# DESIGN DECISIONS FOR THE TAPR TNC LINK LEVEL

David Henderson, KD4NL
2621 W. 164th St.
Torrance, CA 90504

## Abstract

The decisions that were made up front on the software side of the TAPR project have had a very strong impact on the implementation and success of that project. Following is a review of some the design decisions that were made long before coding started, and a chronicle their impact upon implementation and performance.

## Introduction

Before the software description starts, lets run through a quick overview of the hardware. The microprocessor is a Motorola 6809; the memory complement is 24 kilobytes of EPROM, 6 kilobytes of RAM, and 32 bytes of electrically erasable ROM. The peripheral chips consist of a Mostek 6551 asynchronous interface adapter, and a Western Digital 1933 HDLC interface chip. There is also an Mostek 6522 for clock support and onboard parallel I/O. The potential of using a parallel port for terminal I/O is provided with a Motorola 6820 parallel port chip. The computer system used as a software factory for the onboard software was an HP-64000 microcomputer development system present at the University of Arizona. This system made possible the installation of the TNC software on the TNC hardware.

The entire software development cycle was focused on X.25. The AT&T BX.25 document was taken as a reference for LAPB (reference 1), and the AMSAT document (reference 2) was taken as a reference for the header construction. The transition tables in the BX.25 document were followed very closely for the connect/disconnect sequences, but the I-frame manipulation was implemented in other ways, as described in more detail later on.

## Architecture

The number one design choice was to write as much of the TAPR code in Pascal as possible. Pascal was chosen because it is a widely available high level language, and the existence of sophisticated compile time options for debugging implied the Pascal checkout could be started either on a big timesharing system or a microcomputer Pascal system.

The choice which influenced the rest of the implementation the most was to have the high level Pascal code sit in a tight loop checking flags that are continually being set and reset by interrupt code written in assembly language. This design decision paved the way for implementation and checkout of the Pascal source without having to have the hardware actually running, and greatly simplified the design of the Pascal code since it did not have to deal with interrupts. There was also a complicating factor in that flags are continually being set and reset to schedule future actions in the Pascal code; that is, the main loop of code was a sequence of IF and CASE statements that check these important flags. The action taken by the code is not immediately apparent upon reading the code; you have to know the meaning of the flags being manipulated,.

The next design hurdle was the buffering; how many buffers should there be and what mechanisms were to be used to move data from one buffer to another? The main task, being a TNC, needed only two buffers, one for the incoming data flow and one for the outgoing data flow, and it was thought at one time that only two buffers would be needed. Once digipeating and control functions were considered, it was clear that four buffers were needed - there had to be a way to shuttle HDLC input data to the HDLC output queue and a way of talking terminal input data for commands, acting upon the commands, then printing a response to the terminal. The software was then broken up into four distinct sections; each section moves data from one buffer to another, with the possible side effects such as parameter changes. One section moved data from the HDLC input ring to the terminal output ring, another moves data from the terminal input ring to the HDLC output ring, a third took data from the terminal input ring and produced status messages in the terminal output ring, and the digipeating process move HDLC frames from the HDLC input ring to the HDLC output ring.

The next choice was to have the software 'know' as little as possible about the half duplex nature of the radio link. Previous implementations such as Vancouver Area Digital Communications Group protocol (VADCG for short) had used the poll/final bit in messages to turn the link around and have the receiving side turn into the transmitting side. I did not fully comprehend the use of the poll/final bit in this manner, and this use of the poll/final bit certainly conflicted with the LAPB usage. The design decision chosen for this "when do I send" problem can be simply stated: Only acknowledge receipt of messages or send messages when the modem signal data carrier detect is not present, or the data carrier detect signal has dropped at least once since receipt of a message. This rule means that message acknowledgements will be generated once for every sequence of information frames, and that there is a break in the received traffic before any new messages are queued up for transmission.

