

TNC-X: An Expandable Microcontroller-Based Terminal Node Controller

By John A. Hansen, W2FS
Department of Mathematics and Computer Science
State University of New York at Fredonia

1. Introduction

TNC-X is a new Terminal Node Controller design based on the Chepponis/Karn KISS protocol.¹ It is implemented using a Microchip PIC 16F628 microcontroller, a CML MX614 Bell 202 modem chip, an 8K Ramtron FRAM, a Maxim MAX232A level converter chip, and a dual op-amp which provides active audio filtering for the modem. From the beginning, this TNC was designed to be small, have low power consumption, and, most importantly, be expandable. Expandability stems from two sources. First there is an on board socket for a DLP-USB232M USB module which provides the TNC with a USB port. Drivers that are shipped with the module make it appear to the host PC as a standard serial port. Thus PC software that expects to see a serial port on the computer will interpret the TNC-X as being connected to such a port, even when the PC has no serial ports, or they are all used by other applications. In addition, when the USB option is used, the TNC can be powered from the USB port of the computer; no other power supply is needed.

TNC-X is also expandable via an expansion header that allows the addition of “daughter boards”. Power is provided to the daughter board through the expansion header. In addition, signals that would otherwise go to or from a host PC can be intercepted by the daughter board at the TTL level and processed. The I/O on the expansion header speaks “KISS” so that any daughter board only has to send and received data packaged in KISS format (see section 4, below) to access the core module. This makes the development of daughter boards fairly simple and inexpensive. For example, one could use a PIC (or other microcontroller) to create a daughter board that would convert TNC-X into a stand alone digipeater, simply by intercepting the data stream that would normally be sent to the PC, reformatting it (perhaps with call sign substitution) and send it back out the expansion header. The developer of such a board wouldn’t have to worry about the usual nasty details of CRC calculations, bit stuffing, buffering data until the channel was clear, and timing the bits to be sent to the radio, because that would all be handled by the “core” TNC-X module. The parts count of such a daughter board would be extremely low. The most expensive item would likely be the printed circuit board itself. In this configuration, a portable digipeater could be constructed for emergency operation that is extremely cheap and has very low power consumption.

In addition, the expansion header provides access to a second on board serial port. This could be used, for example, to both send and receive data from a GPS receiver. One could develop a daughter board that received data from the GPS and formatted it as KISS frames to be beamed by TNC-X. Again, it would not be necessary to worry about bit stuffing or CRC calculations, and the underlying TNC-X unit would also ensure that the tracker did not transmit while the channel was in use. Additionally, such a daughter board would also have access to the received data stream so this data could be formatted in such a way that it could be displayed on the GPS (assuming that a GPS was used that was capable of displaying incoming position reports).

Even without daughter boards (or the USB module) the device works as a fully functional stand alone KISS mode TNC. The idea behind the expansion options was to provide a platform that would allow additional functionality to be added to the unit with the investment of very little design time and very few parts.

¹ Chepponis, Mike and Karn, Phil, “The KISS TNC: A Simple Host-to-TNC Communications Protocol.
<ftp://ftp.tapr.org/general/kiss.txt>

2. Hardware Description

A tentative schematic drawing of TNC-X appears as Figure 1. I expect that only minor changes will be made before the design is finalized.

Audio entering the TNC from the radio first flows through a 1200 Hz 4th order Butterworth highpass filter. This section of the TNC was added after I realized what poor performance was obtained from the MX614 chip when used on standard amateur packet radio channels. The performance problems associated with the MX614 are probably also endemic to a number of other products that use it as an amateur radio modem. I discovered this when I was testing an early prototype of the TNC-X by routing audio out of my Kenwood D700 receiver to the TNC-X and found that some of the packets that were decoded by the D700 would not be decoded by the MX614-based modem. After watching the data for some time, it became clear that the result was obtained repeatedly... some stations simply couldn't be decoded by the TNC-X that could be decoded by the D700.

There were two possibilities for this behavior. Either these packets were constructed in such a way that my firmware didn't recognize them as packets (see Section 3, below on firmware design) or the data was garbled in the modem chip itself. I dug out an old TNC that I owned that was based on the venerable Texas Instruments TCM3105 chip and found that it copied the data just fine. Using a dual trace storage oscilloscope I was able to compare the data coming out the TI chip with that coming out of the MX614. The data matched exactly on those packets that both TNCs could decode, but it did not match at all on the packets that the MX614 based TNC did not decode. I concluded from this that there was something about the received audio that was causing the MX614 to produce bad data. That is, the problem lay in the modem chip, not in my receive algorithm.

The next step was to analyze the audio from the packet itself. I captured an audio sample that contained both a packet that both modems could decode and one that was decoded by the TI chip, but not by the MX614. I then used a PC-based audio spectrum analyzer to look at the frequency components of the two packets. In addition to the expected peaks at 1200 and 2200 Hz, I found that both packets also had a lot of energy in the range of 200-400 Hz. In addition, the "bad" packet also had quite a bit of energy in the 600-900 Hz range, but the "good" packet did not. I began to suspect that the MX614 was susceptible to noise in this frequency range, while the TI chip was not. I ran the recorded "bad" packet through a fairly primitive PC-based graphic equalizer to attempt to filter the energy below about 1000 Hz. When I did that, the MX614 copied the "bad" packet just fine. My conclusion was that in order for the MX614 to work really well, it was going to be necessary to do some audio filtering ahead of the modem chip. Thus I added U4, a Microchip MCP602 dual op amp configured as a high pass filter.

The modem chip (U1) is set up as per the manufacturer's instructions using a standard colorburst crystal. While I've had good luck with ceramic resonators in the place of crystals, crystals have come down in price a fair amount in recent years and I opted to use them both for the modem chip and for the microcontroller. The microcontroller does all the heavy lifting in this project. I used the PIC16F628 because it is relatively small and now costs less than \$2 each. The project required 14 I/O pins but in its standard configuration, the 16F628 only offers 13. The chip does have a provision, however, for using the Master Clear Pin as an additional I/O pin (but only as an input). This gave me the additional pin that I needed without having to go to larger footprint device.

For the most part, microcontrollers do not contain enough data memory to implement a TNC. Memory is needed both on receive and on transmit. On receive it is necessary to hold an entire packet in memory at once while calculating the CRC and comparing it to the value contained in the frame itself. If the CRC matches, the data can be routed out of the TNC. If the CRC doesn't match, the data must be discarded. Thus, before any data can be passed on, it must be determined that the CRC matches. For a 255 byte long packet, it is therefore necessary to have at least 255 bytes of memory (actually somewhat more than this) devoted to storing the frame. But some applications employ frames that are much longer than this. I know of at least one application that uses a 1.5K byte frame, so I decided I needed 2K of data storage for receive.

On the transmit side, even more storage may be required since data must be buffered for transmission until the radio channel is clear. In this project I allowed 6K for a transmit buffer.

