AIX Communication Service: Streams

By Eddie Ho, Saurabh Desai, Derwin Gavin, and James Partridge

Streams is a standard interface for developing modular, portable networking layers that reuse code from different communication protocols. It facilitates the rapid growth of communication services. The Streams framework on AIX Version 3.2.5 can save development costs for application developers and communications subsystem providers. This article introduces system engineers to Streams.

Communications support is the primary building block for commercial applications on AIX. Other enabling applications include Online Transaction Processing (OLTP), relational databases, and multimedia. These applications have unique real-time interface requirements and use the standard interfaces provided by the communication subsystem to interoperate with other applications.

The latest network technology using Streams can assist and shorten the development time for application developers and communication subsystem providers. Streams is the de facto standard from AT&T® that solves many architectural problems. It is adapted by all vendors using AT&T System V Release 4. Since Streams is a defined, common standard, it also enhances application portability. A Streams module written for one system should run under Streams on another system with little or no modification.

Because Streams is based on the seven-layer Open Systems Interconnection (OSI) model, it promotes modularity. Delivering an old application over a new communications service should require replacing only the lower-level modules of a stream.

There are several benefits of using Streams for communication services:

- The protocol stack can be customized in real time by the user application. This makes the code path more efficient and direct.
- Protocol layers and device drivers can be shared and mixed.
- Runtime protocols can be configured by the user application.
- Conformance to the OSI seven-layer design results in building blocks with a common message-driven interface.
- It promotes reusable code, which saves development time and resources.

What is Streams?

Streams is a flexible set of tools for developing UNIX system communication services. It defines a generic message-driven queuing interface and provides a framework and tools for implementing communication services, such as a network protocol stack. Streams does not impose any specific network architecture—it is simply a framework. Streams includes the following components:

Stream head: The head is the part of the stream that provides an interface to the user process. It supports all the standard device-driver system calls (`open`, `close`, `read`, `write`, and `ioctl`) and three additional calls—`putmsg`, `getmsg`, and `poll`.

Stream modules: Stream modules are a defined set of kernel-level routines and data structures used to process data, status, and control information. They implement processing functions to be performed on data that flows on the stream. There may be zero or more modules depending on processing requirements.

Stream device driver: The Streams-based device driver can support an external I/O device or a pseudo-device. The driver typically handles data transfers between the kernel and the device. It does little or no data processing other than converting Streams messages to hardware events and vice versa. To be modular, the program

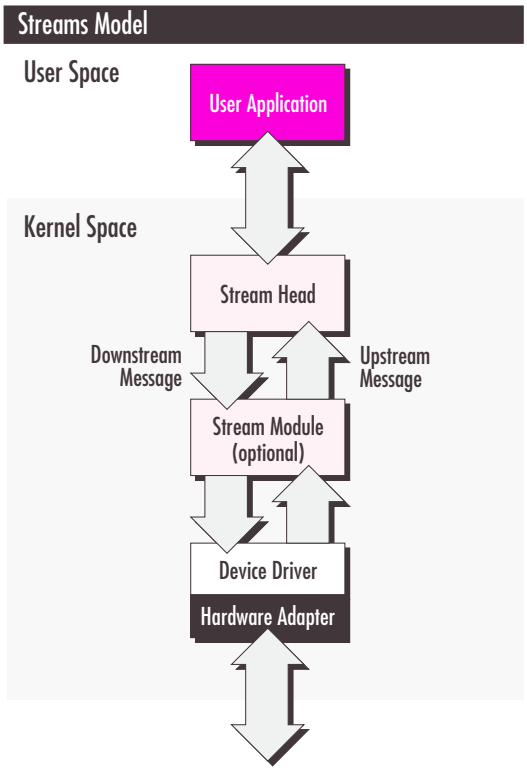

Figure 1. Streams model

System Call	Action
`open()`	Open a stream. The open system call recognizes a Streams file and creates a stream to the specified driver.
`close()`	Close a stream.
`read()`	Read data from a stream.
`write()`	Write data to a stream.
`ioctl()`	Control a stream. This allows the application to control device-specific functions. The `ioctls` supported include all `ioctls` normally supported by a device driver, plus `ioctls` supported by the Streams framework (`I_LIST`, `I_PUSH`, `I_POP`, and so on).
`getmsg()`	Receive a message at the stream head.
`putmsg()`	Send a message downstream.
`poll()`	Notify the application program when selected events occur on a stream. When used with the Streams `I_SETSIG ioctl` command, the `poll()` system call allows an application to process I/O in an asynchronous manner.

