PACKET SWITCH SOFTWARE DESIGN CONSIDERATIONS

Terry Fox, WB4JFI
President, AMRAD
1819 Anderson Rd.
Falls Church, VA 22043

Abstract

Toward the end of 1985, a new device showed up on Amateur Packet Radio, the packet switch. Soon, the packet switch will be replacing digipeaters around the country, giving more reliable (if slower) operation of the overall network. The first switches have been based on TAPR TNC-2 hardware, and therefore are somewhat limited. An accompanying paper describes what types of hardware the author sees being used in the future (both near and down the road) for the switches. This paper will explain the author's view of how the switch software should be organized.

This paper will not provide actual switch code, but rather indicate where progress is being made, and where help is needed. Also, wherever possible, I cite Protocols and Standards I believe should be implemented.

Types Of Packet Switches

Before we delve into the meat of the software, inside the switch, we must decide what the switch's function(s) are. I see these functions falling into two catagories, with one having two sub-catagories; the transit switch, and the end-point switch.

The first type is called the transit switch because it is used to pass network connections through it, but not necessarily act as an end-point of a network connection, except for maintenance functions. The transit switch can be thought of as a software "patch-panel" type of device. This means that packets coming in from one adjacent switch station will pass through the transit switch to another adjacent switch. The length of time the "patch cord" is plugged in, allowing the connection, depends on the type of networking protocols being used. It can vary from a very short period (the single packet) in a dynamic-routing datagram network, to almost forever in a permanent virtual circuit case. Normally, the patch is maintained for the length of the end-to-end Network connection in virtual circuit networks. The transit switch may also perform a similar function as today's remote location digipeaters, that of interconnecting local networks to form a larger network.

Transit switches may have special hardware needs, since they will most likely be placed at remote locations far away from network users. Among these hardware needs will be low-power operation and redundancy of key parts for survivability. Additional software may be required to take advantage of this hardware.

The other type of packet switch is the end-point switch. While it may also perform the transit switch duties mentioned above, it will also have software that allows the end-users of connections to interface to it. This is where a typical Amateur station will put his packets "into the pipe':) and also where the packets come "out of the pipe" at the other end. In our patch panel scenario mentioned above, the end-point switch is analagous to the last patch panel that has the equipment connected to it. In order to perserve the capability of using older packet systems that may be capable of only level 2 connections, there are potentially two different types of network interfaces in the end-point packet switch.

If the end-user is an Amateur that has a newer packet board (called a PAD) with true networking code in it, Amateur A can ask for a Network Layer connection from the switch. The switch, understanding that a PAD is requesting connection, will automatically provide the initial link and network connections, and proceed to attempt to place the requested call to the destination station.

If, however, the end-user is an Amateur using a TAPR TNC-1 with level 2 (link) only software, there must be some additional method of providing a network interface for the user. Typically, this will be through an additional program, which I will call the network access PAD routine. This network access PAD routine will run as an application layer program inside the end-point switch, and will provide all the necessary network interface magic for the Link-only Amateur.

Let me now describe how I think the switch software might be organized.

Top-Down Design of Switch Software

One reason for this paper is to indicate how we in AMRAD are working on the switch software. Up to now, most packet systems have been designed by first putting together some hardware, then writing low-level drivers for that hardware, then writing higher level code to properly tweak the low level drivers, etc. While this building-block approach does indeed work, it can lead to problems if some unanticipated problems occur. It also tends to be difficult for more than one person to work on a project using this approach. AMRAD has started working on the packet switch using more of a top-down design approach. While what we are doing may not be exactly top-down, it does follow that philosophy.

We are studying the various aspects of the switch BEFORE writing a line of code. This includes how the various processes inside the switch interoperate, and how to make the best use of the hardware supplied.