In a previous design, I'd allowed 28K for a transmit buffer, but this now seems like overkill. In that earlier project I'd used a 32K SRAM for storage. This time I wanted to use a chip that could be addressed serially so that the number of I/O lines (and hence the board size) could be reduced. I settled on an 8K Ramtron FM25640 FRAM chip. FRAM technology provides two advantages over an EEPROM, only one of which is important to this project. The memory is non-volatile (I don't really care about this) and it allows a very high number of read/write cycles. Some of the newer FRAM chips allow unlimited read/write cycles. While read/writes are limited with the FM25640 chip, some quick and dirty calculations revealed that if the TNC received one frame a second, it could run for something like 300 years continuously before wearing out the FRAM.

A MAX232A chip provides the TTL to RS232 (and vice versa) level conversions for the project. It is important to use the "A" version of this chip, because it permits much smaller capacitor values than the standard MAX232 chip. The MAX232 provides two incoming ports and two outgoing ports. Thus the additional bi-directional serial port can be made available to the daughter board via the expansion header at virtually no cost.

The optional USB port is a DLP-USB232M based on a USB chip from FTDI. It is supplied in a 24 pin DIP package and makes adding USB to virtually any project a snap. It comes with two kinds of drivers. One set causes the PC to treat it as if it were a standard USB device. The other, called a "virtual com port" driver, causes it to appear as if it were a standard serial port. Drivers are available for free for various flavors of Windows and both MAC OS 8, OS 9 and OS X. Support is built in to the kernel in Linux 2.4.0 and later so no drivers are needed. Windows CE drivers are also available, but they are not free. The only drivers that I have tested thus far are those for Windows.

Unfortunately, the DLP-USB232M is not cheap (about \$25). It would be possible to add USB support using the underlying FTDI chipset itself for less than \$10, but that would involve using surface mount parts. I specifically avoided that because I wanted this to be a project that the average ham could build and most hams these days still shy away from projects that involve surface mount soldering. In this case, the FTDI chip is in a 36-pin surface mount package, which is well beyond the skills of most prospective builders.

There are four configurable jumpers on the board: two control the terminal speed and two are used to engage and disengage the USB port. In addition there are two potentiometers, one to control the transmit audio level and one to set the TXDelay. The KISS standard has a provision for setting the TXDelay in software using the KISS command mode, but my experience is that most programs that support KISS don't support the KISS commands. As a result, I wanted to provide a way to do this in hardware so I wouldn't have to include a "terminal" mode for setting the device parameters. The idea was that you should be able to power it on and just have it run without having to do any software

configuration. The ideal way to implement the TXDelay would have been to use a potentiometer and an analog to digital converter. The 16F628, however, does not include an A to D converter and I did not want to add another chip to provide that function. However the 628 does include a voltage comparator and an internal voltage reference that has 16 different values. TXDelay is implemented by applying a voltage (determined by the potentiometer) to the comparator pin on the PIC. Then the comparator's voltage reference is swept through the 16 steps. When the comparator fires (because the voltage reference has risen above the voltage on the pin) the TXDelay is determined. TXDelay can range from virtually zero to about 500 ms. (this would correspond to a TNC-2 value of 50). Half way is probably a reasonable starting point. If you wish to set it more precisely, you can adjust R13 while measuring the voltage on pin 18 of U3 so as to satisfy the following formula:

$$V = .00625 \text{ TXD}$$

where TXD is the desired delay in milliseconds.

The expansion header is J1. Pin 1 carries the data going out (nominally to a PC), while pin 3 carries the data coming back. If no daughter board is being used, pins 1 and 2 should be jumpered to run the data out to the PC and pins 3 and 4 should be jumpered to route the data coming back. The jumpers should be removed if a daughter board is being used. Pins 5 and 6 provide access to the daughter board for the other MAX232A serial port. Pins 7 and 8 provide the ground and power connections, respectively.

3. Software Description

The firmware for this project was developed using the CCS PCM compiler for the PIC 16 series.² This compiler is not free, but there is no other C compiler available for the 16F628 that is available more inexpensively. I will make available both the source code for the project and the object (.HEX) file so individuals will be able to construct one of these units for themselves without purchasing the compiler itself. Be advised though, that CCS C code will not work with other C compilers. The source code "snippets" presented below are generally not complete functions. For clarity, only the code is included that is necessary to illustrate the particular program behavior under discussion.

3.1 Receive Functions

The receive routines in TNC-X are substantially different than I described in a previous DCC paper on receiving AX.25 using a PIC, in that the current routines are almost entirely interrupt-driven.³ I adopted this approach because it greatly simplified the access of external memory. My previous TNC design used a SRAM for storage. This required a lot of pins (for both addressing and data) but reading and writing occurred almost instantaneously. Reading and writing a byte of data to the FRAM module used in this project takes substantially longer. Writing a byte, for example, requires clocking 32 bits into the FRAM, one at a time. As a result, I wanted to write the data receive routines so that they could interrupt the memory reads and writes. Using interrupts allows all of these activities to peacefully coexist. I don't contend that this approach is original with me; in fact I suspect that most of the people writing AX.25 receive code for PICs are using a similar approach. However, I

² <http://www.ccsinfo.com>

³ Hansen, John, PIC-et Radio II: "How to Receive AX.25 UI Frames Using Inexpensive PIC Microcontrollers" in *19th ARRL and TAPR Digital Communications Conference Proceedings* (Newington, CT: American Radio Relay League, 2000) pp. 67-74.

don't think this approach has been clearly documented in print and I thought it would be useful if the code to do this was carefully explained in a widely disseminated form.

In the discussion that follows, the term "serial port" is used to refer to the local port that data is sent out or received from (such as that connected to a PC). This port might be connected to a daughter board or some other device instead of a PC, however. The term "radio port" is used to refer to data that is encoded as AX.25 and sent to or received from the MX614 modem chip. When the TNC-X is first powered on, it comes up in receive mode (because it has nothing to transmit!). During this period it must balance three tasks simultaneously:

1. It must monitor the radio port and look for the beginning of a frame.
2. It must monitor the serial port for incoming characters that might need to be transmitted (once it has received a complete outgoing frame).
3. It must check to see if there are any characters that are in valid received frames to be sent out the serial port.

Clearly, upon power up tasks 2 and 3 will have nothing to do, but before long as data is received from the radio or from the serial port the PIC must be ready to process it. Fortunately the 16F628 has a built in hardware serial port (USART), so sending and receiving data to and from the serial port takes very little time (routines to handle this are prewritten in CCS C, making it even easier for the C programmer). The program that is receiving data from the radio port has relatively long periods where nothing is happening, so it can handle tasks 2 and 3 in its spare time. So it is a good idea to begin by focusing on task 1.

