Software Design Issues for the PS-186 Advanced Packet Network Controller

Brian Kantor, *WB6CYT*
Academic Network Operations Group
Office of Academic Computing
University of California, San Diego

Abstract

A fast network for amateur radio requires sophisticated node controllers to work well. Key to the performance of advanced node controller hardware is the design of the on-board software. Issues of highly-efficient device drivers, protocol encapsulation, and process management must be addressed to ensure acceptable performance with limited memory and affordable hardware. PS-186 hardware design issues are discussed in a companion paper by Michael Brock, Franklin Antonio, and Tom LaFleur.

1. Introduction

A high-throughput data network must consist of both high speed links and fast network node controllers. To achieve the high throughput in the controller requires both good hardware and efficient software.

The PS-186 offers a highly efficient hardware design including very high speed input/output and a fast processor. The PS-186's high-speed DMA channels allow much of the I/O to proceed in parallel with computation, thus overlapping I/O and processing that in a less sophisticated system might need to proceed serially.

However, even the most advanced hardware can be crippled by inefficient software that wastes CPU and I/O resources rather than applying them to useful processing. To take full advantage of the PS-186 architecture, we have chosen to use a multi-tasking system that can support several programs running at once. By dividing up tasks into those that are time-critical and those that are not, we can set up the critical tasks in the system such that they will receive the required CPU attention. Less critical tasks will proceed as time permits.

The PS-186 multi-tasking operating system is based in general terms upon the **UNIX**@ and other similar simple operating systems. In particular, many of the ideas and practices used have been taken from those presented in the MINIX [IO] and XINU [2,3] model operating systems.

2. Software Organization

The PS-186 operating software can be divided into two categories. One of these is the central or core part of the system, referred to as the *kernel.* It is responsible for all supervisory low-level I/O functions, process and memory management, interrupt handling, and time-critical tasks. The remaining software is termed the *user* **level** software, although there are, strictly speaking, no users on **this** system. The true distinction is that while there **may** be many "user" processes running at one time, there is only one kernel.

Author's current address:
electronic: brian@sdcsvax.ucsd.edu
paper: Office of Academic Computing B-028, La Jolla, CA 92093 **USA**

User processes can be stopped while they are waiting for input, or while the kernel is handling some other event such as an arriving packet. They are typically used for purposes that are not time-critical and that can operate independently of particular hardware status. A few examples of tasks that might better be placed in user processes are table lookups, help menu displays, and the like. These are things that can proceed in parallel with other similar tasks.

The kernel, on the other hand, is strictly *single-threaded* – it can only execute by itself, and is intended for those tasks that need to have exclusive access to the processor or devices, such as interrupt handlers and device drivers.

User processes do NOT directly access devices, nor do they handle interrupts. All user processes communicate with the kernel by means of *system calls* that cause the kernel to perform some task on behalf of the user process. A common example is a read or a write – data transfer between a user process and a device.

The kernel is responsible for performing all encapsulating protocols below some arbitrary level, which we have chosen to be at the "data stream" level. That means that when an AX.25 connection is made to the PS-186, the kernel software is responsible for the acceptance, acknowledgement, and eventual knockdown of the connection. The kernel will extract the data field from the incoming AX.25 packet and make it available to a user process executing an appropriate *read.* Likewise, a user process that wishes to send data over an open AX.25 connection will give the data to the kernel by means of a *write* system call, and the kernel will do the encapsulation necessary to send the data via AX.25 and then queue it for transmittal by the appropriate device.

When there are multiple protocols involved, such as TCP-IP on AX.25, the kernel does the multiple extractions and encapsulations as required, so that the user process again works only at the data level.

The decision on whether to place a particular task in the kernel or leave it to user-level processing is based on a number of criteria, some of them empirical. In general, any task which can wait to complete without impacting the performance of other tasks can generally be placed in a user-level process, whereas tasks that have a number of process dependencies pretty much have to go inside the kernel. Additionally, any protocol encapsulation or unwrapping that does not generate additional packets can be placed in the kernel, thereby making that function available to all user-level processes by means of a uniform system call.

