Drivers 9

Device Drivers

This chapter describes the operation of a STREAMS driver and some of the processing typically required in drivers.

In SunOS 5.x, there are differences between STREAMS drivers and non-STREAMS driver. Though STREAMS drivers can be considered a subset of device drivers in general, only STREAMS-specific information is presented here. For more information on global driver issues and non-STREAMS drivers, see *Writing Device Drivers*.

Overview of Drivers

A device driver is software that provides an interface between the operating system and a device. The driver controls the device in response to requests from the kernel. These requests are issued through the entry points. The driver provides and manages a path for the data to and from the hardware device, and services interrupts issued by the device controller. In STREAMS, drivers are *opened* and modules are *pushed*.

In SunOS 5.x, there are three types of device drivers:

1. **Hardware Driver**
   This type of driver only communicates with a specific piece of hardware. Given the variety of hardware peripherals available, these drivers have many functions.

2. **Pseudo Driver**
   This is configured, installed and acts like a hardware driver, only it does not talk to any hardware.

3. **Multiplexer Driver**
   This is a regular STREAMS driver but has multiple Streams connected to it instead of just one Stream. Multiple connections occur when more than one minor device of the same driver is in use. See Chapter 10, “Multiplexing" for more information.

Unlike a module, a device driver typically has an interrupt routine so that it is accessible from a hardware interrupt as well as from the Stream, unless it is a pseudo driver or a multiplexer driver. However, these particular differences are not recognized by the STREAMS mechanism. They are handled by developer-provided code included in the driver procedures.

The STREAMS framework supports a CLONEOPEN facility. If a STREAMS device driver chooses to support CLONEOPEN, it may be referred to as a cloneable device.

Driver Classification

In general, drivers are grouped according to the *type* of the device they control, the *access* method (the way data is transferred), and the *interface* between the driver and the device.

The type can be hardware or software. A hardware driver controls a physical device, such as a disk. A software driver, also called a pseudo driver, controls software, which in turn may interface with a hardware device. The software driver may also support pseudo devices that have no associated physical device.

Writing a Driver

General Programming

Writing a driver differs from writing other C programs in the following ways:

- A driver does not have a `main` routine. Rather, driver entry points are given specific names and accessed in a variety of ways.
- A driver functions as a part of the kernel. Consequently, a poorly written driver can degrade system performance or corrupt the system.

- A driver cannot use system calls or the C library, because the driver functions at a lower level.
- A driver cannot use floating point arithmetic.
- A driver cannot use archives or shared libraries, but frequently used subroutines can be put in separate files in the source code directory for the driver.

Driver Programming

The following lists rules of driver development:

- Drivers must have `attach`(9E), `probe`(9E) and `identify`(9E) entry points to initialize the driver. The `attach` routine initializes the driver. Software drivers will usually have little to initialize, because there is no hardware involved.
- Drivers will have `open` and `close` routines.
- Most drivers will have an interrupt handler routine. The driver developer is responsible for supplying an interrupt routine for the device's driver. In addition to hardware interrupts, the system also supports software interrupts. A software interrupt is generated by calling `ddi_trigger_softintr`(9F).
- All minor nodes are generated by the routine `ddi_create_minor_node`(9F).

Entry Points

Here are the five entry points through which you can access the driver code:

1. Kernel dynamic loading
   These are the routines that allow the kernel to find the driver in the file system and load it into or unload it from the running kernel. These include `_init`, `_fini`, and `_info`.

2. Initialization entry points
   These routines are accessed through the `dev_ops` data structure during system initialization. They include `getinfo`(9E), `identify`(9E), `probe`(9E), `attach`(9E), and `detach`(9E).

3. Table driven entry points
   These routines are accessed through `cb_ops`, the character and block access tables, when the appropriate system call is issued. The `cb_ops` table contains a pointer to the streamtab structure.

4. STREAMS queue processing entry points
   These routines are pointed to by the streamtab and read and process the messages that travel through the queue structures. They include `put`, `srv`, `open`, and `close`.

5. Interrupt routines
   These are routines to handle the interrupts for the drivers. They are registered with the `ddi_add_intr`(9F) when the kernel configuration software calls `attach`(). This loads the `ddi_add_intr` routine, which has a pointer to the interrupt handler.

STREAMS Drivers

STREAMS Driver Configuration

As with other SunOS 5.x drivers, STREAMS drivers are dynamically linked, allowing them to be loadable. All drivers are dynamically loaded when referenced for the first time. For example, when the system is initially booted, the `pts` pseudo driver will be loaded automatically into the kernel when it is first accessed.

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.

In STREAMS, the header declarations differ between drivers and modules. See Chapter 8, “Modules" and Appendix E, “Configuration", for more information on how to set up the declarations. Also see the appropriate chapters in the *Writing Device Drivers* manual.