## Consequences of the architecture

The Pascal implementation on the HP-64000 development system was not a full ISO standard Pascal. The major difference dealt with character strings; the HP Pascal had a STRING data type whereas the IS0 standard has only arrays of characters. Unfortunately when the HP system writers adopted the STRING data type, they threw out completely any compatibility with ISO standard programs - character strings longer than one byte could not be used. This problem was 'solved' by including all character strings into one area of memory in EPROM. All references to constant strings had to reference the name of the array containing the data in this read only data area.

The Pascal code was checked out on two different computer systems prior to being installed on the TAPR board. The systems were a 36 bit mainframe and an 8 bit micro. The 36 bit mainframe checkout used routines to loop back HDLC input and output together in software; this allowed the software to connect to itself and allowed the basic logic to be checked out; all of this checkout was greatly speeded up via the symbolic debugger on the mainframe, The 8 bit micro allowed on the air tests with VADCG boards, and was invaluable in shaking down more basic logic problems. Again the routines to interface to terminal and HDLC I/O had to be changed to reflect the routines existing in the 8 bit micro system, but this is a pretty easy task to accomplish, The result of this staged checkout was a very robust implementation of X.25. . There were bugs in the Pascal code, but they were only evident under extreme conditions.

## Implementation of the architecture

There are four streams of data flow: Two directions for HDLC I/O and two directions for terminal I/O. Knowing when there is data present in the input streams and when there is enough room in the buffers for more characters in the output streams is handled by global variables. These variables are generally set by interrupt routines and reset when low level routines called by Pascal manipulate data associated by the flags. Each data stream has its own peculiarities, and the flag checking/clearing activity associated with each peculiarity will be covered.

The first stream of data is the HDLC input stream. Here a global variable exists which always contains the HDLC input top frame size, and is zero if there are no HDLC input frames placed into the HDLC input buffer and not yet processed by the Pascal program. The portion of the Pascal code that handles HDLC input notices that the variable is nonzero, and calls a low level routine which will move the HDLC data from the input ring buffer into a private Pascal buffer for further examination.

For HDLC output, there is a global variable which contains the number of free bytes in the HDLC output ring buffer, This cell is checked before any HDLC output frame is generated, and if there is not enough room in the HDLC output ring buffer for the potential output frame, then the generation ofthe output frame is deferred by simply not clearing the flags that cause the output frame to be generated. In the case of digipeated frames, if there is not enough room in the HDLC output ring buffer for the digipeated frame, then the digipeated frame is simply forgotten.

Terminal output is similar to HDLC output, but there are differences. There is a global variable which contains the number of bytes currently available in the output ring buffer. The routine called to queue up terminal data will always wait for space available in the output ring 'buffer to queue the character, The purpose in having the variable output ring free byte count is to avoid waiting for buffer space when X-frames come in on the HDLC input port. When an I-frame comes in that has a data portion too big to buffer, the data in the frame are ignored and the HDLC message "RNR" is scheduled for future transmission. This async output routine is freely callable from anywhere within the Pascal code, and does get called as a part of parameter displays, hex dumps., and internal debug routines, It was decided that when any of these activities were going on, no one would care if the Pascal code was spinning while waiting for more room in the async output buffer.

NOW we come to the most interesting of the buffering problems, terminal input. The nature of X.25 is that there can be up tO seven complete I-frames in flight at once. From the tight RAM limitations, it was clear that the input data for these I-frames could not be duplicated anyplace else in memory. The solution chosen this time around was to build a table describing the active portion of the terminal input ring buffer. The table is eight entries long, and each entry consists of a data start pointer into the terminal input ring buffer, and a data length from that start point. The table is eight entries long because that is the modulus for the X.25

sequence numbers, and these X.25 sequence numbers serve as indexes into the table. Part of the routine activity is to check to see if a new I-frame can be generated, and if so then a check is made for data to fill the I-frame. This checking is performed by calling a routine which returns a removal pointer and a size for data (if any) present within the terminal input ring. If there are data present, then the size will be nonzero, and another table entry can be constructed to describe an P-frame in flight. When I-frames get acknowledged the data space occupied in the terminal input ring buffer is marked no longer in use by advancing to the next sequence number and by changing a pointer, allowing reuse of the memory. One consequence of this design choice on input buffering is a "selective reject" (asking for fills), becomes a more difficult job to implement than if the other feasible approach of using linked lists had been made (It should be noted that selective reject is not part of X.25).