The MX614 chip idles by sending data that varies between ground and five volts on a regular basis. As a result, one can't simply wait for the data line to be asserted on the modem chip to know when data is being received. There is an energy detect pin on the MX614, but it responds equally well to either received noise or data, so to use it you would have to be willing to run the radio with the squelch closed. It is much better to have a software data detect circuit since the transmit/receive turnaround time is much faster if the squelch doesn't have to be tripped. As a result, I decided not to rely on the modem's energy detect pin.

Transmitted packet data is keyed to a change in tones, not the tones themselves. This technique, called "Non-Return to Zero Inverted", is explained in any good data communications text and in one of my previous DCC papers.⁴ Thus, the basic routine that monitors the data from the modem chip and looks for a valid packet uses the internal timer of the PIC to measure the length of the pulses coming out of the modem. When packet stations begin a transmission they either send flags (01111110) or zero's (00000000). It makes no difference whether the packet station is sending a low or a high tone since a zero is represented by a change in state (from low to high or high to low). A one, on the other hand, is represented by no change in state. Thus a flag could look like the bit pattern in Figure 2, with a period of approximately 5.8 milliseconds (ms) in the middle when it doesn't change state, followed by a pulse of 833 microseconds (us) corresponding to the adjacent 0's (one from the first flag and one from the second). If the packet station idles with a string of zero characters on the other hand, the pattern will be a series of pulses that are all 833 us long. Thus, valid packet data will always begin with pulses that are either 5.8 ms or 833 us long. Now it's possible that the modem chip could randomly send one of these pulses even when it's not receiving data, so the routine that looks for a

⁴ Hansen, John, PIC-et Radio: "How to Transmit AX.25 UI Frames Using Inexpensive PIC Microcontrollers" in *17th ARRL and TAPR Digital Communications Conference Proceedings* (Newington, CT: American Radio Relay League, 1998) pp. 29-37.

valid packet waits until it has received seven pulses in a row that meet these criteria before it assumes that someone is transmitting. If it receives several valid pulses but then receives one that is not valid, it restarts the count. Below is the “findflag()” function that looks for a flag that starts a received packet.

```

void findflag(){
    int time, countlegal = 0;

    oldstate = input(radrx);
    while (input(radrx) == oldstate);
    set_rtcc(0);
    oldstate = !oldstate;
    do{
        while (input(radrx) == oldstate);
        time = get_rtcc();
        set_rtcc(0);
        oldstate = !oldstate;
        if ((time > 29) && (time < 35)) countlegal++;
        else if ((time > 221) && (time < 234)) countlegal+=3;
        else countlegal = 0;
    }while(countlegal < 7);
    do{
        while (input(radrx) == oldstate);
        time = get_rtcc();
        set_rtcc(0);
        oldstate = !oldstate;
    }while (time < 210);
}

```

//oldstate - a global variable that retains the pin state.

// oldstate remembers the state of the input pin
// wait until pin state changes
// since pin just changed set timer to 0
// since the pin state changed, flip oldstate
// keep listening until you get 7 legal intervals
// wait until pin state changes again
// pin state just changed time = how long it took.
// reset timer
// since the pin state changed, flip oldstate
// looking for a zero (a legal interval)
// looking for 6 ones (a flag... legal interval)
// it wasn't a either of the above, start over.
// keep doing this until we get 7 legal intervals
// this loop looks for the flag that will start a frame
// wait for a input pin state change
// pin state just changed time = how long it took
// reset timer to measure next interval
// since the pin state changed, flip oldstate
// we are now at the end of the first flag

The idea here is to repeatedly measure pulses coming from the modem chip until we get seven of them in a row that look like approx 833 us or 5.8 ms. We start by looking for a state change and then keep measuring the pulses until we get seven “legal” ones. The PIC is running at 5 MHz (PIC’s run at the oscillator frequency divided by 4). A prescaler is used so that the timer only advances once ever 128 clock pulses. Thus a time value ranging from 30 to 34 as specified in the program would mean an interval length of $(30 * 128)/5,000,000 = 768$ us to $(34 * 128)/5,000,000 = 870$ us. The range of 222 to 233, on the other hand, is designed to capture the 5.8 ms pulse. When seven of these pulses are detected we can safely assume we are receiving data, not noise. Seven is not a particularly significant number here, but the odds of receiving seven pulses this length from the modem chip when no data is present are fairly low. In the event that does occur, however, the receive decoding loop is designed to promptly recognize that data is not there and throw the receiver back into a state of listening for data.

Once we know we have data, then it is mainly a matter of checking the input pin every 833 us to determine whether the state has changed or not. A state change would indicate a zero, while no state change would indicate a 1. I did this in an interrupt, so that the program could do other things while it was going on. An interrupt provides some code (in this case a specific function) that will execute every time a particular event occurs (in this case when the timer rolls over). You can pretty much ignore the interrupt the rest of the time and plan on it taking care of itself when it is needed. In the case of this program, the interrupt code is used to read in bits until a byte is accumulated and then to process each byte as it is completed. This interrupt will generally fire every 833 us. Now that may seem like it is something that is happening so frequently that no other work can be done, but in reality 833 us is a rather long time from the point of view of a microcontroller that is executing 5 instructions every microsecond! I used the timer again, and set its initial value to 207. Here’s why. A flag is the bitstring 0111 1110. This pattern is shown graphically in Figure 2.

Notice that the bit is read in the middle of a bit period. The reason for this is that if the timer is a somewhat off, or if the transmitter timer is somewhat off, we’ll still get the “right” bit reading if

we read the bit just half way between the times that possible state changes could have occurred. So right after the signal falls, we began looking for the 5.8 ms that told us this was a flag. This diagram clearly shows why we waited 7 bit periods rather than 6 bit periods to look for the trailing 0 in the flag (count them!). When the signal rises again, that's the last zero in the flag. Then the falling signal at the end of the bitstream is the first bit of the byte that immediately follows this flag. However, we don't want to read it just where it fell; instead we want to look at the middle of the bit period. Thus we allow 1.5 bit periods to pass (or 1250 us) before we want to read the next bit. So after the first flag is detected, the timer is set to 207. It will fire an interrupt (and trigger the next bit read) when it gets to 256 (which will actually roll over to 0, since the timer only goes from 0 to 255). So the timer will count $256 - 207 = 49$ ticks before the interrupt fires. Each tick corresponds to 128 system clock ticks (because of the way we set up the prescaler) and the system clock runs at 5 MHz. As a result, if we set the timer at 207, the first interrupt will fire $(49 * 128) / 5,000,000$ seconds later ... that's 1254 us, only 4 us off from the desired value of 1250 us.

