknowledged. In addition to M_IOCTL, M_IOCACK, and M_IOCNAK messages, the transparent mechanism also uses M_COPYIN, M_COPYOUT, and M_IOCDATA messages.

The general processing by which a module or a driver reads data from user space for the transparent case involves pairs of request/response messages, as follows:

1. The Stream head does not recognize the *command* argument of an ioctl call and creates a transparent M_IOCTL message (the iocblk structure has a TRANSPARENT indicator, see *"Transparent ioctl Messages")* containing the value of the *arg* argument in the call. It sends the M_IOCTL message downstream.

2. A module receives the M_IOCTL message, recognizes the *ioc_cmd*, and determines that it is TRANSPARENT.

3. If the module requires user data, it creates an M_COPYIN message to request a copyin() of user data. The message will contain the address of user data to copy in and how much data to transfer. It sends the message upstream.

4. The Stream head receives the M_COPYIN message and uses the contents to copyin() the data from user space into an M_IOCDATA response message that it sends downstream. The message also contains an indicator of whether the data transfer succeeded.

5. The module receives the M_IOCDATA message and processes its contents.

   The module may use the message contents to generate another M_COPYIN. Steps 3 through 5 may be repeated until the module has requested and received all the user data to be transferred.

6. When the module completes its data transfer, it performs the ioctl processing and sends an M_IOCACK message upstream to notify the Stream head that ioctl processing has successfully completed.

Writing data from a module to user space is similar except that the module uses an M_COPYOUT message to request the Stream head to write data into user space. In addition to length and user address, the message includes the data to be copied out. In this case, the M_IOCDATA response will not contain user data, only an indication of success or failure.

The module may mix M_COPYIN and M_COPYOUT messages in any order. However, each message must be sent one at a time; the module must receive the associated M_IOCDATA response before any subsequent M_COPYIN/M_COPYOUT request or "ack/nak" message is sent upstream. After the last M_COPYIN/M_COPYOUT message, the module must send an M_IOCACK message (or M_IOCNAK in the event of a detected error condition).

Caution – For a transparent M_IOCTL, user data can not be returned with an M_IOCACK message. The data must have been sent with a preceding M_COPYOUT message.

Transparent ioctl Messages

The form of the M_IOCTL message generated by the Stream head for a transparent ioctl is a single M_IOCTL message block followed by one M_DATA block. The form of the iocblk structure in the M_IOCTL block is the same as described under "General ioctl Processing".. However, *ioc_cmd* is set to the value of the *command* argument in the ioctl system call and *ioc_count* is set to TRANSPARENT. TRANSPARENT distinguishes the case where an I_STR ioctl may specify a value of *ioc_cmd* equivalent to the *command* argument of a transparent ioctl. The M_DATA block of the message contains the value of the *arg* parameter in the call.

Caution – Modules that process a specific *ioc_cmd* which did not validate the *ioc_count* field of the M_IOCTL message will break if transparent ioctls with the same command are performed from user space.

M_COPYIN, M_COPYOUT, and M_IOCDATA messages and their use are described in more detail in Appendix B, "Message Types".

Transparent ioctl Examples

Following are three examples of transparent `ioctl` processing. The first illustrates `M_COPYIN`. The second illustrates `M_COPYOUT`. The third is a more complex example showing state transitions combining both `M_COPYIN` and `M_COPYOUT`.

M_COPYIN Example

In this example, the contents of a user buffer are to be transferred into the kernel as part of an `ioctl` call of the form

```
ioctl(fd, SET_ADDR, (caddr_t) &bufadd);
```

where `bufadd` is a structure of type *struct address* whose elements are:

```
struct address {
    intad_len;                  /* buffer length in bytes */
    caddr_tad_addr;             /* buffer address */
};
```

This requires two pairs of messages (request/response) following receipt of the `M_IOCTL` message. The first will `copyin` the structure and the second will `copyin` the buffer. This example illustrates processing that supports only the transparent form of `ioctl`. *xxwput* is the write-side `put` procedure for module or driver *xx*:

```
struct address {                /* same members as in user space */
    int     ad_len;             /* length in bytes */
    caddr_t ad_addr;            /* buffer address */
};

 /* state values (overloaded in private field) */
#define GETSTRUCT 0             /* address structure */
#define GETADDR 1               /* byte string from ad_addr */

static void xxioc(queue_t *q, mblk_t *mp);

static int
xxwput(q, mp)
    queue_t *q;                 /* write queue */
    mblk_t *mp;
 {
```

```
    struct iocblk *iocbp;
    struct copyreq *cqp;

switch (mp->b_datap->db_type) {
            .
            .
            .
    case M_IOCTL:
            iocbp = (struct iocblk *)mp->b_rptr;
            switch (iocbp->ioc_cmd) {
                /* do non-transparent processing.
                 */


                /*Reuse M_IOCTL block for M_COPYIN request*/
            case SET_ADDR:
                cqp = (struct copyreq *)mp->b_rptr;

                /* Get user space structure address from
                 * linked M_DATA block */


                cqp->cq_addr = (caddr_t) *(long *)mp->b_cont->b_rptr;
                freemsg(mp->b_cont);/*MUST free linked blks*/
                mp->b_cont = NULL;
                /* to identify response */
                cqp->cq_private = (mblk_t *)GETSTRUCT;

                /* Finish describing M_COPYIN message */

                cqp->cq_size = sizeof(struct address);
                cqp->cq_flag = 0;
                mp->b_datap->db_type = M_COPYIN;
                mp->b_wptr=mp->b_rptr+sizeof(struct copyreq);
                qreply(q, mp);
                break;

            default: /* M_IOCTL not for us */
                /* if module, pass on */
                /* if driver, nak ioctl */
                break;
            } /* switch (iocbp->ioc_cmd) */
            break;
    case M_IOCDATA:
            /* all M_IOCDATA processing done here */
```

```
            xxioc(q, mp);
            break;
    }
    return (0);
}
```

xxwput verifies that the SET_ADDR is TRANSPARENT to avoid confusion with an I_STR ioctl, which uses a value of *ioc_cmd* equivalent to the command argument of a transparent ioctl. When sending an M_IOCNAK, freeing the linked M_DATA block is not mandatory as the Stream head will free it. However, this returns the block to the buffer pool more quickly.