As part of this study, we quickly came to the conclusion that in order for the different software "machines" of the switch to work together properly, some sort of operating system should be implemented. This operating system need not be as complicated as a disk operating system, but should provide the necessary tools for the different processes to intercommunicate. It should also provide a common set of utilities for access to shared switch resources, such as the memory or buffer pool, and timer subroutines. I will pursue the operating system in more detail shortly.

We decided to follow the IS0 Open Systems Interconnection Reference Model (OSI-RM) when splitting up the task of designing our switch. This gave us some rather clear divisions of responsibility, both for dividing our labor pool for the design work, and for what software is responsible for accomplishing which sub-task of the overall switch.

I feel that most sub-tasks should have two distinct parts; the operational code, and a management section that errors are reported to and new requests of that sub-task are coordinated through. This will lead to more orderly code at each sub-task, and also provide a better form of sub-task intercommunications.

Some of the sub-tasks of a typical Amateur packet switch are as follows:

A. Operating System (including resource management).
B. Maintenance Application of switch.
C. Message Authentication for maintenance.
D. Database management (User and Routing).
E. Network Access PAD Application (if available).
F. Session Layer Protocol for internal switch applications.
G. Transport Layer Protocol Machine.
H. Network Layer Gateway Machine (if needed).
I. Network Layer Addressing and Routing routines.
J. Network Layer Protocol Machine.
K. Link Layer Protocol Machine.
L. Physical Layer Software Support.

Each of these sub-tasks will now be discussed individually.

Operating System

A typical packet switch will need to perform several different processes, seemingly at once. One method of accomplishing this would be to use a separate microprocessor for each sub-task. While this might make the writing of software easier, it is generally considered a waste of processing power, costs muchmore, and takes more room and power. Until this method of dividing the overall switch task becomes necessary (due to single processor overload), another method can be used more effectively. Someday, as RF channel speeds go up and as more RF channels are used per switch, the use of multi-processors will become necessary.

One microprocessor should be able to perform all the necessary functions of today's packet switch if some method of dividing up it's processing time is devised. This division of time can be done in many ways. One of the most common methods is to have the hardware provide a timed interrupt, which the processor uses to indicate it's time to switch tasks. This way, each task is allowed a certain time slice to perform as much of its duties as possible. Another method of dividing the processor time is to let each task run to completion before switching to another task. There are also various combinations of these two task switching methods. The method of switching, along with the necessary support required to make this task switching transparent to the tasks is one responsibility of an operating system.

Almost all processes in the packet switch will need to communicate with at least two other processes, both to pass information and to request services from the other process. The traditional method to do this is to call a subroutine (or function), passing a pointer to the information to be tranferred, or the service request. Each process may be written at a different time, or by a different person. This sometimes leads to different interprocess interfaces, making the overall system design more complicated. A standard, predefined set of routines to provide information passing and service requests would aid in the designing of the switch. This is another service that is often provided by an operating system.

Still another service that would be frequently requested would be a common set of subroutines to provide memory and buffer management. The packet switch is basically a message transfer machine, with the messages being the packets. These messages are usually maintained inside the switch in buffers. The requesting for, and the freeing of buffers, along with the passing of buffers and accessing of data within the buffers, can easily be standardized into a common set of subroutines. This set of subroutines lumped together is called buffer management. Buffers are made from computer memory, and this memory is usually controlled by another set of subroutines, called memory management. Since buffer and memory management routines are services used by almost all processes, they are normally considered part of the operating system.

The packet switch has several processes that rely on the use of timers for error detection and recovery. In more advanced systems, such as the packet switch, software timing loops are not considered good form, since they occupy too much processor time, essentially putting to sleep all other processes just to keep time. The suggested alternative is to use hardware timers with software support for those timers. Since timers are requested by various processes, it makes sense to provide timer access as an additional service by the operating system.

Noting that the use of an operating system that provided the above support would be beneficial both to those of us writing the switch code now and to those writing additional code for the switch or modifying our switch code, we started looking at available operating systems. Our prerequisites included that the operating system be either free or inexpensive, preferrably written in a higher level language for software portability, and be efficient at its appointed jobs.