STREAMS Entry Points

STREAMS device drivers have interrupt routines that are callbacks registered with the framework. These entry points are accessed via STREAMS, and the call formats differ from traditional character device drivers. (STREAMS drivers are character drivers, too. The non-STREAMS character drivers are considered traditional character drivers or non-STREAMS character drivers.) The `put` procedure is a driver's entry point, but it is a message (not system) interface. The Stream head translates `write` and `ioctl` calls into messages and sends them downstream to be processed by the driver's write queue `put` procedure. `read` is seen directly only by the Stream head, which contains the functions required to process system calls. A driver does not know about system interfaces other than `open` and `close`, but it can detect the absence of a `read` indirectly if flow control propagates from the Stream head to the driver and affects the driver's ability to send messages upstream.

For read-side processing, when the driver is ready to send data or other information to a user process, it prepares a message and sends it upstream to the read queue of the appropriate (minor device) Stream. The driver's open routine generally stores the queue address corresponding to this Stream.

For write-side (or output) processing, the driver receives messages in place of a `write` call. If the message can not be sent immediately to the hardware, it may be stored on the driver's write message queue. Subsequent output interrupts can remove messages from this queue.

Figure 9-1 shows multiple Streams (corresponding to minor devices) connecting to a common driver. There are two distinct Streams opened from the same major device. Consequently, they have the same `streamtab` and the same driver procedures.

The configuration mechanism distinguishes between STREAMS devices and traditional character devices, because system calls to STREAMS drivers are processed by STREAMS routines, not by the system driver routines. In the `cb_ops` structure, the *streamtab* pointer provides this distinction. If it is `NULL` then there are no STREAMS routines to execute. For more detail, see Appendix E, "Configuration".

Multiple instances (minor devices) of the same driver are handled during the initial open for each device. Typically, the `queue` address is stored in a driver-private structure "uniquely identified" by the minor device

number. See also ddi_soft_state (9F). The *q_ptr* of the `queue` will point to the private data structure entry. When the messages are received by the queue, the calls to the driver `put` and `service` procedures pass the address of the `queue`, allowing the procedures to determine the associated device via the *q_ptr* field.

A driver is at the end of a Stream. As a result, drivers must include standard processing for certain message types that a module might simply be able to pass to the next component.

STREAMS guarantees that only one `open` or `close` can be active at a time per major/minor device pair.

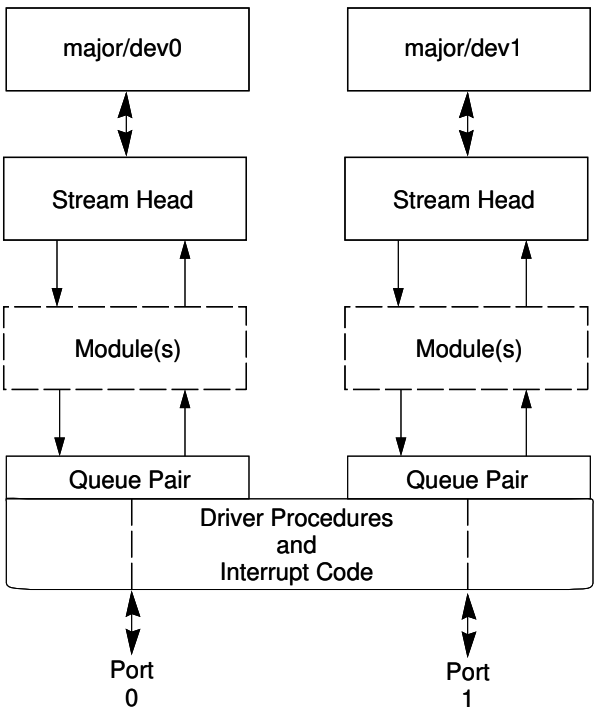

Figure 9-1 Device Driver Streams

Printer Driver Example

The next example shows how a simple interrupt-per-character line printer driver could be written. The driver is unidirectional and has no read-side processing. It demonstrates some differences between module and driver programming, including the following:

- Open handling
  A driver is passed a device number.
- Flush handling
  A driver must loop `M_FLUSH` messages back upstream.
- `ioctl` handling
  A driver must send a negative acknowledgment for `ioctl` messages it does not understand. See Chapter 7, “Overview of Modules and Drivers" in the section “Module and Driver ioctls" for more discussion.

The driver declarations, Code Example 9-1, follow (see also “Module and Driver Declarations” on page 126):

Code Example 9-1 Driver Declarations

```
/* Simple line printer driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo = {
    0xaabb, "lp", 0, INFPSZ, 150, 50 };

static int lpopen(queue_t*, dev_t*, int, int, cred_t*);
static int lpclose(queue_t*, int, cred_t*);
static int lpwput(queue_t*,  mblk_t*);
```

9

Code Example 9-1 Driver Declarations

```
static struct qinit rinit = {
    NULL, NULL, lpopen, lpclose, NULL, &minfo, NULL };

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

struct streamtab lpinfo = { &rinit, &winit, NULL, NULL };

#define SET_OPTIONS (('l'<<8)|1)/* in a .h file */

 /*This is a private data structure,one per minor device number */
struct lp {
    short flags;              /* flags -- see below */
    mblk_t *msg;              /* current message being output */
    queue_t *qptr;            /* back pointer to write queue */
    kmutex_t lp_lock;         /* sync lock */
};

/* Flags bits */
#define BUSY 1     /* dev is running, int will be forthcoming */

extern struct lp lp_lp[];     /*per device lp struct array */
extern int lp_cnt;            /*number of valid minor devices*/

static void lpout(struct lp *lp);
```

The `ddi_soft_state`(9F) manual page describes how to maintain multiple instances of a driver.