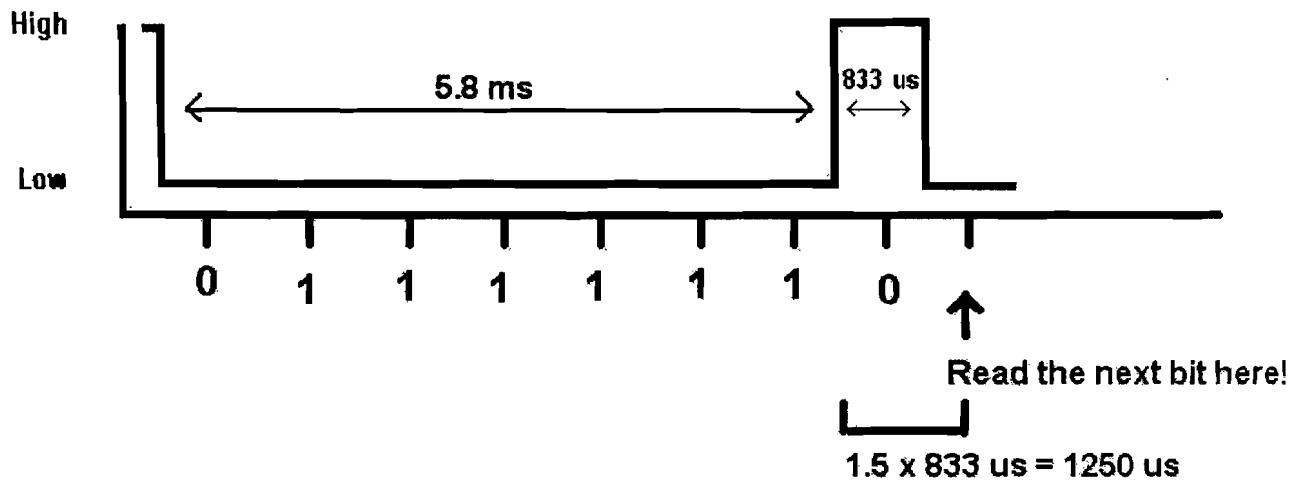


Figure 2: Bit Pattern for a Flag

At this point we are in the middle of the bit period, halfway between the times that a state change could occur. We want to keep reading bits every bit period at this halfway point. Thus, after this first reading, we will want the timer interrupt to fire every 833 us. If we set the timer to 223, it will roll over from 255 to 0 about 845 us later. If we set it to 224, it will roll over every 820 us. Either of these values would be close enough to 833 for our purposes.

Here is a section of the timer interrupt code that shows how the incoming bits are processed:

```

// rbit is used to keep track of the current incoming bit, rcvbyte is the
// current received byte that we are now building, bitcount is the
// number of bits in this byte that we have so far received.
// we set rbit to 1 and then later change it to 0 if the bit is a zero
//input(radx) reads the data pin. if it hasn't changed this is a zero
rbit = 1;
if (input(radx) != oldstate){
    rbit = 0;
    oldstate = oldstate; //flip oldstate variable to reflect the change
}
shift_right(&rcvbyte,1,rbit); //shift the new bit in on the right or rcvbyte.
bitcount++; //increment number of bits received in this byte

```


This approach would work well for reading a byte or two, but it will fail when it tries to read more than 4 or 5 bytes in a row. The reason is that it is “slipping” away from the middle of the bit period by about 13 us every time a bit is read. So in an 8 bit byte, it slips 104 us. Four of those bytes would be enough to move it from the middle of the bit period to before the point where the state change was supposed to occur. As a result, we would then have a read error. Fortunately, we can recalibrate our timer every time we receive a zero because when we see a state change we know that we are 416 us away from the middle of the bit period. All we need is a device to look for these state changes and have it reset the clock every time it sees one.

We would waste a lot of resources if we kept checking the state of the data pin from the modem to see if it had flipped so we would know precisely when that flip occurred. Fortunately, we don’t have to take this approach. The PIC has the ability to generate an interrupt when one of the pins labeled RB4, RB5, RB6, and RB7 is changed. The input from the modem is connected to RB6 and the TNC was designed so that we would be sure that the other three pins would not change during receive. Thus anything that actually fires this interrupt (called a “change on port B interrupt”) must be coming from RB6. When the interrupt fires, we know that the state of the pin just changed. We can use this to reset the timer to eliminate the slippage. Here is the code that does this:

```

if (input(radrx) != oldstate){
    disable_interrupts(INT_RB);           //double check just to make sure the pin has really changed
    set_rtcc(240);                       //no more change on B interrupts until the next bit period
    bit_clear(intcon,2);                 //13 ticks (416 us) to next bit reading
}                                       //don't let it cause an interrupt immediately

```

We use this function to reset the timer to 240 when the interrupt fires. This means the timer will require 16 ticks before it fires again. That amounts to $(16 * 128) / 5,000,000$ seconds = 410 us ... very close to the middle of the bit period.

Fortunately the AX.25 protocol uses a technique called “bit-stuffing” to ensure that the state of this pin will change on a regular basis. As noted above, six ones in a row (an unchanged state for 5.8 ms) indicates a flag. This flag indicates the end of the packet. To ensure that 6 ones never occur in a row anywhere inside the actual data itself, the protocol specifies that a zero should be added (or “stuffed”) every time five ones occurs in a row. This happens regardless of whether then next bit would have been a one or a zero. The bottom line is that we have to keep track of how many consecutive ones we’ve received. If we ever receive five ones in a row, we should then read the next bit. If that next bit is a one, we know that we have arrived at a flag. If the next bit is a zero, we should throw it out and not add it to the byte that is currently being received. If we were to ever get seven (or more) ones in a row, we would know that it was not data, and it was not a flag either. Instead, this indicates that the modem chip is not receiving any data anymore. This feature is used to detect when the transmitting station has stopped transmitting and the modem is just generating random data again.

Fortunately, the MX614 is designed so that it always sends out fairly long strings of ones when it isn’t receiving data. This is used to allow us to break out of the receiving loop if data ceases without a flag or if the modem just happened to accidentally send seven “valid” pulses in a row that mistakenly triggered the start of a packet. Here is the modified receive routine that takes into account bit stuffing:

```

set_rtcc(224); //reset the clock so we'll get the next interrupt at the right time
bit_clear(intcon,2); //don't let it generate an immediate interrupt
if (ones == 7){ //seven ones -- noise from the MX-614... time to start over
    done = true; //we're done receiving, so leave
    return;
}
if (!fiveones){ //do this unless we had 5 ones on last bit
    rbit = 1;
    if (input(radrx) != oldstate){ //if this is a zero
        rbit = ones = 0; //reset ones since this was a zero
        oldstate = loldstate; //flip oldstate variable to reflect the change
    }
    else ones++; //else it must be a one so increment ones
    shift_right(&rcvbyte,1,rbit); //shift the new bit in on the right
    bitcount++; //increment number of bits received in this byte
}
if (fiveones) { // if this is the bit after 5 consecutive ones
    fiveones = false; //turn it off, since we are processing it here
    if (input(radrx) != oldstate){ //if bit is a zero, drop it (no shift right)
        ones = 0; //reset ones since this was a zero
        oldstate = loldstate; //flip oldstate variable to reflect the change
    }
    else{ //otherwise this is another 1
        ones++; //this is the sixth one ... must be a flag or noise
        shift_right(&rcvbyte,1,1); //if it is a one, process it.
        bitcount++; //increment number of bits in this byte.
        isflag = true;
    }
}
}
if (ones == 5) fiveones = true; // if this is the fifth one, next bit gets special processing