Our present packet switch hardware is based on Intel microprocessors (the 8080 and 8088 families), at first it looked like iRMX from Intel might be a good start. It had all the necessary support capability mentioned above, along with a lot more. Upon careful analysis however, it began to look like iRMX would take too long to perform most of the requested tasks.

About the same time we decided iRMX wouldn't fill the bill, Mike O'Dell started talking up a different type of operating system, called the HUB. It was designed to be more efficient in message based applications, especially in packet type devices. The more he talked, the better it sounded. HUB does not have a lot of fancy stuff, such as processes interrupting other processes, but for our application we don't need that fancy stuff, which nine times out of ten gets in the way and takes valuable processing time. Mike was so convinced that the HUB was the right way to go that he wrote the code for us in C. I have been able to compile his C code on an IBM-PC and on a Xerox 820. We are going to use the HUB operating system as the basis for our packet switch, as soon as we get a more thorough understanding of it. Mike has written an introductory paper on the HUB which is elsewhere in these proceedings. I feel this HUB will become a very important part of our packet switches, since it provides a stable, working interface for the rest of the tasks to use.

Eventually, the operating system should probably be written in assembler, since it will be the most commonly used software in the switch.

Maintenance Application Task in the Switch

Now that we have an operating system running in the switch, there must be some tasks that the switch needs to perform. One task might be the maintenance of the switch itself. This task could be divided into two sub-tasks; minor "tweaking" of variables for more efficient switch operation, and recovery after some type of malfunction. While the various protocols might be capable of doing minor tweaking of their own variables, both of these sub-tasks should be available to an Amateur remote from the switch. This means an Amateur could establish a connection to the switch, indicate the purpose is to perform remote maintenance of the switch, pass through some authorization door, and then perform the maintenance of the switch, as long as the switch is operating.

Among the maintenance functions that should be available are:

A. Take the switch completely off the air.
B. Re-boot the switch from mass-storage.
C. Upload new switch code or portion of switch code over the packet channel.
D. List operating parameters of the tasks.
E. List status of connections to switch.
F. List stations and switches heard lately.
G. Gain access to User and Routing databases for listing or updating.
H. Modify operating parameters of switch.
I. Monitor and report switch operation.
J. Dump error reports since last maintenance connection.

If a switch starts malfunctioning, there should be a way to gain access to it, assuming it has not totally failed. Each Physical channel connected to the switch should be allowed to have a connection established through it to gain access to the Maintenance Application program. This way, if one Physical channel is tied up or broken, it could be turned off without removing the whole switch from the network. If however, the switch has some major flaw that prevents proper long-term operation it may be better to totally disable the whole switch until on-site repairs can be performed.

There also needs to be a method of either uploading new switch code, or updating a database from a remote location. The best way to do this is to store the uploaded information on a mass-storage device, then re-boot the switch after the upload has been successfully completed. Modifying portions of the switch code "on-the-fly" could prove disasterous, and should be avoided.

Another maintenance function would be to monitor switch operation. This can be done in real-time (establish a maintenance connection, then let the Maintenance Application program periodically dump status information) or by accumulating the status information in variables or a database for later retrieval.

The maintenance process described above assumes there is not a separate Service Processor connected to the switch, or the Service Processor is not available for some reason. If there is a Service Processor available, most maintenance functions will probably be performed through it, in addition to others that the actual switch could not perform and still operate normally.

Remote maintenance of switches will be a necessary function that will be improved upon as a history of switch operation is documented. There is no way we can predetermine all the maintenance requirements before some actual operating time is logged. This fine-tuning of switch maintenance will be especially important for remotely placed switches, where access may be difficult or impossible for long periods of time.

Since the maintenance application does not have any major speed constraints, it could be written in a higher level language such as C for portability.

