PIC-et Radio II: How to Receive AX.25 UI Frames Using Inexpensive PIC Microcontrollers

by John A. Hansen, W2FS
State University of New York
49 ***Maple Avenue***
Fredonia, NY 14063
hansen@fredonia. edu

Abstract: This paper provides step by step documentation of how to decode AX.25 Ulframes using inexpensive PIC microcontrollers. It is designed primarily for those who wish to receive packet radio WI beacons from point to multipoint communications. The article assumes some knowledge of programming concepts and PIC microcontrollers.

Keywords: AX 25, UI Frames, PK ***Microcontrollers***

Introduction

At the 1998 ARRIJTAPR Digital Communications Conference I presented a paper describing how to encode AX.25 frames using a PIC microcontroller so those frames could be transmitted via Amateur Radio.* However, having learned to transmit packet using cheap chips, it was only natural that people would want to know how to receive packet frames as well. The first effort made in this direction that I know of was some assembler code developed by Byon Garrabrant, N6BG. A number of packet receive routines (by Byon and others) are currently available on the TAPR PICSIG software FTP site (ftp.tapr.org/picsig/software). The purpose of this paper is to explain the basics of how PIC-based packet receive firmware works.

Note that this paper will not provide you with a working packet receive system. If that is what you are seeking, simply use the code provided on the TAPR FTP site. Instead, my purpose is to describe the theory behind how PIC based packet receive routines work so that you will be able to create your own code to decode AX.25 frames. In keeping with this goal, I've tried to make the code that is presented here as simple as possible. To do this, I made a number of simplifications. First, all code examples here are written in C (my code is designed for the CCS C compiler, because it is relatively cheap).[2] I did this because I think that regardless of whether you are planning to develop your project in C or assembler, C is somewhat easier for most people to follow. Secondly, the code presented here is designed to receive packets that are no longer than 40 bytes (including the header). I used this approach because I wanted to use only a PIC 16F84 microcontroller and an MX-6 14 modem chip. Significantly longer packets would require some external storage, but the 16F84 has adequate on-board storage to handle relatively short packets. A number of packet receive implementations have been realized that use external storage and can handle longer packets. For example, Mike Berg NOQBH has used a Ramtron FRAM chip as storage for longer packets. I have used a 32K static RAM chip in my PIC KISS TNC to provide both transmit and receive storage.[3] However, limiting the receive data to 40 bytes means that

' Hansen, John A., "PI&et Radio: How to Send AX.25 UI Frames Using Inexpensive PIC Microprocessors" in 2 7'^{h} *ARRL and TAPR Digital Communications Conference* (Newington, CT: ARRL, 1998) p. 29. Also available as DCCZIP at ftp.tapr.org/pub/waOptv.

[2] www.ccsinfo.com

[3] Hansen, John A., "An Inexpensive KISS mode TNC" forthcoming in QST.

we need not include here routines for writing and reading to storage. Finally, the code presented here is designed to receive UI frames. The results presented here could be extended to cover the range of AX.25 frames, since the principles involved in receiving the data are the same in any case. To make this code relatively simple, however, this paper will present code that merely receives the frame, assumes it is a UI frame and formats it and pumps it out a serial port to a terminal. The construction of UI frames is not covered here in detail. For a discussion of how to build up UI frames, see my 1998 DCC paper.

Receiving Bits

The essence of receiving packet is receiving data bits. While data is transmitted over the air as a series of tones (1200 Hz and 2200 Hz), a bit of information is not represented by the frequency of the tone, but rather it is represented by whether there is a change in tone or not. A shift in tone (either from 1200 to 2200 Hz or vice versa) represents a zero, while no shift in tone represents a one. This project assumes that modem functions are done by another chip (in this case the MX-614) so that an input pin on the PIC sees a "high" (+5 volts) when there is 1200 Hz tone and a "low" (0 volts) when there is a 2200 Hz tone. We will begin the discussion of receiving packet frames with the code that is used to receive a bit. I use the function bitin() to accomplish this:

```
int bitin() (                          //function to read a bit

static int oldstate;                   //oldstate retained between runs of this function
int k;

     for (k=O;k<l21;k++){              //this loop allows 838 us to go by. If no state change, bit is 1
      if ( input(rcvPin) ! = oldstate) {   //if state has changed
          oldstate = input(rcvPin);    //update oldstate
          delay_ us( 430);             // move to halfway thru the next bit
          return 0;                    i/return 0 if state changed
        f/'/end of if
     i//end of for
     return 1:                         //return 1 if state did not change
}//end of bitin()
```