3. Input-Output

Each device in the PS-186 has a module of code associated with it in the kernel that does the low-level input-output interface to the actual hardware. This module is often referred to in the literature as a *device driver.*

At the low level hardware interface, a device driver is responsible for taking data to be output from some generalized system data structure, and actually outputting it through the corresponding piece of hardware. It also must accept incoming data from the hardware device and place it into a system data structure for further processing. It is common for device drivers to operate in an *interrupt-driven* mode, with their actions being invoked in response to "completion" or "ready" signals from the hardware. We have chosen this method over a perhaps simpler scheme where the main software loop simply repetitively checks for device availability, because the latter scheme potentially wastes a tremendous portion of the available processing power. Additionally, the various drivers make use of the PS-186's DMA (Direct Memory Access) capability to move the actual data between memory and the device without the need of the CPU to read and write every byte.

There also must be a simple and consistent higher-level interface to the system data structures that the low-level device drivers access. We have chosen to implement this interface as *read* and *write* system calls that invoke high-level portions of the device drivers. Additionally, there are both high- and low-level configuration, status, and initialization functions that are logically part of the device driver. Thus each driver can be divided into two logical functions, referred to as the "top" and "bottom" of the driver.

The "bottom" function is the interface to the hardware; it is invoked in response to an interrupt from the actual device. Typically its sole function is to move data to and from the device and an associated memory buffer or buffer queue.

The "top" function is the interface to the kernel *read* and *write* system calls. It does the opposite of its corresponding "bottom" half; where the bottom half places incoming hardware data into a buffer, the top half will remove the data from the buffer and give it to the process executing the *read* kernel call.

When the PS-186 kernel has data to be sent out of a serial port (in response to a *write* system call), the device driver is called to accept the outgoing data. The driver adds the data to the tail end of a queue of data waiting to be sent, sets a flag indicating that there is indeed data to be sent, and returns. Later, when the output device finishes with the data it was sending, it will cause an interrupt to occur, and the "bottom half" of the appropriate device driver will take the next chunk of data from the queue and send it to the device. Thus the kernel (and therefore user processes) need not wait for I/O on a device to complete before resuming proceeding.

Data buffers and queues are dynamically allocated; when data is received or generated a "buffer" (a block of memory) is allocated from the pool of available memory to hold it, and a "pointer" that contains the memory address of the block is set up. To save the time that would be wasted in copying from one block of memory to another, data is passed from module to module by passing the pointer to the memory buffer in which the data resides, rather than copying the data itself. When the data is finally consumed, either by being output by a device, or copied into a user-level (outside the kernel) buffer by a *read* system call, the memory space used is returned to the available memory pool, and the buffer pointer is dereferenced.

When a call to the kernel with data to be output (a *write* call) would require allocation of more memory buffer space than is allowed, the process making the call is stopped by the simply not returning from the system call to that process until there is space and the write can be completed. Since "blocking" the process in this manner does NOT stop interrupt service nor other kernel functions the device will eventually output enough data to free up sufficient memory for the write to complete and for the user process to resume.

On input, if a chunk of data arrives and there is no memory available to hold it, the only practical procedure is to simply discard the data. We anticipate having a large amount of memory available for data buffers, as well as expecting good throughput, so we do not anticipate that it will necessary to discard data often. As a practical note, we have decided to provide each input device with its own memory buffer limit so that no one device could hog all available memory and shut out input from other devices even in the most pathalogical of cases. The overriding assumption is that higher-level protocols will handle packets lost due to memory congestion in much the same way that packets lost due to collisions or channel congestion are handled.

A kernel *read* call will return data from the input queue to the user process; if there is no data in the queue, the user process may elect to wait until there is ("read-wait") or just return ("read-no-wait"), When data is available, it is copied into a buffer space provided by the user process (typically a character array), and the memory buffer space is released to be reused on subsequent input events.