The values in the module name and ID fields in the `module_info` structure should be unique in the system.

There is no read-side `put` or `service` procedure. The flow control limits for use on the write-side are 50 bytes for the low watermark and 150 bytes for the high watermark.

The private *lp* structure is indexed by the minor device number and contains these elements:

flags
A set of flags. Only one bit is used: BUSY indicates that output is active and a device interrupt is pending.

msg
A pointer to the current message being output.

qptr
A back pointer to the write queue. This is needed to find the write queue during interrupt processing.

lp_lock
A lock to prevent multithread race conditions.

The STREAMS mechanism allows only one Stream per minor device. The driver open routine is called whenever a STREAMS device is opened. It is `open`'s responsibility to assign a private data structure. The driver `open`, `lpopen` in this example, has the same interface as the module `open`:

```
static int lpopen(
    queue_t *q,                 /* read queue */
    dev_t *devp,
    int flag,
    int sflag,
    cred_t *credp)
{
    extern lp_cnt;               /* max # of lp devices */
    struct lp *lp;
    minor_t device;

    if (sflag) /* driver refuses to do module or clone open */
            return(ENXIO);

    device = getminor(*devp);
    if (device >= lp_cnt)
            return(ENXIO);

    /* Check if open already. Can't have multiple opens */
    if (q->q_ptr) {
            return(EBUSY);
    }
    lp = &lp_lp[device];
    lp->qptr = WR(q);
    q->q_ptr = (char *) lp;
    WR(q)->q_ptr = (char *) lp;
    qprocson(q);
    return(0);
}
```

The Stream flag, *sflag*, must have the value 0, indicating a normal driver `open`. *devp* is a pointer to the major/minor device number for this port. After checking *sflag*, the STREAMS open flag, *lpopen* extracts the minor device pointed to by *devp*, using the `getminor`() function. *credp* is a pointer to a credentials structure.

The minor device number selects a printer. The device number pointed to by *devp* must be less than *lp_cnt*, the number of configured printers. Otherwise failure occurs.

The next check, `if (q->q_ptr)...`, determines if this printer is already open. If it is, EBUSY is returned to avoid merging printouts from multiple users. *q_ptr* is a driver/module private data pointer. It can be used by the driver for any purpose and is initialized to zero by STREAMS before the first `open`. In this example, the driver sets the value of *q_ptr*, in both the read and write `queue` structures, to point to a private data structure for the minor device, *lp_lp[device]*.

There are no physical pointers between the read and write queue of a pair. `WR` is a queue pointer function. `WR`(q) generates the write pointer from the read pointer. `RD` and `OTHER` are additional queue pointer functions. `RD`(q) generates the read pointer from the write pointer, and `OTHER`(q) generates the mate pointer from either. With the DDI, `WR`, `RD`, and `OTHER` are now functions, not macros.

Driver Flush Handling

The following write `put` procedure, *lpwput*, illustrates driver `M_FLUSH` handling. Note that all drivers are expected to incorporate flush handling.

If `FLUSHW` is set, the write message queue is flushed, and (in this example) the leading message (`lp->msg`) is also flushed. lp_lock protects the drivers per instance data structure. Note: there is only one lock for all instances of this driver for the sake of simplicity.

Normally, if `FLUSHR` is set, the read queue would be flushed. However, in this example, no messages are ever placed on the read queue, so it is not necessary to flush it. The `FLUSHW` bit is cleared and the message is sent upstream using `qreply`(). If `FLUSHR` is not set, the message is discarded.

The Stream head always performs the following actions on flush requests received on the read-side from downstream. If FLUSHR is set, messages waiting to be sent to user space are flushed. If FLUSHW is set, the Stream head clears the FLUSHR bit and sends the M_FLUSH message downstream. In this manner, a single M_FLUSH message sent from the driver can reach all queues in a Stream. A module must send two M_FLUSH messages to have the same affect.