The Maintenance Application process will receive its data from one of two sources; a local console if the maintenance is being dons on-site, or from the Message Authentication process if the maintenance requests are coming over a packet channel. The Maintenance Application process should be able request status of each switch process, and control certain aspects of each switch process. The access to each process will normally be through a management routine within that process.

Message Authentication for Maintenance

Since the maintenance application involves direct control of the packet switch internals, some method of allowing certain Amateurs access while preventing other Amateurs access must be employed. Access control schemes such as simple passwords or control codes are not adequate, since any Amateur listening to the packet channel can gain knowledge of them. Since packet radio is a digital communications method, a way of dynamically digitally encoding both the access request and the actual commands sent to the switch can be easily accomplished. Paul Newland presented a paper at the 1985 ARRL Computer Networking Conference describing such a scheme. Coincidentally, Hal Feinstein came up with a very similar scheme at about the same time. Hal has been working on this over the last year, and has written a paper on his activities, found elsewhere in these proceedings. He is also writing the code to perform the message authentication he describes.

The Message Authentication process does not require real fast execution time, so it could be written in a higher level language, such as C, for portability between different types of switches.

The Message Authentication process interfaces between the Maintenance Application process, and the Session Layer Protocol machine, or the Transport Layer Protocol machine if there is no Session Layer.

Database Management

Packet switches will need to know where to route the packets they receive. This routing will be based on information the switch maintains on how the network is organized. Since this information should be quickly accessable, an orderly method of storing it should be used. Also, routing information will change periodically, due to switches going up and down. In order to keep the routing information accurate, a small database manager should be used.

Another function a database manager could provide for end-point switches is to keep a list of Amateur stations it normally services. This way, when a request for one of its users comes in, it will recognize that it should respond, or if the user has indicated that all calls for it should be forwarded to another switch, the switch can pass the new destination end-point switch address back to the source end-point switch.

This database manager does not need to be as sophisticated as a commercial program. A relatively small database program can support these two databases, along with any other databases that might be used by the switch (such as maintenance records). The database program could be written in a higher level language, such as C with no ill effects.

Network Access PAD Application

As mentioned earlier, end-point switches will need to provide two different network access methods to the Amateur. If the Amateur has Network Layer capability it can request connections directly through the switch. If the Amateur only has Link Layer connection capability (such as most present TNC boards have), the switch will have to perform some additional functions for the Amateur.

One method of providing this service would be whenever a Level 2 only TNC requests connection to the switch (note the connection is TO the switch, not thru it as a digipeater is presently used), the switch would accept the Link Layer connection. This Link only connection indicates that the user cannot support network protocols, and the switch then places a request to the Network Access PAD code for assistance. The Network Access PAD routine might then send the user a menu of functions that the net access PAD code can provide, such as:

1. Connect to another Local Amateur.
2. List All Local Amateurs on this Switch.
3. Look up a Remote Amateur's Address.
4. Calculate path to Renote Amateur.
5. Connect to Remote Amateur via supplied path.
6. Connect to Remote Amateur via path implied by supplying Destination Switch Address.
7. Connect to Remote Switch to Monitor it's the remote channel.
8. Add this Amateur station to Switch User directory.
9. Delete this Amateur station from Switch User directory.
10. Indicate an alternate path for this Amateur station (station will be mobile).

The above list is-not meant to be a final version of a menu, but does indicate some of the functions the network access PAD routine should provide.

The user then selects the function or functions to be performed, and the switch takes care of doing the actual work. If for example, the user requests a network connection to another Amateur on the same switch with Network Layer capability. The switch will then request a network connection to the destination Amateur on behalf of the first Amateur. If the network connection is successful, the switch will handle the Network Layer protocol machine for the

Amateur, using the Link Layer connection to provide data integrity between itself and the first Amateur.

One point here is that a lot of additional work will be done inside the switch to provide this service to Link-only Amateurs. If two Link-only Amateurs wish to communicate through a switch, it may be better if they connect directly to each other, using the packet switch in Level 2 digipeater mode. Digipeating is generally not a good idea once true networking arrives, but in some cases it may be the most efficient method of communicating. A link using only one digipeater should be stable enough to provide reliable communications, and the loss of efficiency due to digipeating may be made up by the reduced amount of overhead the Network Layer would otherwise add.