```

Start with the line that starts `if (!fiveones)`. This code is executed if the ones count was less than five the last time the interrupt fired. Aside from keeping track of the updated number of ones, this code is identical to the routine discussed above. Now take a look at the last line. If the ones count has gotten up to five, a variable called “fiveones” is switched on. Yes, it would have been possible to consolidate the “ones” and “fiveones” variable into a single variable, but I believe the code is more readable the way it is. Setting the fiveones variable causes the next received bit to be analyzed for bit stuffing. If fiveones got switched on the last time this interrupt fired, then the code block that starts `if (fiveones)` is executed. It reads the next bit. If that bit is a zero, it does not execute the `shift_right` function because we want to discard it (it was the “stuffed” bit). However, it does reset the ones counter. On the other hand, if the next bit is a one, then we either have a flag or noise, so the bit is shifted in. Note in this case the ones counter is incremented. If we receive yet another one in a row, the code that starts `if (ones == 7)` will be executed, indicating that we are receiving noise from the MX614.

The bits are repeatedly added to `rcvbyte` until the bit count rises to 8. Then the byte is complete and it is time to process it. The routine that does this processing is a bit complex because it must take into account many of the features of the KISS protocol. In particular, it must perform a running CRC calculation and then check the resulting two byte CRC against the value that is transmitted along with the frame to determine whether the frame is good or not. However, neither flags, nor the transmitted CRC bytes should be included in this calculation. Furthermore, we must take into account the possibility that we might receive multiple frames, one right after another, separated by only a single flag. In this case, if, at the end of a frame, we went back to the routine that looks for a flag, we’d miss the next frame altogether. So after a frame is complete (that is, after we receive the terminating flag), we must continue to listen for data because it might be another frame coming right along. Fortunately, if there is no second frame, we MX614 will generate a long string of ones and we’ll be kicked out of the receive routine.

Before looking at the code that processes the received bytes, a few words of explanation are in order about the variables that are used in this routine. "Count" contains the number of bytes received so far in this frame. The only reason to keep track of this is so we know what to do when we receive a flag. Flags could either be used to terminate a frame or, it might be that there are a number of flags at the beginning of the frame. If count is pretty low (actually less than 2) it means the flag is at the beginning of the frame. If count is greater than that, it means we are receiving a terminating flag. We don't allow count to increase past 13 (we could have used 3 just as well) because if this is a long packet and we tried to count the number of bytes in it, eventually the count would hit 256 and roll over to 0. This would cause the program to incorrectly think a terminating flag was at the beginning of a frame.

Memory, as noted above, is a FRAM FM25640 chip. It is implemented as two ring buffers: one for transmit and one for receive. The receive buffer is the only one that is used here. It runs from address 0 through address 2047. The pointer that keeps track of where the current value is being written into memory is the variable rcvinadd. So every time a value is written into memory (using the fram_write() function) we have to increment the rcvinadd variable and check to see if it has run past the end of the ring buffer. If it hits address 2048, we know that we've hit the end of the ring buffer and must wrap around to 0. The variable "begin" keeps track of where the current frame began in the FRAM memory. This is necessary because as data arrives for this frame it is written to memory. However, if the current frame has an error in it, we must know where it began so we can go back to that point to start the next frame.

The variables crchi and crclo contain the two bytes of the running CRC calculation. The receivedpackets variable keeps track of the number of frames in memory that have not been sent out the serial port. The routine that writes this data out the serial port decrements this variable every time it successfully sends a frame.

Some complexity is added to this routine because we don't know we have reached the end of a frame until we hit a terminating flag. But when a flag is received, we know the previous two bytes were the transmitted CRC (that we must compare against our calculation). However, we don't know those were the CRC bytes until two bytes later when we hit the flag. This is handled by maintaining a "history list" so that we do not process any byte until two bytes after it is received. Three variables are used for this purpose. As noted above, rcvbyte is the most recently received byte. The variable minus1 holds the byte that was received one byte ago and the variable minus2 holds the byte that was received two bytes ago. By way of illustration, suppose a frame ends with the following bytes:

54 65 73 74 2E 4D 7E .

where 54 65 73 74 are the last bytes in the text to be sent ("test", in this case) and 2E and 4D are the CRC bytes that finish the frame followed by 7E which is the flag character (the hexadecimal representation of 0111 1110). As each of these bytes is received, here is how the rcvbyte, minus1 and minus2 registers will change:

| <u>minus2</u> | <u>minus1</u> | <u>rcvbyte</u> |
|---------------|---------------|----------------|
| | | 54 |
| | 54 | 65 |
| 54 | 65 | 73 |
| 65 | 73 | 74 |
| 73 | 74 | 2E |
| 74 | 2E | 4D |
| 2E | 4D | 7E |

It isn't until we receive the flag in rcvbyte (the last line of the table) that we know that the two bytes we received previously were the CRC bytes. By waiting two bytes before doing the processing, we can treat these bytes as a CRC rather than text after we've received the final flag.

Finally the KISS data frames that are sent to the terminal must all begin with the byte string C0 00 and end with the byte C0. I elected to add these bytes as the data was being received, rather than waiting until it was being sent out the serial port. In addition, special processing is required if a C0 or DB byte appears anywhere in the text. If a C0 appears, it should be replaced with the two bytes DB DC. If a DB appears, it should be replaced with the bytes DB DD. These substitutions are required by the KISS protocol to preserve "transparency". If they were not done, any time a C0 occurred within a frame it would be interpreted on the PC side as the being the end of a frame.