lpwput queues M_DATA and M_IOCTL messages and, if the device is not busy, starts output by calling lpout. Messages types that are not recognized are discarded.

```
static int lpwput(
    queue_t *q,                             /* write queue */
    mblk_t *mp)                          /* message pointer */
{
    struct lp *lp;

    lp = (struct lp *)q->q_ptr;

    switch (mp->b_datap->db_type) {
    default:
            freemsg(mp);
            break;

    case M_FLUSH: /* Canonical flush handling */
            if (*mp->b_rptr & FLUSHW) {
                flushq(q, FLUSHDATA);
                mutex_enter(&lp->lp_lock); /* lock any access to
lp */
                if (lp->msg) {
                        freemsg(lp->msg);
                        lp->msg = NULL;
                }
                mutex_exit(&lp->lp_lock);
            }
            if (*mp->b_rptr & FLUSHR) {
                *mp->b_rptr &= ~FLUSHW;
                qreply(q, mp);
            } else
                freemsg(mp);
            break;

    case M_IOCTL:
    case M_DATA:
```

```
            putq(q, mp);
            mutex_enter(&lp->lp_lock);
            if (!(lp->flags & BUSY))
                lpout(lp);
            mutex_exit(&lp->lp_lock);
    }
    return (0);
}
```

Driver Interrupt

The following example shows the interrupt routine in the printer driver.

`lpint` is the driver interrupt handler routine.

`lpout` takes a character from the queue and sends it to the printer. For convenience, the message currently being output is stored in `lp->msg`. It is assumed that this is called with the mutex held.

`lpoutchar` sends a character to the printer and interrupts when complete. Printer interface options need to be set before being able to print.

```
/* Device interrupt routine */

static int
lpint(
    caddr_t intr_arg)          /* minor device number of lp */
{
    struct lp *lp;
    minor_t device = (minor_t) intr_arg;
    lp = &lp_lp[device];
    mutex_enter(&lp->lp_lock);

    if (!(lp->flags & BUSY)) {
        mutex_exit(&lp->lp_lock);
            return (DDI_INTR_UNCLAIMED);
    }
    lp->flags &= ~BUSY;
    lpout(lp);
    mutex_exit(&lp->lp_lock);
    return (DDI_INTR_CLAIMED);
}
/* Start output to device - used by put procedure and driver */
```

```
static void
lpout(
    struct lp *lp)
{
    mblk_t *bp;
    queue_t *q;

    q = lp->qptr;

 loop:
    if ((bp = lp->msg) == NULL) { /*no current message*/
            if ((bp = getq(q)) == NULL) {
                lp->flags &= ~BUSY;
                return;
            }
            if (bp->b_datap->db_type == M_IOCTL) {
                lpdoioctl(lp, bp);
                goto loop;
            }
            lp->msg = bp; /* new message */
    }
    if (bp->b_rptr >= bp->b_wptr) { /* validate message */
            bp = lp->msg->b_cont;
            lp->msg->b_cont = NULL;
            freeb(lp->msg);
            lp->msg = bp;
            goto loop;
    }

    lpoutchar(lp, *bp->b_rptr++); /*output one character*/
    lp->flags |= BUSY;
}
```

Driver Close

The driver close routine is called by the Stream head. Any messages left in the queue will be automatically removed by STREAMS. The Stream is dismantled data structures are released.

```
static int
lpclose(
    queue_t *q,                 /* read queue */
    int flag,
    cred_t *credp)
{
    struct lp *lp;

    qprocsoff(q);
    lp = (struct lp *) q->q_ptr;

    /* Free message, queue is automatically
     * flushed by STREAMS */

    mutex_enter(&lp->lp_lock);
    if (lp->msg) {
            freemsg(lp->msg);
            lp->msg = NULL;
    }
    lp->flags = 0;
    mutex_exit(&lp->lp_lock);
}
```

Driver Flow Control

The same utilities (described in Chapter 8, “Modules") and mechanisms used for module flow control are used by drivers.

When the message is queued, `putq`() increments the value of *q_count* by the size of the message and compares the result against the driver’s write high watermark (*q_hiwat*) value. If the count reaches *q_hiwat*, the `putq`() utility routine will set the internal FULL indicator for the driver write queue. This will cause messages from upstream to be halted (`canputnext`() returns FALSE) until the write queue count drops below *q_lowat*. The driver messages

waiting to be output are dequeued by the driver output interrupt routine with `getq`(), which decrements the count. If the resulting count is below *q_lowat*, the `getq`() routine will back-enable any upstream queue that had been blocked.

For priority band data, `qb_count`, `qb_hiwat`, and `qb_lowat` are used.

STREAMS allows flow control to be used on the driver read-side to handle temporary upstream blocks.

To some extent, a driver or a module can control when its upstream transmission will become blocked. Control is available through the `M_SETOPTS` message (see Appendix B, “Message Types") to modify the Stream head read-side flow control limits.

Cloning

In many earlier examples, each user process connected a Stream to a driver by opening a particular minor device of that driver. Often, however, there is a need for a user process to connect a new Stream to a driver regardless of which minor device is used to access the driver. In the past, this typically forced the user process to poll the various minor device nodes of the driver for an available minor device. To alleviate this task, a facility called *clone open* is supported for STREAMS drivers. If a STREAMS driver is implemented as a cloneable device, a single node in the file system may be opened to access any unused device that the driver controls. This special node guarantees that the user will be allocated a separate Stream to the driver on every `open` call. Each Stream will be associated with an unused major/minor device, so the total number of Streams that may be connected to a particular cloneable driver is limited by the number of minor devices configured for that driver.