In this and all following examples in this section, the message blocks are reused to avoid the overhead of releasing and allocating, this is standard practice.

Note – The Stream head will guarantee that the size of the message block containing an iocblk structure will be large enough also to hold the copyreq and copyresp structures.

---

cq_private is set to contain state information for ioctl processing (this identifies what the subsequent M_IOCDATA response message contains). Keeping the state in the message makes the message self-describing and simplifies the ioctl processing. M_IOCDATA processing is done in *xxioc*. Two M_IOCDATA types are processed, *GETSTRUCT* and *GETADDR*:

```
xxioc(queue_t *q, mblk_t *mp)          /* M_IOCDATA processing */
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    struct address *ap;

    csp = (struct copyresp *)mp->b_rptr;
    iocbp = (struct iocblk *)mp->b_rptr;

    /* validate this M_IOCDATA is for this module */
    switch (csp->cp_cmd) {

    case SET_ADDR:
            if (csp->cp_rval){ /*GETSTRUCT or GETADDRfail*/
                freemsg(mp);
```

```
            return;
        }
        switch ((int)csp->cp_private){ /*determine state*/

        case GETSTRUCT:    /* user structure has arrived */
            /* reuse M_IOCDATA block */
            mp->b_datap->db_type = M_COPYIN;
            cqp = (struct copyreq *)mp->b_rptr;
            /* user structure */
            ap = (struct address *)mp->b_cont->b_rptr;
            /* buffer length */
            cqp->cq_size = ap->ad_len;
            /* user space buffer address */
            cqp->cq_addr = ap->ad_addr;
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            cqp->cq_flag = 0;
            cqp->cp_private=(mblk_t *)GETADDR;  /*nxt st*/
            qreply(q, mp);
            break;

        case GETADDR:               /* user address is here */
            /* hypothetical routine */
            if (xx_set_addr(mp->b_cont) == FAILURE) {
                    mp->b_datap->db_type = M_IOCNAK;
                    iocbp->ioc_error = EIO;
            } else {
                    mp->b_datap->db_type=M_IOCACK;/*success*/
                    /* may have been overwritten */
                    iocbp->ioc_error = 0;
                    iocbp->ioc_count = 0;
                    iocbp->ioc_rval = 0;
            }
            mp->b_wptr=mp->b_rptr + sizeof (struct ioclk);
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            qreply(q, mp);
            break;
        default: /* invalid state: can't happen */
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            mp->b_datap->db_type = M_IOCNAK;
            mp->b_wptr = mp->rptr + sizeof(struct iocblk);
            /* may have been overwritten */
```

```
                iocbp->ioc_error = EINVAL;
                qreply(q, mp);
                break;
            }
            break;                  /* switch (cp_private) */

    default: /* M_IOCDATA not for us */
            /* if module, pass message on */
            /* if driver, free message */
            break;
        }   /* switch (cp_cmd) */
}
```

xx_set_addr is a routine (not shown in the example) that processes the user address from the `ioctl`. Since the message block has been reused, the fields that the Stream head will examine (denoted by "may have been overwritten") must be cleared before sending an `M_IOCNAK`.

M_COPYOUT Example

In this example, the user wants option values for this Stream device to be placed into the user's *options* structure (see beginning of example code, below). This can be accomplished by use of a transparent `ioctl` call of the form

```
    ioctl(fd, GET_OPTIONS,(caddr_t) &optadd)
```

or, alternately, by use of a `I_STR` call

```
    ioctl(fd, I_STR, (caddr_t) &opts_strioctl)
```

In the first case, *optadd* is declared *struct options*. In the `I_STR` case, *opts_strioctl* is declared *struct strioctl* where *opts_strioctl.ic_dp* points to the user *options* structure.

This example illustrates support of both the `I_STR` and transparent forms of an `ioctl`. The transparent form requires a single `M_COPYOUT` message following receipt of the M_IOCTL to `copyout` the contents of the structure. *xxwput* is the write-side `put` procedure for module or driver *xx*:

```
struct options {                /* same members as in user space */
    int         op_one;
    int         op_two;
    short       op_three;
    long        op_four;
};

static int
xxwput(
    queue_t *q,                     /* write queue */
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct copyresp *csp;
    int transparent = 0;

    switch (mp->b_datap->db_type) {
            .
            .
            .
    case M_IOCTL:
            iocbp = (struct iocblk *)mp->b_rptr;
            switch (iocbp->ioc_cmd) {

            case GET_OPTIONS:
                if (iocbp->ioc_count == TRANSPARENT) {
                        transparent = 1;
                        cqp = (struct copyreq *)mp->b_rptr;
                        cqp->cq_size = sizeof(struct options);
                /* Get struct address from linked M_DATA block */
                        cqp->cq_addr = (caddr_t)
                         *(long *)mp->b_cont->b_rptr;
                        cqp->cq_flag = 0;

                        /* No state necessary - we will only ever
                         * get one M_IOCDATA from the Stream head
                         * indicating success or failure for
                         * the copyout */
                }
```

```
                if (mp->b_cont)
                    freemsg(mp->b_cont);/*over written below*/
                if ((mp->b_cont=allocb(sizeof(struct options),
                         BPRI_MED)) == NULL) {
                        mp->b_datap->db_type = M_IOCNAK;
                        iocbp->ioc_error = EAGAIN;
                        qreply(q, mp);
                        break;
                }
                /* hypothetical routine */
                xx_get_options(mp->b_cont);
                if (transparent) {
                        mp->b_datap->db_type = M_COPYOUT;
                        mp->b_wptr = mp->b_rptr +
                         sizeof(struct copyreq);
                } else {
                        mp->b_datap->db_type = M_IOCACK;
                        iocbp->ioc_count = sizeof(struct options);
                }
                qreply(q, mp);
                break;

            default: /* M_IOCTL not for us */
                /*if module, pass on;if driver, nak ioctl*/

                break;
            } /* switch (iocbp->ioc_cmd) */
            break;

    case M_IOCDATA:
            csp = (struct copyresp *)mp->b_rptr;
            /* M_IOCDATA not for us */
            if (csp->cmd != GET_OPTIONS) {
                /*if module/pass on, if driver/free message*/

                 break;
            }
            if ( csp->cp_rval ) {
                freemsg(mp);/* failure */
                return (0);
            }
            /* Data successfully copied out, ack */

            /* reuse M_IOCDATA for ack */
            mp->b_datap->db_type = M_IOCACK;
```

```
            mp->b_wptr = mp->b_rptr + sizeof(struct iocblk);
            /* may have been overwritten */
            iocbp->ioc_error = 0;
            iocbp->ioc_count = 0;
            iocbp->ioc_rval = 0;
            qreply(q, mp);
            break;
            .
            .
            .
    } /* switch (mp->b_datap->db_type) */
    return (0);
```