One can view the input-output streams as a series of filtered interfaces to the raw packets that are being received or sent. Thus it is possible to open a connection that consists of raw AX.25 frames, an AX.25 connected mode stream, IP packets in SLIP, IP packets in AX.25, TCP in IP in AX.25, etc. This is controlled by the parameters passed to the kernel in the open system call.

4. Devices

The PS-186 devices that are most interesting are the several serial controller chips that form the communications interfaces. (There is an SCSI controller option for general device access, such as to a disk or floppy controller, but we will not discuss that here.) The serial controller chosen was the Zilog 8530 SCC; the hardware design considerations that lead to it being chosen are discussed in the companion paper on the hardware design of the PS-186.

The 8530 SCC can do both asynchronous serial I/O (as perhaps to a terminal or printer), and HDLC synchronous, such as is used in the AX.25 protocol. Any of the PS-186's serial ports can be configured to operate in either of these modes. We therefore have a more complex device driver than if the PS-186 had fixed serial port allocations, since the device driver must be able to handle both sync- and async-configured devices based on parameters stored in a table: The driver is also responsible for setting up the modes of the serial ports in the first place.

5. Protocol Handling

Fundamental to the operation of communications protocols is the concept of *layering* or *encapsulation*, whereby data is successively encapsulated or "wrapped" in layers of protocol as it is prepared for transmission, and "unwrapped" at its destination.

The basic concept used is known as a switch. As a railroad switch controls the path of a train, the switch controls the path that data takes through the various levels of encapsulation and unwrapping. A protocol *switch* makes the data passing decision based on a field contained in each protocol's header that indicates what kind of protocol may be further encapsulated within the data field of the current packet.

The protocol handling scheme that we chose to use in the PS-186 is located in the kernel software. By keeping all protocol wrapping and unwrapping inside the main single-