The clone device may be useful, for example, in a networking environment where a protocol pseudo-device driver requires each user to open a separate Stream over which it will establish communication.

Note – The decision to implement a STREAMS driver as a cloneable device is made by the designers of the device driver. Knowledge of clone driver implementation is not required to use it.

There are two ways to open as a clone device. The first is by having a CLONEOPEN flag passed in, the result of which is presented in the following example. The second way is to have the driver open itself that way.

For the ptm device, the first technique is:

The module _init routine sets up dev_ops to point to the attach routine, and the cb_ops to point to the open routine through the streamtab. The attach routine shown below creates the device file in /devices, which has a clone major number of 11 and a minor of 23, the major number of the device driver:

```
crw-rw-rw-   1 sys        11, 23 Mar  6 02:05 clone:ptmx
crw-------   1 sys        23,  0 Mar  6 02:05 ptm:ptmajor
```

When the file /devices/pseudo/clone@0:ptmx is opened, the clone code

```
static int
ptm_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
    if (cmd != DDI_ATTACH)
            return (DDI_FAILURE);

    if (ddi_create_minor_node(devi, "ptmajor", S_IFCHR,
        0, NULL, 0) == DDI_FAILURE) {
            ddi_remove_minor_node(devi, NULL);
            return (DDI_FAILURE);
    }

    if (ddi_create_minor_node(devi, "ptmx", S_IFCHR,
        0, NULL, CLONE_DEV) == DDI_FAILURE) {
            ddi_remove_minor_node(devi, NULL);
            return (DDI_FAILURE);
    }

    return (DDI_SUCCESS);
}
```

in the kernel (accessed by major 11) passes the CLONEOPEN flag to the ptmopen routine. ptm then checks sflag to make sure it is a clone driver:

```
static int
ptmopen(rqp, devp, oflag, sflag, credp)
    queue_t *rqp; /* pointer to the read-side queue */
    dev_t   *devp; /* pointer to stream tail's dev */
    int     oflag; /* the user open(2) supplied flags */
    int     sflag; /* open state flag */
```

```
    cred_t *credp; /* credentials */
{
    struct pt_ttys     *ptmp;
    mblk_t             *mop;
    dev_t              dev;

    if (sflag != CLONEOPEN) {
            cmn_err(CE_WARN, "ptmopen: invalid sflag\n");
            return (EINVAL);
    }
```

For the second technique, the log driver will show how it opens a clone device itself. The attach routine is much like the one in the preceeding example.

```
#define    CONSWMIN 0
#define    CLONEMIN 5

static int
log_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
    if (ddi_create_minor_node(devi, "conslog", S_IFCHR,
        CONSWMIN, NULL, 0) == DDI_FAILURE ||
        ddi_create_minor_node(devi, "log", S_IFCHR,
        CLONEMIN, NULL, 0) == DDI_FAILURE) {
            ddi_remove_minor_node(devi, NULL);
            return (DDI_FAILURE);
    }
    return (DDI_SUCCESS);
}
```

But now when the open routine is run, there is special cloning logic in the driver to handle it:

```
static int
logopen(
    queue_t            *q,
    dev_t              *devp,
    int                flag,
    int                sflag,
    cred_t             *cr)
{
    int i;
    struct log *lp;
    /*
     * A MODOPEN is invalid and so is a CLONEOPEN.
```

```
     */
    if (sflag)
            return (ENXIO);

    mutex_enter(&log_lock);
    switch (getminor(*devp)) {

    case CONSWMIN:
            if (flag & FREAD) {/* can only write to this minor */
                mutex_exit(&log_lock);
                return (EINVAL);
            }
            if (q->q_ptr) {           /* already open */
                mutex_exit(&log_lock);
                return (0);
            }
            lp = &log_log[CONSWMIN];
            break;

    case CLONEMIN:
            /*
             * Find an unused minor > CLONEMIN.
             */
            i = CLONEMIN;
            for (lp = &log_log[i]; i < log_cnt; i++, lp++) {
                if (!(lp->log_state & LOGOPEN))
                        break;
            }
            if (i >= log_cnt) {
                mutex_exit(&log_lock);
                return (ENXIO);
            }
            /* clone it */
            *devp=makedevice(getmajor(*devp) (minor_t)i);
            break;
```

Loop-Around Driver

The loop-around driver is a pseudo driver that loops data from one open Stream to another open Stream. The user processes see the associated files almost like a full-duplex pipe. The Streams are not physically linked. The driver is a simple multiplexer that passes messages from one Stream's write queue to the other Stream's read queue.

To create a connection, a process opens two Streams, obtains the minor device number associated with one of the returned file descriptors, and sends the device number in an `ioctl`(2) to the other Stream. For each `open`, the driver open places the passed `queue` pointer in a driver interconnection table, indexed by the device number. When the driver later receives an `M_IOCTL` message, it uses the device number to locate the other Stream's interconnection table entry, and stores the appropriate `queue` pointers in both of the Streams' interconnection table entries.