The PIC pin that is connected to modem chip is referred to as 'VecPin". The main purpose of this routine is to look for state changes on the input pin. The function "input(recPin)" is built into the CCS compiler and returns a 1 when the pin named rcvPin is high and a 0 when the pin is low. Two variables are used here, k (to count the number of times through the loop) and oldstate, which keeps track of the previous state of recPin. Oldstate is a *static* variable so that it won't lose its value between times when this function is called. It cycles through the 'if loop 121 times to see if the state has changed from it's old value. Assuming you are running the PIC at 10 MHz, 121 times through this loop will take 838 microseconds (ps) to complete.[4] In a 1200 baud transmission, each bit lasts 833 ps (1,000,000/1200). Thus if 838 ps go by with no change of the state of recPin, we must be looking at a 1 rather than a zero. So the function returns a 1.

[4] The simulator function of the PIC development environment MPLAB allows you to precisely time events while the code is running. MPLAB is available free of charge from www.microchip.com.

If, on the other hand, a state change is detected during this time period, t.he function returns a zero. If it finds a zero, the function will also introduce a delay of 430 ps. This is an important point. Packet is an asynchronous communications system. While a bit lasts 833 ps, there is no way to know where a particular bit would start and stop other than by looking for those instances when the tone changes. At the moment the tone changes, we know where we are in the bitstream. Thus we recalibrate the receiver on every tone change. Adding 430 ps right after a tone change throws us just past the middle of the next bit. Consequently every time a zero (tone change) is detected the receiver is reliably placed at a spot that is just after the middle of the following bit. If this were not done, small errors in timing would add up as we received more and more bits and eventually we might miss a bit altogether. Packet radio transmissions are designed to ensure that a zero will be transmitted at least once every six bits. Thus the receiving routine is recalibrated at least every five milliseconds.

The function bitin() receives one bit of the packet bitstream. The trick is to make sure that the function is called sufficiently often that no bits are missed. In addition, we must know how to process the accumulated bytes when an entire packet is received.

Let’s Play Capture the Flag

You cannot decode a packet frame unless you receive the entire frame. This is because the frame contains a two byte frame check sequence (FCS) that must match the value transmitted by the sending station in order for the packet to be valid. Hence it does no good to begin receiving in the middle of a frame. When the program first starts, therefore, it must begin by looking for the flags that indicate the beginning of a frame. These flags are simply the byte 01111110 in binary (7E in hexadecimal notation). Most (though not all) TNCs send this series of bytes during the entire transmit delay (TXDelay) period, but all must send at least one flag to indicate the beginning of the frame. The problem is that when you turn your packet receive system on, you don’t know for sure that you are receiving the beginning of a packet. You don’t even know for sure that you are receiving the beginning of a byte. So you must continually receive and check bits until you find the 01111110 sequence. Because of bit stuffing (discussed below) the only time you will receive six ones in a row is when you are receiving a flag.

Here is the code to look for the first flag:

```
int cbyte = 0;                          //initialize
while (cbyte != Ox7e) {                 //find the first flag
        shift_right(&cbyte, 1 ,bitin());   //add a new bit to the left of cbyte, discard right bit
 } //end of while
output_high(LED);  //turn on the DCD light
```

The line that starts with shift right may seem a little strange because it is a function provided by the CCS compiler, not by the C language. What it is says is as follows. Get a new bit from the receiver (using the function bitin(Move each bit in the byte called cbyte one position to the right. Then take the new bit and append it to the left end of cbyte. So, for example, if cbyte was 10001000 before this line of code was run and if the new bit obtained by bitin() was a 1, the resulting cbyte would be 11000100. The bit on the extreme right gets bumped off the end and lost. The purpose of this is to retain a memory of the previous seven bits that were received and to keep getting new bits until the most recent 8 bits match the pattern 011 I 1110 (7E). When this occurs we know that we have found a flag. At that point we turn on the DCD light using the CCS built in function output_high(LED), to indicate that data is being received.