One type of Network Access PAD interface will be based on the CCITT X.28/X.29/X.3 set of standards, commonly referred to as the triplex protocols. X.28 defines the user-to-network PAD interface, X.29 defines the network-to-PAD interface, and X.3 defines the variables used in the PAD, along with some common settings of these variables.

Dave Borden has written a paper on the User interface which can be found elsewhere in these proceedings. The user interface is another task that can be written in a higher level language such as C.

Session Layer Protocol for Internal Switch Operations

The Session Layer is used to multiplex more than one data stream to the upper layers over a single transport/network connection. This may be necessary in the switch for maintenance and for Link Layer only connections. An example of the latter would be if a Link Only TNC requests more than one network connection, or if the TNC requests both a network connection and some other Network Access PAD function simultaneously.

I am leaning toward the use of a sub-set of the CCITT X.225 Session Layer protocol for starters. This protocol is more than adequate for use by the packet switch. The use of a higher level language for the Session Layer Protocol in the switch would not pose any major problems.

Transport Layer Protocol Machine

The Transport Layer is responsible for absolute data integrity across the network connections. While the Network Layer makes the individual connections between switches, the Transport Layer provides a logical connection between the two end-points (source and destination). Different Network Layer Protocols place different requirements on the Transport Layer Protocol.

A network made up of datagram type switches will need a more complicated Transport Layer Protocol machine, partially because datagram switches do not maintain a "connection" between each other, and therefore do not have ANY error recovery (that's recovery, not detection) procedures operating between themselves. Also, the only real recourse a datagram switch has when detecting an error is to throw the offending packet in the garbage queue and hope it is retransmitted.

A network based on virtual-circuit type switches will need little, if any, Transport Layer Protocol machine. Since individual logical connections are maintained between switches involved in a network connection, these individual logical connections will detect and correct almost all errors incurred along the network connection. The only two error conditions that the individual logical connections between switches won't ALWAYS correct for properly is a total transit switch failure somewhere along the network connection, and data corruption inside a switch, most likely due to partial memory failures.

For the above stated reasons, end-point switches may want to employ some form of a Transport Layer Protocol machine on connections where absolute data integrity is necessary. At the 1985 ARRL Computer Network, I proposed the Amateur community adopt the use of the CCITT X.224 Transport Protocol network-wide, both for virtual circuit networks (AX.25 types) and datagram networks (the TCP/IP crowd). This Transport Layer actually defines five different classes of Transport Layer Protocols, with negotiation of classes allowed at the Transport Layer connection establishment. The various classes provide different forms of end-point error detection and correction. Interested readers should refer to the 1985 ARRL Computer Networking Conference proceedings.

The important part of the X.224 Transport Layer Protocol is that with the various classes defined, a switch can request only the amount of overhead necessary, without having to live with a lot of excess baggage. The datagram-based network will need to use the Class 4 protocol, which has all the bells and whistles. If a Transport Layer Protocol machine is deemed necessary in a virtual circuit network, Class 1 should be sufficient, especially if the checksum option is implemented.

The X.224 Class 1 machine does very little, as far as Transport Protocols go. It starts by establishing an end-point logical connection between the two end-pont devices (normally the two end-point switches). It then relies on the use of state variables and timers to detect and recover from errors, just like Link and virtual circuit Network Layer protocols. The difference is these state variables are maintained end-to-end ONLY, they are not affected by individual Link or Network Layer connections. If an intermediate switch in a network connection fails, the Transport Layer Protocol machine will eventually detect it, due to timer failures. The Transport Layer Protocol machine will then attempt to re-establish the network connection from the source end-point to the destination end-point, without any User Amateur intervention, unless requested. Lost and duplicates packets will be detected by their wrong sequence numbers. This corrects for all but one problem, data being mangled inside a switch.