Subsequently, when messages other than `M_IOCTL` or `M_FLUSH` are received by the driver on either Stream's write-side, the messages are switched to the read queue following the driver on the other Stream's read-side. The resultant logical connection is shown in Figure 9-2. Flow control between the two Streams must be handled explicitly, since STREAMS will not automatically propagate flow control information between two Streams that are not physically connected.

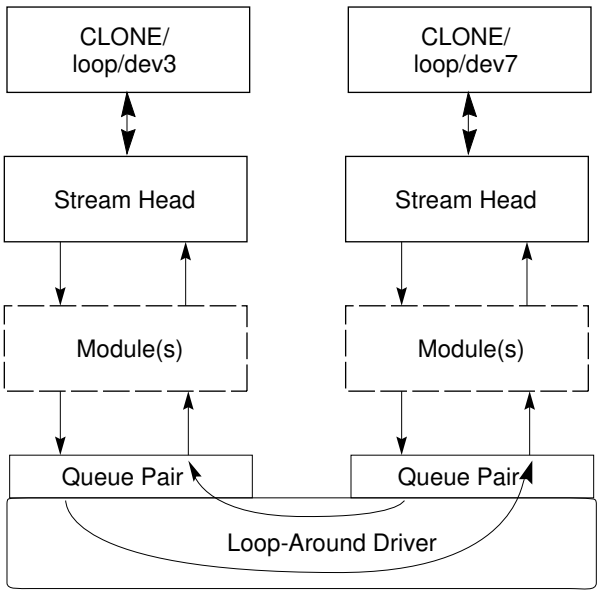

Figure 9-2 Loop-Around Streams

The next example shows the loop-around driver code. The *loop* structure contains the interconnection information for a pair of Streams. *loop_loop* is indexed by the minor device number. When a Stream is opened to the driver, the arriver places the address of the corresponding *loop_loop* element in *q_ptr* (private data structure pointer) of the read-side and write-side `queues`. Since STREAMS clears *q_ptr* when the `queue` is allocated, a NULL value of *q_ptr* indicates an initial `open`. *loop_loop* is used to verify that this Stream is connected to another open Stream.

Note – The code presented here for the loop-around driver represents a single threaded, uni-processor implementation. Multi-processor and multithreading issues such as locking for data corruption and to prevent race conditions are not discussed. See Chapter 13, “Multi-Threaded STREAMS" for details.

The declarations for the driver are:

```
/* Loop-around driver */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static struct module_info minfo = {
    0xee12, "loop", 0, INFPSZ, 512, 128 };

static int loopopen (queue_t*, dev_t*, int, int, cred_t*);
static int loopclose (queue_t*, int, cred_t*);
static int loopwput (queue_t*,  mblk_t*);
static int loopwsrv (queue_t*);
static int looprsrv (queue_t*);
static void loopcopy(mblk *, mblk_t *, uint, unsigned char);

static struct qinit rinit = {
```

```
    NULL,looprsrv,loopopen,loopclose,NULL,&minfo,NULL};

static struct qinit winit = {
    loopwput, loopwsrv, NULL, NULL, NULL, &minfo, NULL };

struct streamtab loopinfo={ &rinit, &winit,NULL,NULL};

struct loop {
    queue_t *qptr;              /* back pointer to write queue */
    queue_t *oqptr;        /* pointer to connected read queue */
};

#define LOOP_SET (('l'<<8)|1) /* in a .h file */
extern struct loop loop_loop[];
extern int loop_cnt;
```

The open procedure includes canonical clone processing that enables a single file system node to yield a new minor device/vnode each time the driver is opened:

```
static int loopopen(
    queue_t *q,
    dev_t *devp,
    int flag,
    int sflag,
    cred_t *credp)
{
    struct loop *loop;

    minor_t newminor;

    if (q->q_ptr) /* already open */
            return(0);
    /*
     * If CLONEOPEN, pick a minor device number to use.
     * Otherwise, check the minor device range.
     */
    if (sflag == CLONEOPEN) {
            for(newminor=0;newminor<loop_cnt;newminor++){
                if (loop_loop[newminor].qptr == NULL) break;
            }
    } else
            newminor = getminor(*devp);
```

```
    if (newminor >= loop_cnt)
            return(ENXIO);

     /*
      * construct new device number and reset devp
      * getmajor gets the major number
      */

    *devp = makedevice(getmajor(*devp), newminor);
    loop = &loop_loop[newminor];
    WR(q)->q_ptr = (char *) loop;
    q->q_ptr = (char *) loop;
    loop->qptr = WR(q);
    loop->oqptr = NULL;

    qprocson(q);
    return(0);
}
```

In *loopopen*, *sflag* can be CLONEOPEN, indicating that the driver should pick an unused minor device (that is, the user does not care which minor device is used). In this case, the driver scans its private *loop_loop* data structure to find an unused minor device number. If *sflag* has not been set to CLONEOPEN, the passed-in minor device specified by `getminor(*devp)` is used.