Bidirectional Transfer Example

This example illustrates bidirectional data transfer between the kernel and user space during transparent `ioctl` processing. It also shows how more complex state information can be used.

The user wants to send and receive data from user buffers as part of a transparent `ioctl` call of the form

```
    ioctl(fd, XX_IOCTL, (caddr_t) &addr_xxdata)
```

The user *addr_xxdata* structure defining the buffers is declared as *struct xxdata,* shown below. This requires three pairs of messages following receipt of the `M_IOCTL` message: the first to `copyin` the structure; the second to `copyin` one user buffer; and the last to `copyout` the second user buffer. *xxwput* is the write-side `put` procedure for module or driver *xx*:

```
struct xxdata {                 /* same members in user space */
    int                x_inlen; /* number of bytes copied in */
    caddr_t            x_inaddr;/* buf addr of data copied in */
    int                x_outlen;/* number of bytes copied out */
    caddr_t            x_outaddr;/* buf addr of data copied out */
};
/* State information for ioctl processing */
struct state {
    int                        st_state;      /* see below */
    struct xxdata              st_data;       /* see above */
};
/* state values */

#define GETSTRUCT                   0   /* get xxdata structure */
```

```
#define GETINDATA                   1   /*get data from x_inaddr */
#define PUTOUTDATA          2   /* get response from M_COPYOUT */

static void xxioc(queue_t *q, mblk_t *mp);

static int
xxwput(
    queue_t *q,                         /* write queue */
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
    struct state *stp;
    mblk_t *tmp;

    switch (mp->b_datap->db_type) {
            .
            .
            .
    case M_IOCTL:
            iocbp = (struct iocblk *)mp->b_rptr;
            switch (iocbp->ioc_cmd) {
            case XX_IOCTL:
                /* do non-transparent processing. (See I_STR ioctl
                 * processing discussed in previous section.)
                 */
                /*Reuse M_IOCTL block for M_COPYIN request*/

                cqp = (struct copyreq *)mp->b_rptr;

                /* Get structure's user address from
                 * linked M_DATA block */

                cqp->cq_addr = (caddr_t)
                 *(long *)mp->b_cont->b_rptr;
                freemsg(mp->b_cont);
                mp->b_cont = NULL;

                /* Allocate state buffer */

                if ((tmp = allocb(sizeof(struct state),
                 BPRI_MED)) == NULL) {
                        mp->b_datap->db_type = M_IOCNAK;
                        iocbp->ioc_error = EAGAIN;
                        qreply(q, mp);
```

```
                        break;
                }
                tmp->b_wptr += sizeof(struct state);
                stp = (struct state *)tmp->b_rptr;
                stp->st_state = GETSTRUCT;
                cqp->cq_private = tmp;

                /* Finish describing M_COPYIN message */

                cqp->cq_size = sizeof(struct xxdata);
                cqp->cq_flag = 0;
                mp->b_datap->db_type = M_COPYIN;
                mp->b_wptr=mp->b_rptr+sizeof(struct copyreq);
                qreply(q, mp);
                break;

            default: /* M_IOCTL not for us */
                /* if module, pass on */
                /* if driver, nak ioctl */
                break;

            } /* switch (iocbp->ioc_cmd) */
            break;

    case M_IOCDATA:
            xxioc(q, mp);/*all M_IOCDATA processing here*/
            break;
            .
            .
            .
    } /* switch (mp->b_datap->db_type) */
}
```

xxwput allocates a message block to contain the state structure and reuses the `M_IOCTL` to create an `M_COPYIN` message to read in the *xxdata* structure.