Here, then is the code that processes the received bytes. It is part of the timer interrupt routine, but note that it only runs when the bitcount variable reaches 8 (that is, when a full byte has been received):

```

if (bitcount == 8) {
    if (count < 13) count++;
    if ((rcvbyte == 0x7E) && (count < 3)) count = 0;
    if ((rcvbyte == 0x7E) && (count > 11)){
        crchi = crchi ^ 0xFF;
        crclo = crclo ^ 0xFF;
        if ((crclo != minus2) || (crchi != minus1)){
            rcvinadd = begin;
            rcvinadd++;
            if (rcvinadd == 2048) rcvinadd = 0;
            rcvinadd++;
            if (rcvinadd == 2048) rcvinadd = 0;
        }
        else{
            receivedpackets++;
            fram_write(0xC0, rcvinadd++);
            if (rcvinadd == 2048) rcvinadd = 0;
            fram_write(0xC0, rcvinadd++);
            if (rcvinadd == 2048) rcvinadd = 0;
            fram_write(0x00, rcvinadd++);
            if (rcvinadd == 2048) rcvinadd = 0;
            begin = rcvinadd;
        }
        crchi = crclo = 0xFF;
        count = 0;
    }
    if (rcvbyte != 0x7E){
        if (count > 2){
            switch(minus2){
                case 0xC0: ; fram_write (0xDB,rcvinadd++); if (rcvinadd == 2048) rcvinadd = 0;
                           fram_write(0xDC,rcvinadd++); if (rcvinadd == 2048) rcvinadd = 0;break;
                case 0xDB: fram_write(0xDB,rcvinadd++); if (rcvinadd == 2048) rcvinadd = 0;
                           fram_write(0xDD,rcvinadd++); if (rcvinadd == 2048) rcvinadd = 0;break;
                default:  fram_write(minus2,rcvinadd++);if (rcvinadd == 2048) rcvinadd = 0;
            }
            addcrc(minus2);
            minus2 = minus1;
            minus1 = rcvbyte;
        }
        rcvbyte = bitcount = 0;
    }
}

```

3.2 Transferring the Data to the PC

When a packet is received, it must be eventually transferred out the serial port to a PC or other application (for example, to an application running on a daughter board). This is done with a function called `sendbyteserial()`. This routine is called fairly frequently from the main program (for example, when the TNC is looking for an initial flag from a packet). If any data is available to be sent, this routine will do it. If there is no data to be sent, execution of this function takes very, very little time. The code is fairly straight forward:

```
int store;
if (bit_test(pir1,4)){
    if (receivedpackets > 0){
        disable_interrupts(global);           //don't allow interrupt of memory operation
        store = fram_read(rcvoutadd++);       //get a byte from memory to route out serial port
        enable_interrupts(global);           //interrupt ok now.
        if (rcvoutadd == 2048) rcvoutadd = 0; //memory rollover, if necessary
        if (store == 0xC0){                  //if we are on a packet boundary
            if (rmiddle){                    //if middle = true we have the ending C0
                rmiddle = false;             //no longer in the middle
                receivedpackets--;           //decrement the number of received packets.
            }
            else rmiddle = true;              //if we weren't in the middle, we now are
        }
        putc(store);                          //send the data to the serial port
    }
}
```

I used the built-in serial port (USART) of the 16F628 to send the data. This simplifies things a lot because all that needs to be done to send an entire byte is to write it to the proper register within the PIC. This is accomplished in C by using the `putc()` function. The PIC is then responsible for clocking the data out the serial port. This occurs in the background while our code continues on. As a result, if one is not careful, it is possible to write bytes to the serial port too fast. That is, you have to make sure you don't write a second byte before the first one has been completely sent. The `bit_test` of register PIR1 takes care of this problem. If bit 4 of register PIR1 isn't set, it means the USART isn't ready for the next byte. In this case the `sendbyteserial()` function terminates immediately.

If the USART is ready, then the code checks the `receivedpackets` variable to see if there is anything ready to be sent. If so, it reads a byte from the receive buffer of the FRAM. The `rcvinadd` pointer discussed above keeps track of where the next byte of data should be written in the receive buffer. We need a second pointer (`rcvoutadd`) to keep track of where the next byte of data is that should be sent out of the receive buffer. The only other problem is that when a frame is completely sent it is important to decrement the `receivedpackets` variable so that when it gets down to zero the function doesn't keep sending data that doesn't exist. Using the KISS protocol, each frame has a C0 byte at both the beginning and the end. So when a C0 is read, we must determine whether it was the C0 that is at the beginning of the frame or the end. If it is at the end of the frame, the `receivedpacket` variable is decremented.

Finally note that the interrupts are disabled for a very short period of time. I did this because if a timer interrupt were to fire while we were in the middle of reading data out of the FRAM, the PIC might be trying to both read and write data to/from the FRAM at the same time. This would undoubtedly result in an error. I wanted to disable the interrupts for as short a period of time as possible, so that the program could very quickly get back to processing incoming bits.

3.3 Transferring the Data from the PC

The TNC never knows when the PC is going to send some data. It could be going on while the TNC is in the middle of receiving data off the air, while the TNC is looking for a valid data stream or even while the TNC is in the middle of a data transmission. Whenever new bytes present themselves at the serial port, the TNC must be ready to transfer them into the transmit buffer. The transmit buffer is implemented in the same FM25640 FRAM chip as the receive buffer. The transmit buffer starts at address 2048 and runs through address 8191. As the program works it's way up the transmit buffer, when it reaches memory location 8192, it wraps around to memory location 2048. Two pointers are also used for the transmit buffer, one to indicate where the next character should be put in memory and one to indicate where the next character should be read. The code that grabs characters from the incoming serial port (USART) looks like this:

```
int inbyt;

if (bit_test(pir1,5)){ //if there is a byte in the buffer
    inbyt = getc(); //get the incoming byte
    if (inbyt == 0xC0) {
        disable_interrupts(global); //don't allow interrupt of memory operation
        fram_write(inbyt,txinadd++); //write byte to memory
        if (txinadd == 8192) txinadd = 2048; //rollover memory space if necessary
        enable_interrupts(global); //interrupt ok now.
        xmiddle = !xmiddle; //flip middle indicator
        c0xmit++; //c0xmit/2 is the number of completed packets in the buffer.
    }
    else{
        if (xmiddle == true) {
            disable_interrupts(global); //don't allow interrupt of memory operation
            fram_write(inbyt,txinadd++); //write byte to memory
            if (txinadd == 8192) txinadd = 2048; //rollover memory space
            enable_interrupts(global); //interrupt ok now.
        }
    }
}
```

The function first checks register PIR1 to see if there is anything waiting in the USART to fetch. If there is, it grabs it and puts it in a variable called inbyt. The counting of the number of frames is handled a little differently here than in the receive module. The variable C0xmit holds the number of C0's that have come in, so the total number of frames currently available to be sent is C0xmit/2 (Note that if C0xmit is 3, dividing it by two will produce 1, not 1.5 since this is integer math). Note that the program also keeps track of whether the byte is in the middle of a frame or in between frames. Bytes that are not equal to C0 are only transmitted if they are in the middle of a frame. All bytes *should* be in the middle of a frame. That is, because each frame begins and ends with a C0, there should never be a byte that is after the end of one frame but before the beginning of another frame. However, I've included this just in case something gets set improperly on the PC side. For example, if someone used a non-KISS packet program with this TNC, very unpredictable results could occur if this type of filtering was not included.

The data is put in memory exactly as it comes in so that issues relating to KISS substitutions are handled when the data is being transmitted, not when it's being received from the PC. The reason for this is that at the same time that this routine is going on, the TNC may be receiving data. Because this is a half duplex TNC, when it is transmitting it doesn't have to worry about the timing of the interrupts associated with receiving. So there is plenty of time at that point to deal with the conversion of the KISS data stream into a stream of bytes that will be sent over the air.

