540

SEQUENT

DYNIX/ptx® STREAMS Application Programming Guide

1003-48616-01

An important difference between STREAMS drivers and modules is illustrated here:

- Drivers are accessed through a node or nodes in the filesystem and can be opened just like any other device.
- Modules do not occupy a filesystem node. Instead, they are identified through a separate naming convention, and are inserted into a stream using the I_PUSH `ioctl` command.

The name of a module is defined by the module developer, and is typically included on the man page describing the module. (Man pages describing STREAMS drivers and modules are found online and in Section 7 of the *DYNIX/ptx Reference Manual.*)

Modules are always pushed onto a stream immediately below the stream head. Therefore, if a second module is pushed onto this stream, it is inserted between the stream head and the case-converter module.

Module and Driver Control

The next step in this example is to pass the input string and output string to the case-converter module. This is accomplished by issuing `ioctl` calls to the case-converter module.

`ioctl` requests are issued to drivers and modules indirectly, using the I_STR `ioctl` command (refer to **streamio**(7)). The argument to the I_STR `ioctl` command must be a pointer to a `strioctl` structure that specifies the request to be made to a module or driver. This structure is defined in <*stropts.h*> and has the following format:

```
struct strioctl {
    int ic_cmd; /* ioctl request */
    int ic_timout; /* ACK/NAK timeout */
    int ic_len; /* length of data argument */
    char *ic_dp; /* ptr to data argument */
}
```

The putmsg system call enables a user to create messages and send them downstream. The user supplies the contents of the control and data portions of the message in two separate buffers. The getmsg system call retrieves such messages from a stream and places the contents into two user buffers.

The syntax of putmsg is as follows:

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

These are the putmsg parameters:

fd	Identifies the stream to which the message is passed
ctlptr and *dataptr*	Pointers to the control and data portions of the message
flags	Used to specify whether a priority message should be sent

The strbuf structure describes the control and data portions of a message. It has the following format:

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

These are the fields of strbuf:

maxlen	Specifies the maximum number of bytes the designated buffer can hold; meaningful only when getmsg retrieves information into the buffer
len	Specifies the number of bytes of data in the buffer
buf	Points to a buffer containing the data

The getmsg system call retrieves messages available at the stream head. It has the following syntax:

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

The parameters to getmsg are the same as those for putmsg.

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

putpmsg() has the following interface:

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

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

The call

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

is equivalent to the system call

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

The driver declares the variable `pollfds` as an array of `pollfd` structures. This structure is defined in *<poll.h>* and has the following format:

```
struct pollfd {
   int fd; /* file descriptor */
   short events; /* requested events */
   short revents; /* returned events */
}
```

For each entry in the array, *fd* specifies the file descriptor to be polled, and *events* is a bitmask that contains the bitwise inclusive OR of events to be polled for the file descriptor. On return, the *revents* bitmask indicates which of the requested events occurred.

The following code opens two separate minor devices of the communications driver and initializes the `pollfds` entry for each. `poll` is used to process incoming data.

```
            /* set events to poll for incoming data */
pollfds[0].events = POLLIN;
pollfds[1].events = POLLIN;

while (1) {
            /* poll and use -1 timeout */
            /* (infinite) */
   if (poll(pollfds, NPOLL, -1) < 0) {
      perror("poll failed");
      exit(3);
   }
   for (i = 0; i < NPOLL; i++) {
      switch (pollfds[i].revents) {
      default: /* default error case */
         perror("error event");
         exit(4);
      case 0: /* no events */
         break;
      case POLLIN:
            /* echo incoming data on "other" stream */
         while ((count = read(pollfds[i].fd, buf, 1024))
         > 0)
            /*
            * the write loses data if flow control
            * prevents the transmit at this time.
            */
            if (write(
```

Setting Up Autopush for a STREAMS Module

A STREAMS driver can establish a list of modules that the streams subsystem will automatically push on the stream whenever any of its minor devices is first opened.

A STREAMS module can also be bound to another such that anytime the module is pushed on a stream, the other module is automatically pushed after it (a *compound module*).

The operating system provides a routine, `str_ap_add()`, for setting up autopush of a streams module on the first open of a streams driver. During system initialization, this routine dynamically allocates and adds entries to the static autopush table. Following is a description of this routine:

```
void *

str_ap_add(char *driver_name,          /* driver name or "" */
               unsigned int lminor,  /* lowest minor to autopush on */
               unsigned int hminor,  /* highest minor to autopush on */
               char *module_name,    /* module name to be autopushed */
              void *next_ap)       /* ptr to next module to be autopushed*/
```

Add an Entry to the Global str_ap Autopush Table

STREAMS drivers call the `str_ap_add()` function from their **init** routines to establish a list of modules to be autopushed on first open of any minor device matching the `lminor-hminor` criteria. Any module in the list can be limited to a specific range of minor devices (via the `lminor-hminor` range specification).

Example 1: Simple List Of Modules

A driver *drv* specifies two modules, *mod1* and *mod2*, to be autopushed whenever any of its minor devices is first opened. *mod1* will be pushed first; *mod2* will be pushed last. Both modules will be pushed for all minor devices of driver *drv*.