Figure 2. System calls

should isolate all hardware dependencies to this module.

Figure 1 shows a simple transport using Streams. The user application makes requests to the stream head, which satisfies the request if possible. Otherwise, the request is passed downstream until a module can satisfy it. The response to the request is passed upstream, back to the application.

The Streams framework is a set of utilities that supports operating the stream and provides programmatic control to the application. This control consists of functions such as listing Streams module names on the stream stack, real-time pushing and popping of Streams modules onto and off the stack, and obtaining statistical information from the Streams modules.

The Streams framework also supports a facility for high-priority messages that are transmitted and processed out of sequence—before processing normal data and independent of any flow control imposed on that data.

Streams API

The Streams facility is controlled directly by using system calls. The interface is upwardly compatible with the existing character I/O facilities. Figure 2 shows the system calls.

The Streams-specific system calls are as follows:

putmsg interface: This interface is similar to `write()`. The `putmsg()` subroutine provides a data buffer that is converted into an `M_DATA` message. It can also provide a separate control buffer to be placed into an `M_PROTO` block. The `putpmsg()` supports priority messages, which are processed before ordinary messages and stream data. The normal `write()` interface provides only the stream data that is converted into an `M_DATA` message.

getmsg interface: This interface is similar to `read()`. The primary difference is that `read()` accepts only the data (messages sent upstream to the stream head as message type `M_DATA`), while `getmsg()` can simultaneously accept both data and control information (messages sent upstream as types `M_PROTO` for `getmsg()` or `M_PCPROTO` for `getpmsg()`). The `getmsg()` interface also differs from `read()` by preserving message boundaries so that the same boundaries exist above and below the stream head. A normal `read()` generally ignores message boundaries.

Stream Message Format

Messages are the communicating vehicle within Streams. A message contains data or information

Streams Message Queue Structure

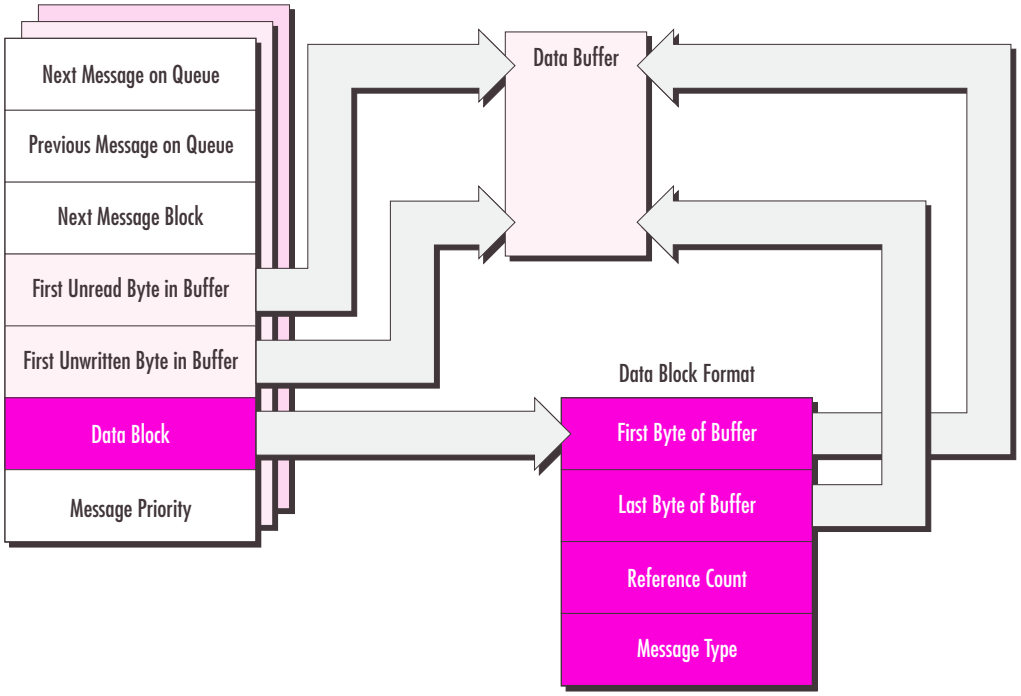

Figure 3. Streams message queue structure

based on the type of message. The messages can originate with a driver, a module, or the stream head. Transferring messages between modules is done via system calls that remove a message from a queue or place a message on a queue to be serviced by another Streams module. The Streams framework maintains its own storage pool for messages. Figure 3 shows the message block structure.