The next area that was simplified by the design decisions is the generation of HDLC frames for information transfer. There is a definite priority in X.25 for the kinds of frames that have to be sent, The priority scheme is implemented by the order in which flags are checked. These flags are generally set in the HDLC input routine, but may be set by anyone to schedule HDLC output. The flags that are checked and the order in which they are checked are: The send RNR flag (set when terminal buffer space gets low) to send a RNR frame, the send REJ flag (set when an out of order frame is received) to send a REJ frame, the received RNR flag must be reset (set when an RNR frame is received) to send an I-frame, the send RR flag (set when I-frame received) to send an RR frame. This section of code is also where the half-duplex decision must be made. The code to generate these output frames is only executed if the modem signal DCD (data carrier detect) is low or if the DCD signal dropped after any of the flags scheduling RNR, REJ or RR frames were set. This simple test is all it takes to prevent the sending of an RR frame for every I-frame received. Notice also that with the order in which the flags are checked, sending an I-frame will be attempted before sending an RR frame, so that if both sides have I-frames to send, then there are no RR messages sent when an I-frame would also acknowledge receipt to the other side.

Another detail that was made quite easy by the "no interrupts in Pascal" decision was the handling of multiple software timers. Actually, there are only two timers, the beacon timer and X.25 timer t1, but they are handled exactly the same way. The generalized timer code is implemented via a Pascal structure; this structure contains an expiration time and a Boolean flag that indicates whether or not the timer is running. "Time" is used loosely, for the only way the Pascal code is aware of the passage of time is by looking at yet another global variable. The time global variable is incremented at a one per second rate by an assembly language interrupt routine. Whenever a timer needs to be started, its expiration time is set to the time global variable plus the number of seconds in the timer interval, and a Boolean flag is set within the timer structure to indicate the clock is active, The big loop of code that is continually checking flags now has to check the timers for expiration, and this is quite easily done by comparing the global variable time with the expiration time in the timer.

## Debugging and checkout

A subset of IS0 standard Pascal was selected which would *work* both with the HIP-64000 compiler and the two systems that I had available for writing and checkout, The real reason for this subset decision was to allow as much checkout of the software as possible in a friendly environment. The "no interrupt code in Pascal" decision made possible the replacement of low level routines in assembly language with routines written in Pascal which could invoke standard Pascal I/O. On the mainframe, a dummy set: of HDLC input and output routines was written to loop back the HDLC output internally (from the Pascal programs point of view) to the HDLC input. On the 8 bit micro system, existing routines for MDLC input and output were modified to use new calling sequences and the Pascal code was actually executed on the air. In both debugging testbeds, the clock was simulated by incrementing the global variable holding the second tick whenever the code in the main loop noticed the system time of day change from a previous value. These sets of routines allowed checkout of the major logic flow of the Pascal software under a symbolic debugger, This self-test arrangement was extremely valuable, and allowed about ninety percent of the bugs in the Pascal code to be eliminated under the friendly environment of the symbolic debugger. Not everything could be checked, and the things that could not be checked were not: setting flags for the low level routines (which didn't exist) or missing transitions in the state/event table that were never exercised during this self-test. One clear fallout of the debugging procedure was transportability, because the software was running on two different Pascal systems before the TAPR TNC software even met the TAPR TNC hardware.

## Summary

The broad design decisions that were made before implementation of the TAPR TNC software served as an aid in inplementation, supplying a framework that would support the detailed coding process. These decisions were made in the light of previous experience (and mistakes) in the implementation of three other systems similar to X.25. There were other choices that could have been made to supply the implementation framework, but my intent was to illustrate the decisions which made the TAPR TNC software almost write itself.

References

1. BX.25 Technical Reference@ Issue 2, June 1980. American Telephone and Telegraph Company.

2. Protocol Specification for Level 2(link level) Version 1.1, Paul Rinaldo, W4RI, et al, October 10, 1982.