Since the messages are switched to the read queue following the other Stream's read-side, the driver needs a `put` procedure only on its write-side: {

```
static int loopwput(queue_t *q, mblk_t *mp)
{
    struct loop *loop;
    int to;
    loop = (struct loop *)q->q_ptr;

    switch (mp->b_datap->db_type) {

        case M_IOCTL: {
               struct iocblk             *iocp;
               int                       error=0;
               iocp = (struct iocblk *)mp->b_rptr;

               switch (iocp->ioc_cmd) {
```

```
        case LOOP_SET: {

            /*
             * if this is a transparent ioctl then convert the
             * message into an M_COPYIN message so that the
             * data will ultimately be copied from user space
             * to kernel space.
             */
            if (iocp->ioc_count == TRANSPARENT) {
                loopcopy(mp, (mblk_t *)NULL,
                    sizeof (struct loop), M_COPYIN);
                qreply(q, mp);
                break;  /* leave LOOP_SET case */
            }

            /* fetch other minor device number */
            to = *(int *)mp->b_cont->b_rptr;

            /*
             * Sanity check.  ioc_count contains the amount
             * of user supplied data which must equal the
             * size of an int.
             */
            if (iocp->ioc_count != sizeof(int)) {
                    error = EINVAL;
                    goto iocnak;
            }

            /* Is the minor device number in range? */
            if (to >= loop_cnt || to < 0) {
                    error = ENXIO;
                    goto iocnak;
            }

            /* Is the other device open? */
            if (!loop_loop[to].qptr) {
                    error = ENXIO;
                    goto iocnak;
                }

            /* Check if either dev is currently connected */
            if (loop->oqptr || loop_loop[to].oqptr) {
                      error = EBUSY;
                      goto iocnak;
```

```
                }

            /* Cross connect the streams via the loopstruct */
            loop->oqptr = RD(loop_loop[to].qptr);
            loop_loop[to].oqptr = RD(q);

            /*
             * Return successful ioctl.  Set ioc_count
             * to zero, since no data is returned.
             */
            mp->b_datap->db_type = M_IOCACK;
            iocp->ioc_count = 0;
            qreply(q, mp);
            break;
    }

    default:
            error = EINVAL;

    iocnak:
        /*
             * Bad ioctl.  Setting ioc_error causes the
             * ioctl call to return that particular errno.
             * By default, ioctl will return EINVAL on failure.
             */
            mp->b_datap->db_type = M_IOCNAK;
            iocp->ioc_error = error;
            qreply(q, mp);
    }
    break;
}
}
}

/*
 * Convert mp to an M_COPYIN or M_COPYOUT message (as specified
 * by type) requesting size bytes. Assumes mp denotes a
 * TRANSPARENT M_IOCTL or M_IOCDATA message.  If dp is
 * non-NULL, it is assumed to point to data to be
 * copied out and is linked onto mp.
 */
static void
loopcopy(mblk_t *mp, mblk_t *dp, uint size, unsigned char type)
{
```

```
    struct copyreq *cp = (struct copyreq *)mp->b_rptr;

    cp->cq_private = NULL;
    cp->cq_flag = 0;
    cp->cq_size = size;

    cp->cq_addr = (caddr_t)(*(long *)(mp->b_cont->b_rptr));
    if (mp->b_cont != NULL)
            freeb(mp->b_cont);

    if (dp != NULL) {
            mp->b_cont = dp;
            dp->b_wptr += size;
    } else
            mp->b_cont = NULL;

    mp->b_datap->db_type = type;
    mp->b_wptr = mp->b_rptr + sizeof (*cp);
}
```

loopwput shows another use of an `ioctl` call (see Chapter 7, "Overview of Modules and Drivers" in the section *"Module and Driver ioctls".)* The driver supports a `LOOP_SET` value of *ioc_cmd* in the `iocblk` of the `M_IOCTL` message. `LOOP_SET` instructs the driver to connect the current open Stream to the Stream indicated in the message. The second block of the `M_IOCTL` message holds an integer that specifies the minor device number of the Stream to which to connect.

The driver performs several sanity checks:

- Does the second block have the proper amount of data?
- Is the "to" device in range?
- Is the "to" device open?
- Is the current Stream disconnected? Is the "to" Stream disconnected?

If it passes these sanity checks, the read `queue` pointers for the two Streams are stored in the respective *oqptr* fields. This cross-connects the two Streams indirectly, via *loop_loop*.