A module can send any message type in either direction on a stream. However, based on their intended use and their treatment by the stream head, each message can be categorized as upstream, downstream, or bidirectional, allowing for full-duplex communication.

The most common normal message types are as follows:

- **M_DATA message**: Intended for user data. The message is generally sent bidirectionally.
- **M_PROTO message**: Contains control information and data. The message format is a single `M_PROTO` message followed by one or more `M_DATA` messages.
- **M_PCPROTO message**: Contains the same format and characteristics as the `M_PROTO` message except for priority and some additional information. This message is intended to send data and control information outside the normal-flow control message path.

Streams in AIX

Communications subsystem vendors use the Streams framework as the implementation of choice. NetWare® services and Transport Layer Interface (TLI) applications use Streams.

Using Streams, NetWare/6000 shown in Figure 4 supports such protocols as SPX/IPX, NetBIOS, and NVT protocol.

The TLI facility allows applications to create connections to remote peers in a protocol-neutral environment. Protocol layer substitution is the means to achieve this in the Streams environment. Depending on the transport provider specified by the application during the session open time, the Transport Interface module can access either a Streams-based protocol stack natively or a socket-based protocol stack such as TCP/IP.

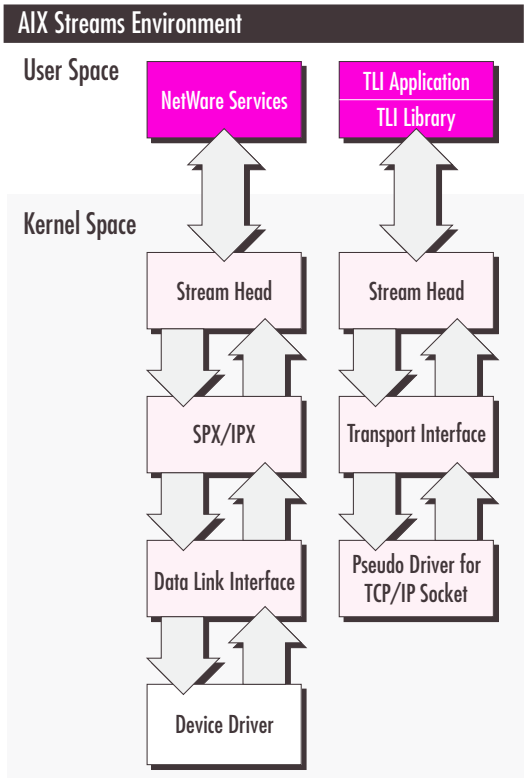

Figure 4. AIX Streams environment

Conclusion

Because Streams is a generic message-driver interface, Streams modules make few assumptions about the modules above and below them in the stream stack. This isolates upper-level Streams modules and application code from changes in lower-level environments and hardware interfaces, providing greater flexibility in configuring applications for delivery.

Eddie Ho, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Ho is a senior programmer in AIX Communications. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.*

Saurabh Desai, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Desai works in AIX Communications. He has a BS in Electronics from S.P. University in India and a MS in Computer Systems Design from the University of Houston.*

Derwin Gavin, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Gavin is an associate programmer in AIX Communications. He has a BS in Computer Science from Grambling State University in Louisiana.*

James Partridge, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Partridge is a staff programmer in AIX Communications. He has a BA in English from the University of Texas.*

Awards for PowerPC, RISC System/6000

BYTE Magazine—The IBM PowerPC 601 microprocessor recently received *BYTE* Magazine's Editorial Award of Excellence for one of the best technologies of 1993. According to *BYTE*, the PowerPC 601 "was the biggest overall vote-getter by a wide margin." IBM Microelectronics Division manufactures the PowerPC 601, which was co-developed with Motorola™.

Reseller Management—The 60,000 readers of *Reseller Management* magazine recently named IBM's RISC System/6000 the "Best-To-Sell RISC Workstation" in its 1993 Readers Choice Awards. In the November 1993 issue announcing the award, the publication said, "IBM's RISC System/6000 is the winner in the workstations category. It not only won, but garnered three times the votes of the longtime champ, Sun Microsystems®."

Datamation—The RISC System/6000 received the 1993 "Product of the Year" award from *Datamation* in the PC/Workstation category. ■