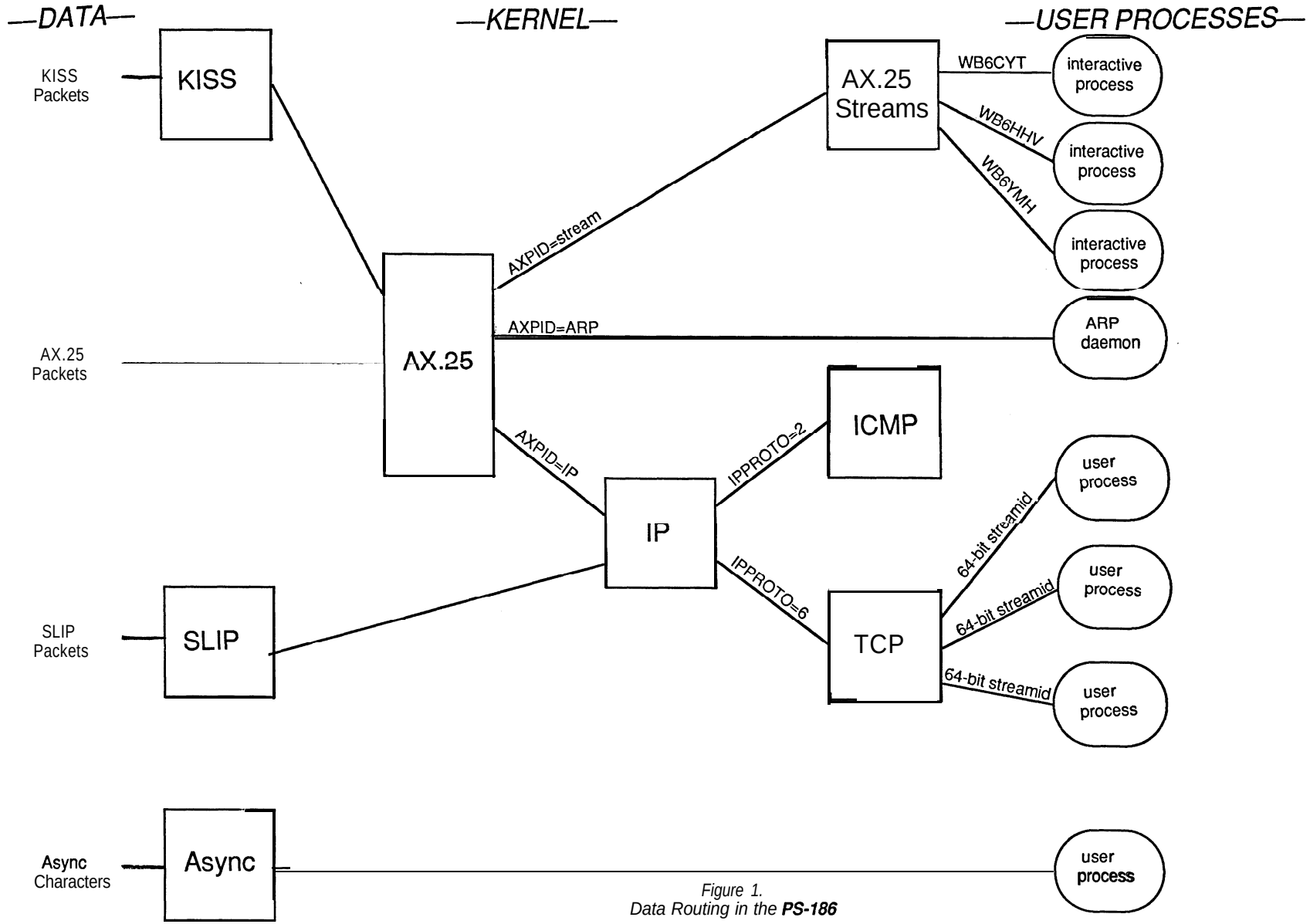

Figure 1.
Data Routing in the ***PS-186***

Showes the routes that data packets may take depending on protocols involved. Data is received as packets at left; as it passes through the various protocol handlers and switches it is sent to other protocol modules as required until finally it is fully unwrapped and available to the user processes at the right. Data to be sent follows essentially the reverse path from the user process through successive layers of encapsulation until it is ready to be sent. The criteria for determining further processing are shown on the lines connecting the various boxes.

thread portion of the operating system and thus making them available to all processes on the system, we simplify greatly the amount of protocol handling required in the various other processes.

Each PS-186 communications interface is configured at system startup time to handle one type of outermost protocol (for example, KISS AX.25 is appropriate for a serial interface to a radio link, or perhaps SLIP for a hardwire line.) As a packet is received from an interface, it is examined according to the rules that have been set up for that interface. When that packet has been received and the appropriate acknowledgements generated, the contents of the packet and selected fields extracted from its header are passed to the appropriate protocol switch. The protocol switch then examines the packet contents and routes the data further to the next protocol module, as appropriate. This process repeats until there is no further enclosed protocol, until the data has been fully extracted and is available in a buffer queue to be used by some user process. Not until the data has been fully extracted does it become ready to leave the kernel environment.

A concrete example may make this clearer: Suppose that we receive an AX.25 packet on a serial link that is attached to a radio. If it is for us (as shown by the destination callsign) we will acknowledge that AX.25 packet (if appropriate), and if it was a data packet (UI or I frame), we will pass it to the AX.25 protocol switch. That switch will examine the Protocol ID byte that is part of the AX.25 packet. If the PID is for a stream connection (a normal mode AX.25 connection such as is commonly in use today for keyboard-to-keyboard typing), then the packet will be further switched by the AX.25 Stream switch, which will send it (based on the callsign in the source field of the AX.25 header, since there can be only one connection per source callsign) to the user process that is servicing that stream connection.

