Gently Down The STREAMS

You have probably heard about STREAMS. They are those gentle water-ways we like to peacefully drift down while relaxing on a warm summer day. In a recreational universe yes, but in a computing universe, not quite. In a computing universe STREAMS is a facility for developing UNIX system communications services. As such, it defines a processing model and standard interface for character input/output within the kernel and between the kernel and the rest of the UNIX system. It consists of a set of system calls, kernel resources, and kernel routines. (So much for that imagery of relaxing on a warm summer day.)

A Quick Low Level Look

This article isn't about writing STREAMS modules or device drivers. It is about using STREAMS for application development. However, a quick low-level look at things will help with perspective when we get on to what can be done with them.

STREAMS are implemented in the kernel. A *Stream* is a full-duplex processing and data transfer path between a STREAMS *driver* in kernel space and a process in user space (see Figure 1). A complete Stream is constructed in the kernel by linking a Stream head with a STREAMS driver and zero or more STREAMS modules between the Stream head and the driver.

```
scale=100
define m0 |
[ box invis ht 456 wid 352 with .sw at 0,0
line -> from 112,176 to 112,136
line  from 0,368 to 352,368 dashed
line -> from 112,288 to 112,248
line -> from 64,344 to 64,288
line <-> from 160,64 to 160,24
line -> from 264,80 to 264,136
ellipse ht 48 wid 112 at 160,432
line <-> from 160,408 to 160,360
box ht 72 wid 120 with .nw at 104,360
box ht 72 wid 120 with .nw at 104,248
box ht 72 wid 120 with .nw at 104,136
line <- from 216,288 to 216,248
line <- from 216,176 to 216,136
"User Space" at 284,381
"downstream" at 52,353
"upstream" at 264,69
"Stream Head" at 168,325
"Module" at 164,221
"(optional)" at 164,201
"Driver" at 164,97
"External Interface" at 164,13
"Kernel Space" at 292,349
"User Process" at 160,429
] |

box invis ht 456 wid 352 with .sw at 0,0
m0 with .nw at 0,456
```

Figure 1: A Simple Stream

The user-level interface to a Stream is via the system calls

- `open()`
- `close()`
- `read()`

- `write()`
- `putmsg()`
- `getmsg()`
- `ioctl()`

A *Stream head* is the end of a Stream nearest user space. Systems calls made by a user process on a Stream are processed at the Stream head. A STREAMS *driver* may be a device driver that provides the services of an external I/O device, or a software driver (a.k.a. a psuedo-device driver.) Drivers simply handle data transfer between the kernel and the device they drive and do little or no processing other than converting data structures, used by the STREAMS mechanism, to and from data structures used by the device.

A STREAMS *module* provides processing functions that are used to manipulate data flowing on the Stream. The module is defined as a set of kernel-level routines and data structures used to process data, status, and control information. This can involve changing the representation of data (e.g. conversion or filtering), adding/deleting header and trailer information to data, and/or packetizing/depacketizing data. A STREAMS module is self-contained and functionally isolated from any other component in the Stream except its two neighboring components. A module communicates with its neighbors by passing messages.

The beauty of a STREAMS is that it is a well-defined interface for doing character-based I/O. STREAMS were incorporated into UNIX System V in release 3. At that time many device drivers were converted into STREAMS-based drivers. That is, they conformed to the STREAMS interface definition. Among the character-based I/O facilities not converted to be STREAMS-based were pipes and the TTY-subsystem. This has been remedied in SVR4. All character-based I/O is now STREAMS-based and several new modules have been added.

The power of STREAMS is in the mechanism itself and the modules. First, STREAMS have abilities far beyond those of mortal I/O channels. For example, now that pipes are STREAMS-based their functionalty has increased dramatically. They are still created using `pipe()` or `mknod()`, and you can still use `read()` and `write()` to send and receive data. However, now you can

- avoid blocking on `read()`'s and `write()`'s by using the `poll()` system call to detect arriving data and ‘‘clear to send’’;
- detect data arriving on the pipe by having the kernel generate a signal when data arrives;
- send an open file-descriptor from one process to another using a pipe;
- send normal data along the pipe, define your own ‘‘bands’’ of data, or send ‘‘out-of-band’’ data - i.e. high-priority messages one the same Stream you are sending normal messages;
- mount on end of a pipe onto the file-system.

Second, since STREAMS modules reside in the kernel, what they do in the course of processing data is wide open. Therefore, the ability to configure and control a Stream from user level is the ability to dynamically control kernel resources.

What Can You Do

Since we have covered most of the background material and made some wild claims, at this juncture it would be prudent to get on with the examples (waves hands like George Bush).

A Client-Server Example

I guess client-server computing has gained a new level of legitimacy now that Sun is spending big-bucks on advertising that reminds us of *Return Of The Jedi*. So lets look at a client-server example that isn’t quite as exciting as riding some sort of jet-cycle through a redwood forest but enriches our spirit by bringing closer to the ‘‘purest’’ form of computing :-)

Consider a client-server application where the server is a standing server (i.e. a daemon UNIX parlance) that takes
queries from its clients and runs them on their behalf against a database. Clients connect to the server by opening a well-known named pipe and then send their queries down the pipe to the server. The server parses the query, schedules it to be run, eventually runs it and returns the results to the client via mail.

```
#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <fcntl.h>
#include  <stropts.h>

#define   FIFO "/var/spool/server-name/FIFO"

typedef   struct    strbuf         strbuf_t;
typedef   struct    pollfd         pollfd_t;
typedef   struct    strrecvfd strrecvfd_t;

int
main (void)
{
     int       fds [2];
     pollfd_t  pollfd;
     strrecvfd_t    recvfd;
     /*
     **  Create the mount point for the pipe we
     **  intend to mount.
     */
     fds [0] = open (FIFO, O_RDWR|O_CREAT, 0666);
     (void)    close (fds [0]);
     /*
     **  Create the pipe, push the CONNLD module
     **  onto the end we intend to mount and then
     **  mount it.
     */
     (void)    pipe (fds);
     (void)    ioctl (fds [1], I_PUSH, "connld");
     (void)    fattach (fds [1], FIFO) < 0)
     (void)    chmod (FIFO, 0666);
     /*
     **  Now the only kind of input we will get
     **  on fds[0] will be incomming file-
     **  descriptors.
     */
     pollfd.fd = fds [0];
```

```
     pollfd.events = POLLIN;
     for (;;)
     {
          (void)    poll (pollfd, 1, -1))

          (void)    ioctl (fds [0], I_RECVFD, &recvfd);

          recvfd.uid
          recvfd.gid

          ProcessClient (&recvfd);
     }
     return    0;
}
```

```
#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <stropts.h>

#define   FIFO "/var/spool/server-name/FIFO"

int
main ()
{
     fd = open (FIFO, O_RDWR);

     RequestService (fd);

     return    0;
}
```

Acknowledgements

I would like to thank Torez Hiley and ???, of UNIX Systems Laboratories, Inc., for their help of reviewing this article.

Bibliography

[1] *UNIX System V Release 4: User's Reference Manual,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] *UNIX System V Release 4: Programmer's Guide: STREAMS,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] *UNIX System V Release 4: Programmer's Reference Manual,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[4] *UNIX System V Release 4: System Administrator's Reference Manual,* Prentice Hall, Englewood Cliffs, NJ, 1990.