The CCITT X.224 Transport Protocol has an option to append a checksum to all data packets. Using this option, data integrity can be assured. Since the Transport machine is end-to-end, the checksum is also end-to-end. This means the checksum needs to be calculated only at the two ends, not at every intermediate switch.

The Amateurs at each end of the network connection may not have the X.224 Transport Protocol in their PADs for a while, so the switch will have to provide the Transport Protocol machine. This is the way most commercial virtual circuit networks operate. The end user accesses the network via an X.25 connection. The network then attempts to make the requested network connection using a Transport Layer protocol for end-to-end data integrity, and X.75 or a similar Network Layer Protocol. X.75 is basically the same as X.25 except it operates in a "balanced" mode, while X.25 is more of a master/slave protocol. The X.25 user-to-end-point-switch connection maintains proper data flow from the user to the network, the X.75 connections maintain proper data flow between the switches involved in the network connection, and the Transport Layer connection makes sure that the data traversed the whole network connection properly.

Keep in mind that a network of X.25/X.75 Network Layer connections may be reliable enough for most communications, allowing the Transport Layer machine to be developed and refined while the Network Layer is "on-the-air". This will allow a study of exactly how much of a Transport Layer Protocol needs to be used to back-up the Network Layer connections.

Another requirement that may be imposed on the Transport Layer machine would be that of "gatewaying" between different Transport Layer Protocols. The best way to take care of this potential problem would be to negotiate to the proper class of Transport Protocol to be used from the outset of a network connection. If a Transport Protocol is implemented that does not allow this negotiation (such as TCP), there may need to be a Transport Layer gateway at the interface switch between the two incompatible Transport Protocols. Technically, this violates the ISO Reference Model, since the Transport Layer Protocol each end-point sees is not the same and therefore the information at each end-point may not be valid across the entire network connection. Still, if the Transport Layer gateway is written carefully this can be a viable alternative.

The Transport Layer Protocol maching should be written in a higher level language such as C since it may need to be ported to different types of microprocessors.

Network Layer Gateway machine

Just as there may be a need for a Transport Layer gateway, there may also be a need for a Network Layer gateway. This process would be able to transform packets from one Network Layer protocol to that of a different type, making each side of the gateway appear to be working with a device of the same type protocol. This Network Layer gateway will most likely be easier to write than the Transport Layer gateway, since the change would not necessarily have end-to-end repercussions. This task is made even easier if the Transport Protocol has been negotiated to an agreed upon class, eliminating the need for Transport gateways.

The Network Layer gateway machine could be written in a higher level language for portability.

Network Layer Addressing and Routing Issues

There has been a lot of lively discussion over the last year regarding the Network Layer Protocols of choice. In addition to that basic discussion there are two other related subjects that invoke an equally lively debate. Those two subjects are what network addressing 'scheme is to be used, and how routing information is to be stored and how routes are to be determined.

When I first suggested the use of the Amateur callsigns as the addresses for the Link Layer of AX.25, it seemed like a natural. The best part of using the callsign is that they have been pre-assigned by the FCC., totally eliminating the need for some organization to assign addresses, which was the case for the older SDLC protocols used. I believe this is still the case, even for the upper-layer protocols. However, the Amateur callsign alone may not present enough information to "help" a network connection along.

Several additional addressing schemes have been devised by the Amateur community over the last year. Fortunately, most of these do not require an organization or groups of organizations to assign network addresses. Some of the more common addressing schemes are:

A. Callsign Only.
B. Callsign Plus Area Code/Phone Number.
C. Callsign Plus Airport Designators.
D. Callsign Plus Zip-Codes (Zip Plus 4?).
E. Callsign Plus Latitude and Longitude.
F. Callsign Plus Gridsquares.
G. Assigned Numbers by a heiarchical group of organizations.