If, instead, the PID is for ARP, RARP, RIP, or one of the other raw packet protocols, the data will be sent to the user process that handles that kind of packet - to build address or routing tables, for example.

A packet with the PID indicating an encapsulated IP packet is passed to the module that does IP protocol - checksums and other integrity checks. If the packet is ok by IP standards, the IP module will call the IP protocol switch, which will examine the Protocol ID byte in the IP packet (distinct from the PID in the AX.25 packet). This will in turn route the IP packet to another protocol handler, such as UDP, ICMP, RDP, or TCP - whichever we have implemented. Again, those are expected to route the data based on fields in the headers of these protocols.

TCP is an interesting example of imbedded protocols and switching. Each TCP connection as seen on a host is designated uniquely by a 64-bit number that is the concatenation of the distant host's Internet address (32 bits) and the local and distant TCP logical port numbers (16 bits each). Since when a TCP connection is initiated, the originating host must chose a new (not currently nor recently used) logical source port number, there can be multiple logical connections between, TCPs on the same two hosts even to the same distant port. The data switch in the receiving TCP is required to separate out the streams of data based upon the 64-bit stream identifier, and deliver each to potentially separate user processes as appropriate.

6. Process Control

The PS-186 is organized as a multi-tasking System, implying that more than one process may be running at a time. There is always a "null" process that is constantly ready to run; when there is no other process ready to run, the null process is active.

Processes are created as needed and destroyed when no longer needed. In this manner, resources are not consumed on idling processes that are merely sitting around waiting in case they are needed. For example, when an AX.25 connection is made to the PS-186 network node, a process is started to handle incoming stream data. This process will exit and its resources will be deallocated when the connection is closed. Each such connection will cause a separate process to be spawned.

User-level processes do not perform I/O operations to devices; they instead make *system* calls to the kernel that invoke the required I/O. When a process makes a system call that would require some time to complete (such as I/O), that process is *blocked* - that is, placed in suspended state, and another process is resumed. Periodically (in response to interrupts from the system real-tirne clock), the current process will be suspended and another selected to run. Thus no process can hog CPU resources, and I/O can proceed in parallel with ordinary processing.

We feel that multitasking is a superior method in this application, although it is much more complex than a single-threaded program, because much of what goes on in a device like the PS-186 is not time-critical, and we can therefore devote the CPU to high-priority events (such as the arrival and buffering of a packet) that are truly critical.

7. Conclusion

We feel that the PS-186 Advanced Packet Controller represents a significant step towards the construction of an efficient and practical amateur radio data network. By combining fast hardware and efficient software into a flexible package that accomodate today's and tomorrow's protocols, we believe we have advanced the network one step further along the road to completion.

8. References

[1] AT&T, "Communications Protocol Specification BX.25", Publication 54001 Issue 2 (June 1980)

[2] Comer, D., *Operating System Design – the XINU Approach,* Prentice-Hall (1984)

[3] Comer, D., *Operating System Design – Internetworking with XINU,* Prentice-Hall (1987)

[4] DEC/Intel/Xerox, "The Ethernet – A Local Area Network Data Link Layer and Physical Layer Specification.", Version 1 .O (Sep 30 1980)

[5] Fox, T. L., "AX.25 Amateur Packet-Radio Link-Layer Protocol", Version 2.0, ARRL (Oct 1984)

[6] Griffiths, Georgia, and G. Carlyle Stones, "The Tea-Leaf Reader Algorithm: An Efficient Implementation of CRC-1 6 and CRC-32", Communications of the ACM, 30,7 (July 1987)

[7] IEEE, *Logical Link Control,* ANSI/IEEE Std 802.2-1985 (1984)

[8] Postel, J. et al., "DDN Protocol Handbook", USC-ISI (1986)

[9] Tanenbaum, A., *Computer Networks,* Prentice-Hall (1981)

[10] Tanenbaum, A., *Operating Systems – Design and Implementation,* Prentice-Hall (1987)