M_IOCDATA processing is done in *xxioc*:

```
xxioc(                                  /* M_IOCDATA processing */
    queue_t *q,
    mblk_t *mp)
{
    struct iocblk *iocbp;
    struct copyreq *cqp;
```

```
struct copyresp *csp;
struct state *stp;
mblk_t *xx_indata();

csp = (struct copyresp *)mp->b_rptr;
iocbp = (struct iocblk *)mp->b_rptr;
switch (csp->cp_cmd) {

case XX_IOCTL:
        if (csp->cp_rval) { /* failure */
            if (csp->cp_private) /* state structure */
                    freemsg(csp->cp_private);
            freemsg(mp);
            return;
         }
        stp = (struct state *)csp->cp_private->b_rptr;
        switch (stp->st_state) {

        case GETSTRUCT:    /* xxdata structure copied in */
                /* save structure */

            stp->st_data =
             *(struct xxdata *)mp->b_cont->b_rptr;
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
            /* Reuse M_IOCDATA to copyin data */
            mp->b_datap->db_type = M_COPYIN;
            cqp = (struct copyreq *)mp->b_rptr;
            cqp->cq_size = stp->st_data.x_inlen;
            cqp->cq_addr = stp->st_data.x_inaddr;
            cqp->cq_flag = 0;
            stp->st_state = GETINDATA; /* next state */
            qreply(q, mp);
            break;

        case GETINDATA: /* data successfully copied in */
            /* Process input, return output */
            if ((mp->b_cont = xx_indata(mp->b_cont))
             == NULL) { /* hypothetical */
                       /* fail xx_indata */
                       mp->b_datap->db_type = M_IOCNAK;
                       mp->b_wptr = mp->b_rptr +
                           sizeof(struct iocblk);
                   iocbp->ioc_error = EIO;
```

```
                        qreply(q, mp);
                        break;
                }
                mp->b_datap->db_type = M_COPYOUT;
                cqp = (struct copyreq *)mp->b_rptr;
                cqp->cq_size = min(msgdsize(mp->b_cont),
                 stp->st_data.x_outlen);
                cqp->cq_addr = stp->st_data.x_outaddr;
                cqp->cq_flag = 0;
                stp->st_state = PUTOUTDATA; /* next state */
                qreply(q, mp);
                break;
            case PUTOUTDATA: /* data copied out, ack ioctl */
                freemsg(csp->cp_private); /*state structure*/
                mp->b_datap->db_type = M_IOCACK;
                mp->b_wtpr=mp->b_rptr + sizeof (struct iocblk);
                /* may have been overwritten */
                iocbp->ioc_error = 0;
                iocbp->ioc_count = 0;
                iocbp->ioc_rval = 0;
                qreply(q, mp);
                break;

            default: /* invalid state: can't happen */
                freemsg(mp->b_cont);
                mp->b_cont = NULL;
                mp->b_datap->db_type = M_IOCNAK;
                mp->b_wptr=mp->b_rptr + sizeof (struct iocblk);
                iocbp->ioc_error = EINVAL;
                qreply(q, mp);
                break;
            } /* switch (stp->st_state) */
            break;
    default: /* M_IOCDATA not for us */
            /* if module, pass message on */
            /* if driver, free message */
            break;
    } /* switch (csp->cp_cmd) */
}
```

At `case GETSTRUCT`, the user *xx*`data` structure is copied into the module's `state` structure (pointed at by *cp_private* in the message) and the `M_IOCDATA` message is reused to create a second `M_COPYIN` message to read the user data. At `case GETINDATA`, the input user data is processed by the *xx_indata* routine

(not supplied in the example), which frees the linked M_DATA block and returns the output data message block. The `M_IOCDATA` message is reused to create an `M_COPYOUT` message to write the user data. At `case PUTOUTDATA`, the message block containing the state structure is freed and an acknowledgment is sent upstream.

Care must be taken at the "can't happen" *default* case since the message block containing the state structure (*cp_private*) is not returned to the pool because it might not be valid. This might result in a lost block. The ASSERT will help find errors in the module if a "can't happen" condition occurs.

I_LIST ioctl

The `ioctl I_LIST` supports the `strconf` and `strchg` commands (see `strchg`(1)) that are used to query or change the configuration of a Stream. Only the super-user or an owner of a STREAMS device may alter the configuration of that Stream.

The `strchg` command does the following:

- Pushes one or more modules on the Stream.
- Pops the topmost module off the Stream.
- Pops all the modules off the Stream.
- Pops all modules up to but not including a specified module.

The `strconf` command does the following:

- Indicates if the specified module is present on the Stream.
- Prints the topmost module of the Stream.
- Prints a list of all modules and topmost driver on the Stream. If the Stream contains a multiplexing driver, the `strchg` and `strconf` commands will not recognize any modules below that driver.

The `ioctl I_LIST` performs two functions. When the third argument of the `ioctl` call is set to `NULL`, the return value of the call indicates the number of modules, including the driver, present on the Stream. For example, if there are two modules above the driver, 3 is returned. On failure, *errno* may be set to a value specified in `streamio`(7). The second function of the `I_LIST ioctl` is

to copy the module names found on the Stream to the user supplied buffer. The address of the buffer in user space and the size of the buffer are passed to the `ioctl` through a structure `str_list` that is defined as:

```
struct str_mlist {
    char l_name[FMNAMESZ+1]; /*space for holding a module name*/
};
struct str_list {
    int sl_nmods;   /*#of modules for which space is allocated*/
    struct str_mlist *sl_modlist;/*addr of buf for names*/
};
```

Here *sl_nmods* is the number of modules in the *sl_modlist* array that the user has allocated. Each element in the array must be at least `FMNAMESZ+1` bytes long. The array is `FMNAMESZ+1` so the extra byte can hold the null character at the end of the string. `FMNAMESZ` is defined by <`sys/conf.h`>.

The user can find out how much space to allocate by first calling the `ioctl` `I_LIST` with *arg* set to `NULL`. The `I_LIST` call with *arg* pointing to the `str_list` structure returns the number of entries that have been filled into the *sl_modlist* array (the number includes the number of modules including the driver). If there is not enough space in the *sl_modlist* array (see note) or *sl_nmods* is less than 1, the `I_LIST` call will fail and *errno* is set to EINVAL. If *arg* or the *sl_modlist* array points outside the allocated address space, `EFAULT` is returned.

Note – It is possible that another module was pushed on the Stream after the user invoked the `I_LIST` `ioctl` with the `NULL` argument and before the `I_LIST` `ioctl` with the structure argument was invoked.