Some of the people in the commercial network world warn against using addressing schemes that include or imply routing information. I feel that while we are building the Amateur Packet Network, and until our switches are sophisticated enough to contain all the routing information necessary to route packets without any other help, we may need some routing information included in the network addresses. This is why I suggested the use of callsigns plus gridsquares in my AX.25 Level Three proposal in the Third ARRL Computer Networking Proceedings. The suggested method there was to put the Amateur callsigns in the normal AX.25 address fields, and add the gridsquare information as optional facilities. This way, the callsigns will always be there, but the additional gridsquare information can be dropped when it is no longer needed.

Meanwhile, the first actual implementation of AX.25 Level Three has recently come out, and it supports callsigns plus area codes and phone numbers. Both of these systems do implicitly carry routing information, since they both carry a recognizable method of identifying where the destination station is physically located. It is beyond this paper to indicate which is better, however both are addressed in additional papers presented elsewhere in these proceedings.

The point I want to emphasize is that one reason why AX.25 Level 2 has been so successful is that the addressing issue was resolved without creating a bureaucracy to maintain records of who was assigned what address. I feel this is equally important at the Network Layer.

The routing issue is another area of great debate. Not only is the actual route determination an issue, but how the routing information is stored is also an issue.

Eric Scace, K3NA has been working on a routing algorithm that is self-generating. Whenever a packet switch comes on the air, it notifies other packet switches it hears that it is now available. All packet switches it notifies then add it to their routing database, modifying any routes that can now be run through it. All the switches involved then pass the new routes they have come up with to each other. The last step is to erase any duplicate routes in the database. This process can take a while, and since the whole routes are kept in the database, it can conceivably grow rather large. A method of shrinking this database size is to tokenize the various paths, and then expand the tokens whenever a route is looked up. The problem with most methods of shrinking the Routing database is that the routes must be expanded back whenever they are accessed, either for look-up or for database modification. This can add time to the look-up, which may be a worse situation than the large table size.

Routing is one subject that will need more work. For now, we may be better off letting the User specify the route (explicit routing). As the Amateur Network grows, a better feel for the automatic routing algorithms will emerge.

Routing algorithms could be written in a higher level language, especially ones to be used in virtual circuit switches, since they are accessed only once (during network connection setup).

Network Layer Protocol Machine

Since I am one of the prime pushers of virtual circuits, it should come as no surprise that AX.25 Level Three is my choice for the Network Layer Protocol. I feel that since virtual circuits tend to catch and correct errors when they occur, less overall network facilities are wasted correcting these errors.

The Network Layer Protocol machine should be capable of accepting multiple network connections, from a variety of other switches. Network connections that end at the switch should be passed on to the proper higher layer protocol machine. First, if a Transport Layer Protocol machine is involved, the data will be passed to it. If the switch itself is the destination end-point, the data will then be passed to the Session Layer Protocol machine for further processing. If the destination station is a Link Layer only TNC, the data must go to the Network Access PAD program for Layer 3 processing before being transferred to the Link Only TNC. If the destination station has an AX.25 Network Layer connection to the switch, the data will be passed directly to the destination station PAD.

The Network Layer Protocol machine could be written in a higher level language. The language of choice for the switch seems to be C. I see no time constraints of the Protocol requiring that the Network machine be written in assembler. The Link Layer affords enough of a time buffer.

A Network Layer management routine should be employed as an interface between the network machine and the rest of the packet switch. This management routine will monitor the Network Layer Protocol machine operation, and make minor adjustments to certain variables. Recoverable errors may also be reported to the Network Layer Protocol machine management in order to keep a record of malfunctions.

Link Layer Protocol Machine

The Link Layer Protocol machine uses AX.25 Level 2 as the Link protocol, both between the switch and the individual Amateurs, and between the switches, running under the Network Layer connections. Since there will be many devices trying to connect to the switch, the Link Layer Protocol machine must be capable of multiple Link connections, possibly coming from more than one Physical channel.