3.4 Transmitting Data Over the Air

Transmitting data over the air is handled by three functions. The function `xmit()` handles issues relating to keying the transmitter, reading the transmit buffer, and processing the KISS substitutions. A second function, `sendbyte()` handles the details associated with flipping the output pin (that goes to the modem) to generate the correct bit pattern for a byte. Bit stuffing also occurs in this function. Finally, a third function, `crcbit()` is responsible for the ongoing calculation of the CRC value that must be transmitted at the end of the frame. Starting with an overview, here is the code for the `xmit()` function:

```
int c,outbyte,txdelay;
short DB, firstpack = true;

while (c0xmit/2 > 0x00){
    stuff = 0;
    crcflag = FALSE;
    xcrclo=xcrcchi=0xFF;
    output_high(m0);
    outbyte = fram_read(txoutadd++);
    if (txoutadd == 8192) txoutadd = 2048;
    outbyte = fram_read(txoutadd++);
    if (txoutadd == 8192) txoutadd = 2048;
    if (outbyte == 0) output_high(PTT);
    if (firstpack){
        txdelay = gettxdelay();
        flag = TRUE;
        set_rtcc(0);
        for (c=0;c<txdelay;c++) (sendbyte(0)); //send nulls while TXDelay
        sendbyte(0x7E);
        sendbyte(0x7E);
        flag = FALSE;
        firstpack = false;
    }
    outbyte = fram_read(txoutadd++);
    if (txoutadd == 8192) txoutadd = 2048;
    While (outbyte != 0xC0){
        switch (outbyte){
            case 0xDB : db = TRUE; break;
            case 0xDC : if (db == TRUE){
                sendbyte(0xC0); db = FALSE;
            } else sendbyte(0xDC);
                break;
            case 0xDD : if (db == TRUE){
                sendbyte(0xDB);db = FALSE;
            }else sendbyte(0xDD);
                break;
            default : sendbyte(outbyte);
        }
        outbyte = fram_read(txoutadd++);
        if (txoutadd == 8192) txoutadd = 2048;
    }
    crcflag = TRUE;
    xcrclo = xcrclo^0xff;
    xcrcchi = xcrcchi^0xff;
    sendbyte(xcrclo);
    sendbyte(xcrcchi);
    FLAG = TRUE;
    crcflag = false;
    sendbyte(0x7e);sendbyte(0x7e);
    flag = false;
    c0xmit = c0xmit - 2;
}
output_low(PTT);
output_low(M0);

//do this if there is any data to transmit
//keeps track of number of ones
//we aren't transmitting the CRC now
//reset CRC values
//set up the MX614 to transmit
//reads the C0 byte
//rollover if necessary
//reads the next byte (00 if it's data);
//rollover if necessary
//key the transmitter, but only if data, not KISS command
//got to do the txdelay if this is the first frame
//read the txdelay pot
//start with some flags (no bit stuffing)
//set the timer to zero to begin timing
//send nulls while TXDelay
//send flag
//each byte takes approx 6.7 ms
//done with flags
//subsequent frames in this transmission don't need txdelay

//reads the first address byte.
//rollover if necessary
//does the kiss substitutions and transmits byte

//now sending CRC... don't do crc calc on these bytes
//finish CRC calculation

//send the low byte of crc
//send the high byte of crc
//going to send flags now (no bit stuffing)
//done sending CRC
// Send flags to end packet
//done with flags
// decrement the number of frames waiting.

//unkey PTT
//MX 614 to recv mode
```

Multiple frames can be sent in a single transmission. Every transmission will transmit zeros for the length of time specified by the TXDelay. Then two flags are transmitted to indicate the beginning of a frame. If a second frame is transmitted right after the first frame, we do not want to repeat the process of sending a TXDelay period. So a variable called “firsttime” is used to indicate the first frame in a transmission. After initializing some variables, the xmit() function puts the MX614 into transmit mode by toggling the M0 control line high. The data in the transmit buffer is stored in KISS format, so each frame will begin with a C0 00. If the frame starts with C0 01, that means that it is a control frame, used to set things like TXDelay, Slottime, and Persist. My experience is that KISS mode programs very rarely use these commands and at this point I have not supported them in TNC-X. However, if a command frame is received, it is important to make sure it isn’t transmitted over the air, so this function does check to make sure the value of the second byte is 0. As a result, the PTT line is only keyed if this value is a 0. Then, if this is the first frame in the transmission the gettxdelay() function is called to read the TXDelay pot. Any time flags are being sent, it is important to set the flag variable because it keeps the sendbyte function from doing bit stuffing. Furthermore, any time either flags or CRC bytes are being sent, it is important to exclude them from the ongoing CRC calculation. The variable called “flag” indicates that a flag is being sent while the variable called “crcflag” indicates that a CRC byte is being sent.

The next section of code repeatedly reads the next variable out of memory, and does any modifications required by the KISS substitutions (changing DB DC to C0 and DB DD to DB). This goes on until it reads a concluding C0 coming from the transmit memory. The variable txoutadd is the pointer that keeps track of the current location in the transmit memory. At the end of this procedure, the two byte CRC is transmitted followed by a couple of more flags. If there are more frames to be sent, the program moves on to the next one (though it skips the TXDelay transmission). If there are no more frames, it flips the MX614 back to receive and unkeys the transmitter.

Individual bytes are sent using the sendbyte() function. It looks like this:

```
int k, bt;
for (k=0;k<8;k++){
    bt = inbyte & 0x01;
    while (get_rtcc() < 32) rcvbyteserial();
    delay_us(11);
    set_rtcc(0);
    if ((!crcflag) && (!flag)) (crcbit(bt));
    if (!bt){
        stuff = 0;
        portb = portb^0b00001000;
    }
    else{
        stuff++;
        if ((!flag) && (stuff == 5)){
            while(get_rtcc() < 32) rcvbyteserial();
            delay_us(11);
            set_rtcc(0);
            stuff = 0;
            portb = portb^0b00001000;
        }
    }
    rotate_right(&inbyte,1);
}
```

The byte to be sent is stored in a variable called inbyte. The bits of this byte are stripped off one by one starting on the right and stored in a variable called bt to be transmitted. We need a 833 us delay between transmitted bytes. Allowing the timer to tick 32 times provides a delay of $(32*128/5,000,000) = 819$ us. An additional delay of 11 us is added in. Together with the time needed to execute the instructions, this creates a total delay of 833 us. This could have been accomplished

simply by using the `delay_us(830)` instruction, but I wanted to repeatedly execute the `rcvbyteserial()` function at the same time in order to receive and process additional incoming bytes from the serial port.