Flush Handling

All modules and drivers are expected to handle `M_FLUSH` messages. An `M_FLUSH` message can originate at the Stream head or from a module or a driver. The first byte of the `M_FLUSH` message is an option flag that can have following values:

`FLUSHR` Flush read queue.

`FLUSHW` Flush write queue.

`FLUSHRW` Flush both, read and write, queues.

`FLUSHBAND` Flush a specified priority band only.

The next two figures further demonstrate flushing the entire Stream due to a line break. Figure 7-1 shows the flushing of the write-side of a Stream, and Figure 7-2 shows the flushing of the read-side of a Stream. In the figures dotted boxes indicate flushed queues.

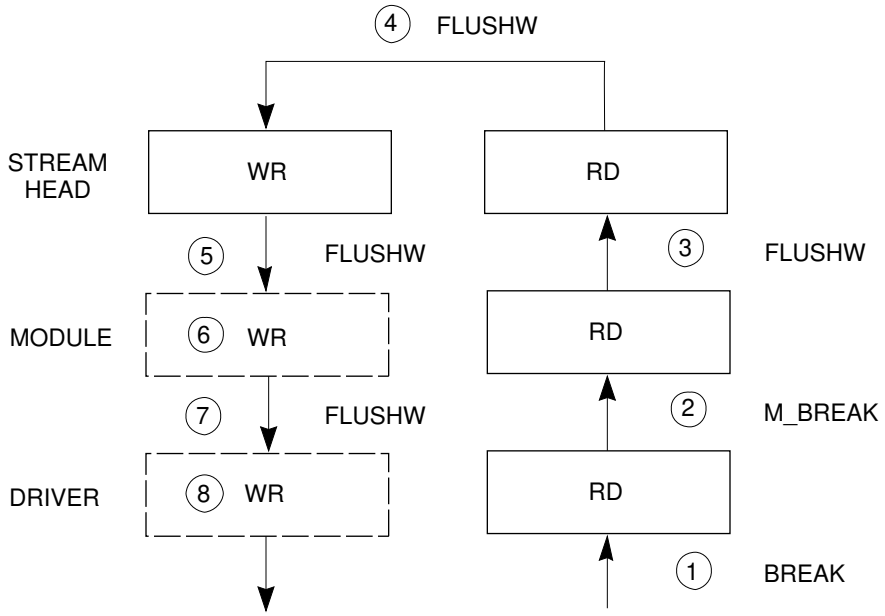

Figure 7-1 Flushing The Write-Side of A Stream

The following takes place (dotted lines mean flushed queues):

1. A break is detected by a driver.
2. The driver generates an `M_BREAK` message and sends it upstream.
3. The module translates the `M_BREAK` into an `M_FLUSH` message with `FLUSHW` set and sends it upstream.
4. The Stream head does *not* flush the write queue (no messages are ever queued there).
5. The Stream head turns the message around (sends it down the write-side).
6. The module flushes its write queue.
7. The message is passed downstream.
8. The driver flushes its write queue and frees the message.

This figure shows flushing read-side of a Stream.

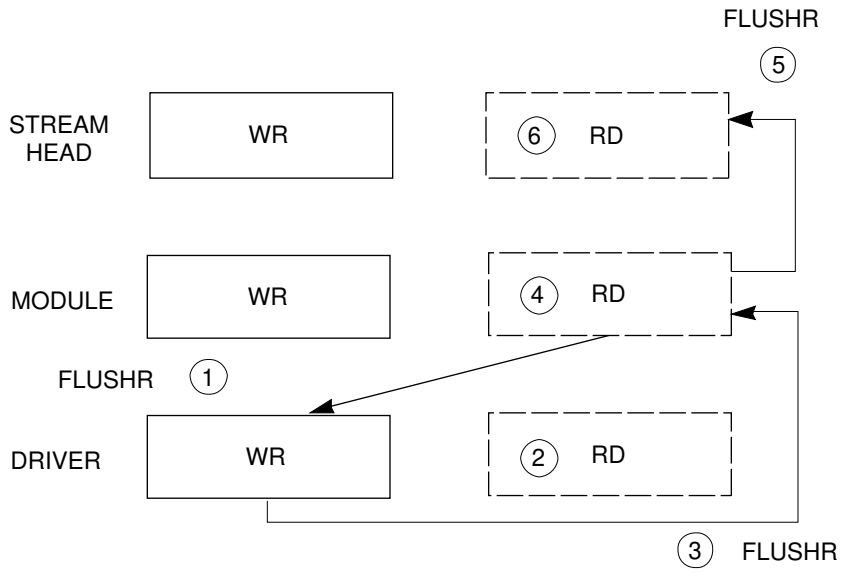

Figure 7-2 Flushing The Read-Side of A Stream

The events taking place are:

1. After generating the first M_FLUSH message, the module generates an M_FLUSH with FLUSHR set and sends it downstream.
2. The driver flushes its read queue.
3. The driver turns the message around (sends it up the read-side).
4. The module flushes its read queue.
5. The message is passed upstream.
6. The Stream head flushes the read queue and frees the message.

The following example shows line discipline module flush handling:

```
static int
ld_put(
    queue_t *q,            /* pointer to read/write queue */
    mlkb_t *mp)            /* pointer to message being passed */
{
    switch (mp->b_datap->db_type) {
        default:
            putq(q, mp); /* queue everything */
            return (0);         /* except flush */

        case M_FLUSH:
            if (*mp->b_rptr & FLUSHW)         /* flush write q */
                    flushq(WR(q), FLUSHDATA);

            if (*mp->b_rptr & FLUSHR)         /* flush read q */
                    flushq(RD(q), FLUSHDATA);

            putnext(q, mp);                   /* pass it on */
            return(0);
    }
}
```

The Stream head turns around the M_FLUSH message if FLUSHW is set (FLUSHR will be cleared). A driver turns around M_FLUSH if FLUSHR is set (should mask off FLUSHW).