The Link Layer Protocol machine will send its received data (after it processes it) either to the Network Layer Protocol machine, or to the Network Access PAD if the source of the data is a Link Layer only TNC.

Even though packet activity operates over half-duplex channels right now, the Link Layer Protocol machine should be able to operate full-duplex. This will allow easier testing of the code now, and the addition of full-duplex channels running at faster speeds, which may be available in the not-too-distant future.

The Link Layer Protocol machine should have a separate management section that is capable of monitoring Link Protocol machine operation. This management section should be able to report the status of the Link machine to upper layers, fill requests from the upper layers for additional Link connections, and control certain Link variables to optimize Link Layer operation. In addition, these same variables should be adjustable from the Maintenance Application routines.

Even though the packet switch will essentially replace the digipeaters, there may be a need to keep the digipeater code in the switch. One case is mentioned above, that of operation between two Link-only TNC devices. The cost is nearly nothing (since digipeating is a very simple function) to keep the function active.

The Link Layer Protocol machine could be written in a higher level language, such as C, or it can be written in assembler. I personally feel that since the Link Layer is speed dependent, it should eventually be written in assembler. This way processing time in the Link Protocol machine will not seriously affect the Link Layer Operation, particularly in the areas of time-outs. This will become especially true as the speed of the Physical channels increases.

Physical Layer Support Software

The Physical Layer support software is the packet switches interface to the "real world". It is what supports the sending and receiving of packets over the Physical channels. Since this software directly supports the hardware of a Physical HDLC channel, I feel it should be written in assembler for speed. The software has to be written specifically for the hardware device anyway, so the assembler requirement is not out of line. The end result is that whole frames should be passed between the Physical Layer support software and the Link Layer Protocol machine. These frames should also indicate which channel they were received on.

While the Physical Layer support software will need to be tailored to the type of HDLC channel used, it should be able to run full-duplex for debugging, even for half-duplex RF channels.

The Physical Layer management should be able to control certain variables such as timeouts and channel speeds. In addition, access to the same variables should be available through the Maintenance Application.

Conclusions

AMRAD will be working on the next generation of packet switch software based largely on the ideas presented in this paper. We plan to have the switch code running first on a PC compatible computer, with smaller versions of the code running on the Xerox 820 and TAPR NNC systems. We feel that designing the overall switch software first, then dividing the project up fo actual coding may take slightly longer to start, but the resulting system will more than make up for the initial lag. It should work more efficiently, and should be easier to understand and interface to.

I am looking forward to reporting next year on the status of this ongoing project.

References

Fox, T.,	"User and Switch Packet Digital Hardware," Fifth ARRL Amateur Radio Computer Networking Conference, ARRL, 1986.
O'Dell, M.,	"An Introduction to the HUB Operating System"., Fifth ARRL Amateur Radio Computer Networking Conference, ARRL, 1986.
Newland, P.,	"A Few Thoughts on User Verification Within a Party-Line Network", Fourth ARRL Amateur Radio Computer Networking Conference, ARRL, 1985
Feinstein, H.,	"Authentication of the Packet Radio Switch Control Link". Fifth ARRL Amateur Radio Computer Networking Conference, ARRL, 1986
Borden, D.,	"The Network User Interface", Fifth ARRL Amateur Radio Computer Networking Conference, ARRL, 1986
Fox, T.,	"CCITT X.224 Transport Layer Protocol Basic Description", Fourth ARRL Amateur Radio Computer Networking Conference, ARRL, 1985
Fox, T.,	"AX.25 Network Sublayer Protocol Description", Third ARRL Amateur Radio Computer Networking Conference, ARRL, 1984
Fox, T.,	"Annex A Through F For AX.25 Level 3 Protocol", Third ARRL Amateur Radio Computer Networking Conference, ARRL, 1984
Fox, T.,	"Amateur Routing and Addressing", Fifth ARRL Amateur Radio Computer Networking Conference, ARRL, 1986