When the delay is completed, the timer is reset to 0 so that the delay for the next bit will be correct. Then, if the current byte is neither a CRC byte nor a flag, it is added to the ongoing CRC calculation by calling the `crclbit()` function. Finally the bit is transmitted. We must keep track of the number of consecutive one bits that have been transmitted so that if five ones are transmitted in a row, a zero can be “stuffed”. The current number of consecutive ones is stored in a variable called “stuff”. If the current bit is a zero, the stuff variable is reset to zero and the output pin (RB3) state is flipped. Otherwise, if the bit is a one, it is not necessary to change the state of the port pin, but it is necessary to increment the stuff variable. When stuff gets to five, it is necessary to add an extra zero in the bit stream. This is done by repeating the 833 us wait and flipping the RB3 port pin. At that point the stuff variable is reset to zero. This process is repeated for all 8 bits in the byte.

The last function related data transmission is the `crclbit()` function. The function is very short so it will be reproduced here:

```
#asm
    BCF  03,0
    RRF  xcrchi,F           //rotates the entire 16 bits to the right
    RRF  xcrclo,F
#endasm
    if (((status & 0x01)^(tbyte)) ==0x01){
        xcrchi = xcrchi^0x84;
        xcrclo = xcrclo^0x08;
    }
```

The process for calculating a CRC is described in many data communications textbooks and can also be found in an excellent 1986 article in *Byte* magazine.⁵ I fell back to assembly language for this because CCS C has no simple mechanism for rotating 16 bits.⁶ Register 03 is the STATUS register of the PIC.

3.5 Putting It All Together

Relying on interrupts makes it fairly easy to put all of these pieces together into a working modem. Aside from initializing a lot of variables, the main part of the program is a fairly short infinite loop (see below). The `findflag()` function is called to look for a flag that starts a packet. While `findflag()` is running, it also repeatedly does the following:

1. Handles incoming data from the serial port (using `rcvbyteserial()`);
2. Sends any waiting data out the serial port (using `sendbyteserial()`);
3. Switches to transmit mode if there is every a complete frame awaiting transmit (using `xmit()`).

None of these functions takes very long, unless there is data to be transmitted. If there is data to be transmitted, it doesn't matter that we are no longer looking for a flag, because we won't be receiving and transmitting at the same time. Note that even if we wanted to develop a full duplex TNC, the MX614 is inherently a half duplex chip.

⁵ Morse, Greg, “Calculating CRCs by Bits and Bytes” (*Byte*, September 1986, pp. 115-124).

⁶ Actually this is probably incorrect. I wrote this section of code some years ago and haven't revisited it.

```

while (true){
  crcl0 = crchi = 0xFF; //set up next packet to be received
  done = fiveones = false;
  ones= rcvbyte = bitcount= count = 0;
  fram_write(0xC0,rcvinadd++);
  if (rcvinadd == 2000) rcvinadd = 0; //start KISS frame
  fram_write(0x00,rcvinadd++);
  if (rcvinadd == 2000) rcvinadd = 0;

  findflag(); //go find a flag
  output_low(LED); //packet is here, turn on DCD LED
  set_rtcc(207); //delay 1.5 bit intervals - middle of the next bit.
  bit_clear(intcon,2); //don't get an interrupt right away
  enable_interrupts(INT_RTCC | INT_RB | GLOBAL); //enable interrupts to begin to collect data
  while (!done) { //Do this while data is being
    //received.. stop when there is noise
    sendbyteserial(); //send received data out serial port
    rcvbyteserial(); //take in data to transmit from serial port
  }
  output_high(LED); //packet is over, turn off DCD LED
  disable_interrupts(INT_RTCC | INT_RB | GLOBAL); //disable interrupts to prepare for next findflag
}

```

When a flag is found it means we are receiving real data so the first thing that happens is that the DCD light (LED) is turned on. Then the timer is set up to receive the next bit of data and the interrupts are enabled so that data can be received. At this point we enter a loop that handles communication from the serial port (both ways) except when it is processing the interrupts that are generated by the incoming data stream. When the data goes away, the loop is exited, the DCD light goes off, and everything is set up to start looking for a flag again.

There are a number of other functions in the program (reading and writing to memory, for example) but they are all pretty straight forward. The complete code will be posted on my TNC-X website:

<http://www.tnc-x.com>

4.0 Developing Daughter Boards for TNC-X: Explorations in Object-Oriented Hardware

Over the last couple of decades “object-oriented programming” has become all the rage in software development ... and for good reason. Among other advantages it allows developers with a rather limited knowledge of many areas to develop sophisticated programs. The average Visual Basic developer, for example, doesn’t have a clue as to how windows and dialog boxes are actually constructed (at the operating system level), he simply knows how to use them... that is, he understands the user interface. This shouldn’t seem strange or magical, though sometimes it does. My wife, for example, doesn’t have a clue as to how a modern car actually works (well, ok, I don’t either). However, we are both experts at the car’s “user interface” so we can use it to get real work done.

This is the basic concept that I had in mind with TNC-X. You don’t need to understand much of anything that was discussed in section 3 of this paper in order to develop daughter boards for TNC-X. All you need to know is this:

1. Five volts is available to you on pin 8 of the expansion header... Ground is on pin 7.
2. You can communicate with the packet engine in TNC-X using standard serial data at TTL level voltages at either 1200, 4800, 9600, or 19200 baud (depending on the jumper settings). However, no flow control is supported.

3. The data will come in from TNC-X on pin 3 and can be sent to TNC-X on pin 1 of the expansion header.
4. You have available to you two serial ports. The data on these ports will be converted to RS232 levels. You can transmit RS232 data on pins 2 and 5 of the expansion header and receive RS232 data on pins 4 and 6 of the header. The port on pins 2 and 4 may optionally be routed to the USB port instead.
5. All data that comes in from TNC-X will be in KISS mode. This means the following:
 - a. Each data frame will start with C0 00 and end with C0.
 - b. Every time you see the bytes DB DC you should delete them and put in a C0.
 - c. Every time you see the bytes DB DD you should delete them and put in a DB.
6. All the data you send to TNC-X must be put in KISS mode. This means the following:
 - a. Add a C0 00 to the beginning of each frame and a C0 to the end.
 - b. Every time you have a C0 in your data, substitute a DB DC.
 - c. Every time you have a DB in your data, substitute a DB DD.

That's about it. You don't have to worry about CRC calculations, bit stuffing, buffering data or any of the other messy details discussed in section 3. Let the TNC-X object take care of all that for you, just focus on the user interface.

So, suppose you wanted to build a digipeater daughter board. You have a data stream coming in from TNC-X that will be in AX.25 format wrapped in KISS. You just read it from pin 3 and evaluate the packet header that has the addressing in it. If it meets your criteria to be digipeated, you read it out pin 2, adding your call sign in the from field. While you're at it, you might want to do some work on the digipeater path to improve network efficiency. My point here is that this would be a very trivial project, because the underlying packet engine is already created for you.