Well, almost. It seems the MX614 chip sends a random series of ones and zeros whenever it is not actually receiving audio tones. As a result, sometimes it just happens to send the 7E byte. When this happens the DCD light will come on even though no data is being received. However, whatever random series of bits that is received afterwards will almost certainly not pass the FCS check, so these “errors” will only show up as a flickering of the DCD light, not as the reception of bad data?

After the first flag has been received, the next byte may be a flag, or it may be the beginning of the actual data. However, from here on out, we can grab the data eight bits at a time because we know that we are at the beginning of a byte. So instead of examining the byte after every bit is added, we can simply repeat the procedure of grabbing a bit and shifting it into cbyte eight times:

```
int 1;
int  bte[40]:
While @byte = z Ox7e)(                       //find the other flags
        for(  i=O;i<&i++)  (                 //repeat this 8 times
                shift_ right(  &byte,  1  ,bitin());   //add a new bit to the left of cbyte, discard right bit
        i                                    //end of for
}                                            //end of while -- now at end of all the flags
bte[O] = cbyte;                              //you’ve now got the first address byte
```

The while loop continues to execute until it finds something other than a flag. When the loop exits, the value in cbyte will be the first value of address portion of the AX.25 frame. I used a forty element array called bte to hold the actual data from the packet. Thus the value in cbyte is assigned to the first (zero”‘) element of the bte array.

Collecting the Data

From here we need to continue to collect data until another flag is encountered telling us that we have reached the end of the packet. However, there is one more complication. As noted above, the AX.25 protocol ensures that there will be a zero (and hence a tone change) at least every six bits. Furthermore, a flag is the only instance in which six ones in a row are allowed to occur in the data stream. If six ones were to occur in the data itself, the protocol requires that a ‘zero’ bit be inserted after the fifth ‘one’ bit. This procedure is referred to as “bit-stuffing”. For example, suppose the data included the two byte sequence 00001111 11110000 (in hexadecimal notation this would be OF FO). In that case, a zero would be added after the fifth one in the sequence, so now seventeen bits would be

’ Note that in a “real” receiving system you might want to add a mechanism for providing a limit on the number of bytes of random data that the unit would interpret as real data before testing (and presumably failing) the FCS check.

transmitted as follows: 00001111   101110000. This extra zero needs to be removed on the receive end in order for the data stream to be the intended sequence of OF FO. So while receiving it is important to monitor the bit stream for any zero that occurs after five consecutive ones. C)f course, if the next bit after five consecutive ones is another one, we know that we have found the flag that ends the packet and it is time to stop receiving data.

Note that it is not necessary to monitor the first byte for bit stuffing. The reason is that each byte of the address field (except the last byte) must end in a zero. Since the addrelss field must be at least fourteen bytes long, you really don't need to worry about bit-stuffing for the first few bytes.

Here is the code that is used to collect the rest of the data:

```
int test, newbit, numbyte, ones;
test =O;
numbyte = 1;                                   //we already collected 1 byte
while (test != l){                             //do this until the flag at the end of the packet
        for(i=O;i43;i++) {                     //collect 8 bits
                newbit = bitin();              //get a bit
                if (newbit = = 1) (ones++);    //increment the ones counter
                        else (ones = 0);       //if bit is a zero, reset the ones counter
                if (ones = = 5) {             //removes bit stuffing
                     test = bitin();          //get the next bit but don't add it to cbyte
                     ones = 0;                //reset the ones counter
                ›
                shift_ right( &byte, 1 newbit);  //append the new bit to cbyte
        ›                                     //end of for
        if (test = = 0) {
                bte[numbyte] = cbyte;         //add cbyte to the array
                numbyte++;                    //increment the number of received bytes
        ›
}//end of while
```

Test is a variable that is a one if a flag is encountered and a zero otherwise (it could be boolean rather than int). This snippet of code collects each bit and tests to see if it is a zero or a one, and appends it to the value of cbyte. The variable 'ones' keeps track of the number of consecutive ones that have been received. If the new bit is a one, it increments the ones counter; if it is a zero it resets the ones counter to zero. If five consecutive ones are ever detected, an additional bit is retrieved. This additional bit is not added to the current byte, but rather it is put in a variable called 'test'. If value of test is a one, it means that we have received a flag and the data transmission is done. If test is a zero, it means that bit stuffing occurred and the value is discarded. As each byte is completed, it is added to the array bte and the number of bytes (stored in numbyte) is incremented.