Canonical flush handling is incorporated in the `put` procedure:

```
    case M_FLUSH:
            if (*mp->b_rptr & FLUSHW) {
                flushq(q, FLUSHALL);              /* write */
                flushq(loop->oqptr, FLUSHALL);
                /* read on other side equals write on this side */
            }
            if (*mp->b_rptr & FLUSHR) {
                flushq(RD(q), FLUSHALL);
                flushq(WR(loop->oqptr), FLUSHALL);
            }
            switch(*mp->b_rptr) {
            case FLUSHW:
                *mp->b_rptr = FLUSHR;
                break;
            case FLUSHR:
                *mp->b_rptr = FLUSHW;
                break;
            }
            putnext(loop->oqptr, mp);
            break;
    default: /* If this Stream isn't connected,
              * send M_ERROR upstream. */
            if (loop->oqptr == NULL) {
                freemsg(mp);
                (void) putnextctl1(RD(q), M_ERROR, ENXIO);
                break;
            }
            putq(q, mp);
    }
    return (0);
}
```

Finally, *loopwput* queues all other messages (for example, `M_DATA` or `M_PROTO`) for processing by its `service` procedure. A check is made to see if the Stream is connected. If not, an `M_ERROR` is sent upstream to the Stream head.

Certain message types can be sent upstream by drivers and modules to the Stream head where they are translated into actions detectable by user processes. The messages may also modify the state of the Stream head:

M_ERROR
: Causes the Stream head to lock up. Message transmission between Stream and user processes is terminated. All subsequent system calls except close(2) and poll(2) will fail. Also causes an M_FLUSH clearing all message queues to be sent downstream by the Stream head.

M_HANGUP
: Terminates input from a user process to the Stream. All subsequent system calls that would send messages downstream will fail. Once the Stream head read message queue is empty, EOF is returned on reads. This can also result in the SIGHUP signal being sent to the process group's session leader.

M_SIG/M_PCSIG
: Causes a specified signal to be sent to the process group associated with the stream.

putnextctl() and putnextctl1() are utilities that allocate a non-data (that is, not M_DATA, M_DELAY, M_PROTO, or M_PCPROTO) type message, place one byte in the message (for putctl1next) and call the put procedure of the specified queue.

Service procedures are required in this example on both the write-side and read-side for flow control:

```
static int loopwsrv(queue_t *q)
{
    mblk_t *mp;
    struct loop *loop;

    loop = (struct loop *)q->q_ptr;

    while ((mp = getq(q)) != NULL) {
        /* Check if we can put the message up
         * the other Stream read queue */
        if (mp->b_datap->db_type <= QPCTL &
                !canputnext(loop->oqptr)) {
            putbq(q, mp); /* read-side is blocked */
            break;
        }
        /*
         * send message to queue following
         * other Stream read queue
         */
        putnext (loop->oqptr, mp);
```

```
    }
    return (0);
}

static int looprsrv(queue_t *q)
{
    /* Enter only when "back enabled" by flow control */
    struct loop *loop;
    loop = (struct loop *)q->q_ptr;

    if (loop->oqptr == NULL)
            return (0);
    /* manually enable write service procedure */
    qenable(WR(loop->oqptr));
    return (0);
}
```

The write `service` procedure, *loopwsrv*, takes on the canonical form. The queue being written to is not downstream, but upstream (found via *oqptr*) on the other Stream.

In this case, there is no read-side `put` procedure so the read `service` procedure, *looprsrv*, is not scheduled by an associated `put` procedure, as has been done previously. *looprsrv* is scheduled only by being back-enabled when its upstream becomes unstuck from flow control blockage. The purpose of the procedure is to re-enable the writer (*loopwsrv*) by using *oqptr* to find the related `queue`. *loopwsrv* can not be directly back-enabled by STREAMS because there is no direct `queue` linkage between the two Streams. Note that no message is queued to the read `service` procedure. Messages are kept on the write-side so that flow control can propagate up to the Stream head. The `qenable`() routine schedules the write-side `service` procedure of the other Stream.

loopclose breaks the connection between the Streams:

```
static int loopclose(
    queue_t *q,
    int flag,
    cred_t *credp)
{
    struct loop *loop;

    loop = (struct loop *)q->q_ptr;
    loop->qptr = NULL;
```

```
    /* If we are connected to another stream, break the
     * linkage, and send a hangup message.
     * The hangup message causes the stream head to reject
     * writes, allow the queued data to be read completely,
     * and then return EOF on subsequent reads.
     */
    if (loop->oqptr) {
            (void) putnextctl(loop->oqptr, M_HANGUP);
            loop->oqptr = NULL;
    }
    qprocsoff(q);
    ((struct loop *)loop->oqptr->q_ptr)->oqptr = NULL;
    return (0);
}
```

loopclose sends an M_HANGUP message up the connected Stream to the Stream head.

Design Guidelines

Driver developers should follow these guidelines:

- Messages that are not understood by the drivers should be freed.
- A driver must process all M_IOCTL messages. Otherwise, the Stream head will block for an M_IOCNAK, M_IOCACK, or until the timeout (potentially infinite) expires.
- If a driver does not understand an ioctl, an M_IOCNAK message must be sent to upstream.
- The Stream head locks up the Stream when it receives an M_ERROR message, so driver developers should be careful when using the M_ERROR message.
- A hardware driver must have an interrupt routine.
- Multithreaded drivers need to protect their own data structures

Also see the section “Design Guidelines" in Chapter 7, “Overview of Modules and Drivers".

9