7

Flushing Priority Bands

The `flushband`() routine (see Appendix C, "STREAMS Utilities") provides the module and driver with the capability to flush messages associated with a given priority band. A user can flush a particular band of messages by issuing:

```
ioctl(fd, I_FLUSHBAND, bandp);
```

where *bandp* is a pointer to a structure `bandinfo` that has a format:

```
struct bandinfo {
    unsigned char             bi_pri;
    int                       bi_flag;
};
```

The *bi_flag* field may be one of `FLUSHR`, `FLUSHW`, or `FLUSHRW`.

The following example shows flushing according to the priority band:

```
queue_t *rdq;                 /* read queue */
queue_t *wrq;                 /* write queue */

    case M_FLUSH:
        if (*bp->b_rptr & FLUSHBAND) {
            if (*bp->b_rptr & FLUSHW)
                flushband(wrq, FLUSHDATA, *(bp->b_rptr + 1));
            if (*bp->b_rptr & FLUSHR)
                flushband(rdq, FLUSHDATA, *(bp->b_rptr + 1));
        } else {
            if (*bp->b_rptr & FLUSHW)
                flushq(wrq, FLUSHDATA);
            if (*bp->b_rptr & FLUSHR)
                flushq(rdq, FLUSHDATA);
        }
        /*
         * modules pass the message on;
         * drivers shut off FLUSHW and loop the message
         * up the read-side if FLUSHR is set; otherwise,
         * drivers free the message.
         */
        break;
```

Note that modules and drivers are not required to treat messages as flowing in separate bands. Modules and drivers can view the queue having only two bands of flow, normal and high priority. However, the latter alternative will flush the entire queue whenever an M_FLUSH message is received.

One use of the field *b_flag* of the `msgb` structure is provided to give the Stream head a way to stop `M_FLUSH` messages from being reflected forever when the Stream is being used as a pipe. When the Stream head receives an `M_FLUSH` message, it sets the `MSGNOLOOP` flag in the *b_flag* field before reflecting the message down the write-side of the Stream. If the Stream head receives an `M_FLUSH` message with this flag set, the message is freed rather than reflected.

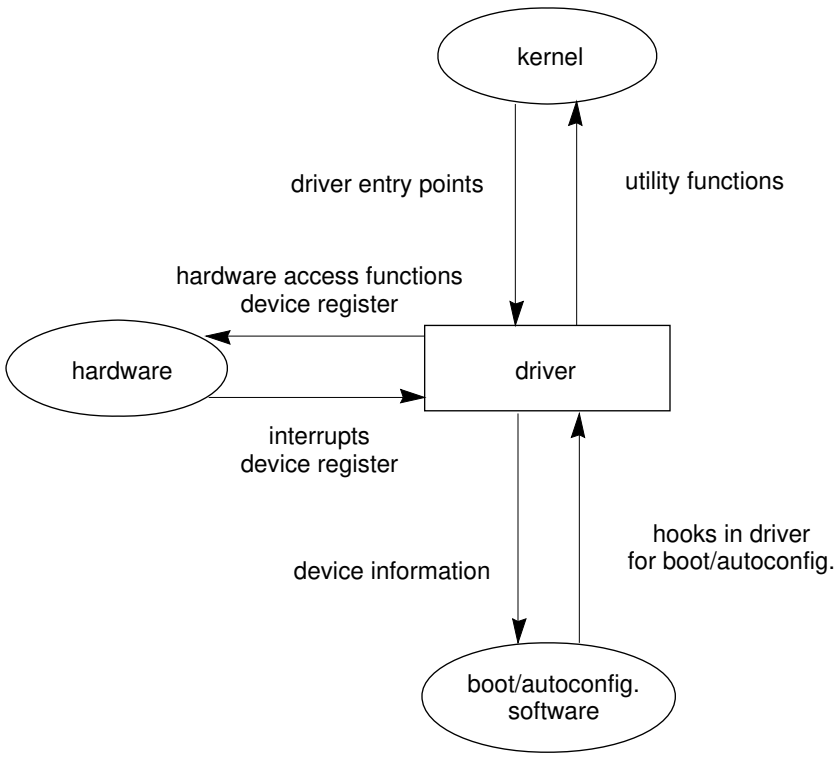

Figure 7-3 Interfaces Affecting Drivers

7

The set of STREAMS utilities available to drivers are listed in Appendix C. No system-defined macros that manipulate global kernel data or introduce structure-size dependencies are permitted in these utilities. Therefore, some utilities that have been implemented as *macros* in the prior Solaris system releases are implemented as *functions* in SunOS 5.x. This does not preclude the existence of both *macro* and *function* versions of these utilities. It is intended that driver source code will include a header file that picks up *function* declarations while the core operating system source includes a header file that defines the *macros*. With the DKI interface the following STREAMS utilities are implemented as C programming language functions: `datamsg`, `OTHERQ`, `putnext`, `RD`, and `WR`.

Replacing *macros* such as `RD`() with *function* equivalents in the driver source code allows driver objects to be insulated from changes in the data structures and their size, further increasing the useful lifetime of driver source code and objects. Multithreaded drivers are also protected against changes in implementation-specific STREAMS synchronization.

The DKI interface defines an interface suitable for drivers and there is no need for drivers to access global kernel data structures directly. The kernel functions `drv_getparm` and `drv_setparm` are provided for reading and writing information in these structures. This restriction has an important consequence. Since drivers are not permitted to access global kernel data structures directly, changes in the contents/offsets of information within these structures will not break objects. The `drv_getparm`(9f) and `drv_setparm`(9f) functions are described in more detail in the appropriate sections of the *man Pages(9F): DDI and DKI Kernel Functions* Manual.

Device Driver Interface and Driver–Kernel Interface

The Device Driver Interface (DDI) is a SunOS 5.3 interface that facilitates driver portability across different Solaris versions on the SPARC hardware. The Driver–Kernel Interface (DKI) is an interface that also facilitates driver source code portability across implementations of SVR4 on all machines. DKI driver code, however, will have to be recompiled on the machine on which it is to run.

The most important distinction between the DDI and the DKI lies in scope. The DDI addresses vendor specific architecture interfaces for block, character, and STREAMS interface drivers and modules. For more information see *Writing Device Drivers*.

STREAMS Interface

The entry points from the kernel into STREAMS drivers and modules are through the `qinit` structures (see Appendix A, “STREAMS Data Structures") pointed to by the `streamtab` structure, *prefix*info. STREAMS drivers may need to define additional entry points to support the interface with boot/autoconfiguration software and the hardware (for example, an interrupt handler).

Here is a simple incomplete example of a driver header. For the complete version see Appendix E, “Configuration", which has both data structures and entry points. If the STREAMS module has prefix *mod* then the declaration is of the form:

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

static int modwput(queue_t*, mblk_t*);
static int modwsrv(queue_t *);

static struct qinit rdinit =
       {modrput, modrsrv, modopen, modclose, NULL, NULL, NULL};
static struct qinit wrinit =
       {modwput, modwsrv, NULL, NULL, NULL, NULL, NULL};
struct streamtab modinfo = {&rdinit, &wrinit, NULL, NULL};
```

where

- *modrput* is the module’s read queue `put` procedure
- *modrsrv* is the module’s read queue `service` procedure
- *modopen* is the `open` routine for the module
- *modclose* is the `close` routine for the module
- *modwput* is the `put` procedure for the module’s write queue, and
- *modwsrv* is the `service` procedure for the module’s write queue

Each `qinit` structure can point to four entry points. (An additional function pointer has been reserved for future use and must not be used by drivers or modules.) These four function pointer fields in the `qinit` structure are: *qi_putp*, *qi_srvp*, *qi_qopen*, and *qi_close*.

7

The utility functions that can be called by STREAMS drivers and modules are listed in Appendix C. They must follow the call and return syntaxes specified in the appendix. Manual pages relating to the Driver–Kernel Interface and Device Driver Interface are provided in *man Pages(9F): DDI and DKI Kernel Functions* the System for STREAMS Drivers and Modules.

Configuring the System for STREAMS Drivers and Modules

STREAMS drivers and modules must be kernel-loadable modules. Creating a kernel-loadable module involves including several specific data structures and entry points in the driver or module. See *Writing Device Drivers* for general information on creating kernel-loadable modules. See Appendix E, “Configuration" and Chapter 8, “Modules” and Chapter 9, “Drivers” for specific configuration information for STREAMS modules and drivers.

Design Guidelines

This section summarizes guidelines common to the design of STREAMS modules and drivers. See Chapter 8, “Modules" and Chapter 9, “Drivers" for additional rules pertaining to modules and drivers.

Rules for Modules and Drivers

Below are some rules for Modules and Drivers

1. Modules and drivers are not associated with any process, and therefore have no concept of process or user context, except during open and close routines (see “Rules for Open/Close Routines").
2. Every module and driver must process an `M_FLUSH` message according to the value of the argument passed in the message.
3. A module or a driver should not change the contents of a data block whose reference count is greater than 1 (see `dupmsg`() in Appendix C) because other modules/drivers that have references to the block may not want the data changed. To avoid problems, data should be copied to a new block and then changed in the new one.
4. Modules and drivers should manipulate queues and manage buffers only with the routines provided for that purpose, (see Appendix C).

5. Modules and drivers should not require the data in an M_DATA message to follow a particular format, such as a specific alignment.

6. Care must be taken when modules are mixed and matched, because one module may place different semantics on the priority bands than another module. The specific use of each band by a module should be included in the service interface specification.

   When designing modules and drivers that make use of priority bands one should keep in mind that priority bands merely provide a way to impose an ordering of messages on a queue. The priority band is not used to determine the service primitive. Instead, the service interface should rely on the data contained in the message to determine the service primitive.

7. Drivers must NAK all unrecognized M_IOCTL messages.

8. Drivers must silently discard unrecognized message types.

9. Modules must forward all unrecognized message types.

Rules for Open/Close Routines

Here are some rules for Open/Close Routines

1. open and close routines must use condition variables to access the functionality that was provided before by sleep.

2. The open routine should return zero on success or an error number on failure. If the open routine is called with the CLONEOPEN flag, the device number should be set by the driver to an unused device number accessible to that driver. This should be an entire device number (major/minor).

3. open and close routines have user context.

4. If a module or a driver wants to allocate a controlling terminal, it should send an M_SETOPTS message to the Stream head with the SO_ISTTY flag set. Otherwise signaling will not work on the Stream.

5. A driver or module must call qprocson to enable its put and service routines and qprocsoff to disable them.

Rules for ioctls

Here are some rules for *ioctls*

- Do not change the *ioc_id*, *ioc_uid*, *ioc_gid*, or *ioc_cmd* fields in an `M_IOCTL` message.
- These rules also apply to fields in an `M_IOCDATA`, `M_COPYIN`, and `M_COPYOUT` message. (Field names are different; see Appendix A, “STREAMS Data Structures" for more information.)
- Always validate *ioc_count* to see whether the `ioctl` is the transparent or `I_STR` form.

Rules for Put and Service Procedures

To ensure proper data flow between modules and drivers, the following rules should be observed in `put` and `service` procedures:

- `Put` procedures process messages immediately; `service` procedure processing is deferred.
- `Put` and `service` procedures must not sleep.
- Return codes can be sent with STREAMS messages `M_IOCACK`, `M_IOCNAK`, and `M_ERROR`.
- Protect data structures common to `put` and `service` procedures by using `(mutex)` routines. or perimeter (see the “MT STREAMS perimeters" section of Chapter 13, “Multi-Threaded STREAMS").
- `Put` and `service` procedures cannot access the information in the `uarea` of a process.
- Processing `M_DATA` messages by both `put` and `service` procedures could lead to messages going out of sequence or causing race conditions. The `put` procedure should check if any messages were queued before processing the current message. On the read-side, it is suggested that you have the `put` procedure check if the `service` procedure is running, to avoid the possibility of a race condition. That is, if there are unprotected sections in the `service` procedure, the `put` procedure can be called and run to completion while the `service` procedure is running (the `put` procedure can interrupt the `service` procedure on the read-side). For example, the `service` procedure is running and it removes the last message from the queue, but before it puts the message upstream the `put` procedure is called

(for example,from an interrupt routine) at an unprotected section in the `service` procedure. The `put` procedure sees that the queue is empty and processes the message. The `put` procedure then returns and the `service` procedure resumes; but at this point data is out of order because the `put` procedure sent upstream the message that was received after the data the `service` procedure was processing.

Put Procedures

1. Each queue must define a `put` procedure in its `qinit` structure for passing messages between modules.

2. A `put` procedure must use the `putq`() (see Appendix C, “STREAMS Utilities" for more information) utility to queue a message on its own queue. This is necessary to ensure that the various fields of the `queue` structure are maintained consistently.

3. When passing messages to a neighboring module, a module may not call `putq`() directly, but must call its neighbor module’s `put` procedure (see `putnext`() in Appendix C).

   However, the `q_qinfo` structure that points to a module’s `put` procedure may point to `putq`() (for example, `putq`() is used as the `put` procedure for that module). When a module calls a neighbor module’s `put` procedure that is defined in this manner, it will be calling `putq`() indirectly. If any module uses `putq`() as its `put` procedure in this manner, the module must define a `service` procedure. Otherwise, no messages will ever be processed by the next module. Also, because `putq`() does not process `M_FLUSH` messages, any module that uses `putq`() as its `put` procedure must define a `service` procedure to process `M_FLUSH` messages.

4. Do not do a `putnext`() to a queue you don’t control. Only `putq()` on your own queue or one you do control. The only entry point into another queue is via the STREAMS framework.

Service Procedures

1. If flow control is desired, a `service` procedure is required. The `canputnext`() or `bcanputnext`() routines should be used by `service` procedures before doing `putnext`() to honor flow control.

2. The `service` procedure must use `getq`() to obtain a message from its message queue, so that the flow control mechanism is maintained.

3. The `service` procedure should process all messages on its queue. The only exception is if the Stream ahead is blocked (for example, `canputnext`() fails) or some other failure like buffer allocation failure. Adherence to this rule is the only guarantee that STREAMS will enable (schedule for execution) the `service` procedure when necessary, and that the flow control mechanism will not fail.

   If a `service` procedure exits for other reasons, it must take explicit steps to assure it will be re-enabled.

4. Basic service procedure scheduling involves `qenable`() and `backenable`(). This assures that no messages are lost.

5. The `service` procedure should not put a high priority message back on the queue, because of the possibility of getting into an infinite loop.

6. The `service` procedure must follow the steps below for each message that it processes. STREAMS flow control relies on strict adherence to these steps:

   a. Remove the next message from the queue using `getq`(). It is possible that the `service` procedure could be called when no messages exist on the queue, so the `service` procedure should never assume that there is a message on its queue. If there is no message, return.

   b. If all of the following conditions are met:

   - `canputnext`() or `bcanputnext`() fails and
   - the message type is not a high priority type and
   - the message is to be put on the next queue, continue at Step c. Otherwise, continue at Step d.

   c. The message must be replaced on the head of the queue from which it was removed using `putbq`(). Following this, the `service` procedure is exited. The `service` procedure should not be re-enabled at this point. It will be automatically back-enabled by flow control.

   d. If all of the conditions of Step b are not met, the message should not be returned to the queue. It should be processed as necessary. Then, return to Step a.

Data Structures

Only the contents of *q_ptr, q_minpsz, q_maxpsz, q_hiwat*, and *q_lowat* in the `queue` structure may be altered. *q_minpsz*, *q_maxpsz*, *q_hiwat*, and *q_lowat* are set when the module or driver is opened, but they may be modified subsequently via the `strqset`() utility.

Drivers and modules should not change any fields in the `equeue` structure. The only field of the `equeue` structure they are allowed to reference is *eq_bandp*.

Drivers and modules are allowed to change the *qb_hiwat* and *qb_lowat* fields of the `qband` structure via `strqset`(). They may only read the *qb_count*, *qb_first*, *qb_last*, and *qb_flag* fields.

The routines `strqget`() and `strqset`() must be used to get and set the fields associated with the queue. They insulate modules and drivers from changes in the `queue` structure and also enforce the previous rules.

Dynamic Allocation of STREAMS Data Structures

Previous releases of STREAMS statically configured data structures to support a fixed number of Streams, read and write queues, message and data blocks, link block data structures, and Stream event cells. The only way to change this configuration was to reconfigure and reboot the system. Resources were also wasted because data structures were allocated but not necessarily needed.

In SunOS 5.x, STREAMS mechanisms dynamically allocate the following data structures: `stdata`, `queue`, `linkblk`, `strevent`, `datab`, and `msgb`. STREAMS allocates memory to cover these structures as needed.

Dynamic data structure allocation has the advantage of the kernel being initially smaller than a system with static configuration. The performance of the system may also improve because of better memory utilization and added flexibility.

7