The while loop continues until a flag is encountered. When the loop exits, all of the data in the packet is stored in the array bte and the number of data bytes that were received is stored in numbyte.

Checking the FCS and Formatting the Output

All that remains is to check to see whether the received data is valid and to format it and route it out the serial port if it is valid. The following line of code performs this function:

```
if (fcschecko) (printout());                    //if the fcs checks output the packet.
```

This code calls two functions, fcschchecko to determine if the data is valid, and, if it is, printout0 to route it to a serial port. The AX.25 protocol calls for the use of a cyclical redundancy check that it refers to as a frame check sequence (FCS). Two bytes are appended to the end of each frame after the data that are the FCS values. At the receiving end, to ensure the integrity of the data, we must calculate the FCS and compare the values with those that are transmitted in the packet itself (and have been calculated at the transmitting end). The fcscheck() function that performs this check is virtually identical to that included in my 1998 paper on transmitting A.X.25?

```
short fcscheck() {                                      //computes the fcs
int i,k,bt,inbyte, fcslo, fcshi;

  fcslo=fcshi=OxFF;                                     Iintialize FCS values
  for (i = O;i<(numbyte-2);i++) {                       //calculate the FCS for all except the last two bytes
      inbyte = bte[i];
      for(k=O;k<8;k++) {                                //perform this procedure for each bit in the byte
            bt = inbyte & 0x01;
            #asm                                        //embedded assembly language route to do a 16 bit rotate
                BCF  03,O
                RRF  fcshi,F
                RRF  fcslo,F
            #endasm
            if (((status & OxOl)/‘(bt)) ==OxOl){
                fcshi = fcshPOx84;
                fcslo = fcslo”OxO8;
            ›                                           //end of if
            rotate_ right(&inbyte, 1);                  //set up to do the next bit
      |                                                 //end of for
  〉
  fcslo = fcs10*0xff;
  fcshi = fcshi”Oxff;
  if ((bte[numbyte-1] == fcshi) && (bte[numbyte-21 == fcslo)) {
      return 1;
  }
     else return 0;                                     //if the computed values equal the last two data bytes
›                                                       //return a 1, otherwise return a 0
```

[6] For a discussion of how this seemingly arcane calculation is performed, see Morse, Greg “Calculating CRCs by Bits and Bytes” (Byte, Sept 1986, pp. 115 124).

All that remains is to format the output and route it out the serial port. Doing this requires an understanding of the construction of AX.25 UI frames, but otherwise is fairly trivial. Here is the printout0 procedure that I used to accomplish this:

```
void printout0 {                                          //function to display the received packet
int i,L,m,temp;

    for  (m=7;m<l3;m++){                                  //print the source callsign
         if (bte[m] != 0x40) (putc(bte[m]>>l));           //note spaces (40) are not printed
    1
    putc('-');
    putc(((bte[ 131 & Ox7F)>>1));                         //print source SSID
    putc('>');
    for (m=O;m<@m++) {                                    //print the destination callsign
         if (bte[m] != 0x40) (putc(bte[m]>>l));
    1                                                     //end of for
    putc('-');
    putc(((bte[6]  &  Ox7F)>>1));                         //print the dest SSID
    L = 7 ;

    if ((bte[l3] & 0x01) != l){                           //print any path that: may exist
         do-t
            pW'9');
            L=L+7;
            for (m=L;m<(L+6);m++) {
                   if (bte[m] != 0x40) (putc(bte[m]>>l));
            ›                                             //end of for
            putc('-');
            putc(((bte[(L+6)] & Ox7F)>>l));
         } while ((bte[L+6] & 0x01) != 1);
    ›                                                     //end of if
    putc(':');
    putc(' ');
    L=L+9;                                                //add 9 to move past the last callsign, cntl and PID
    while (L< numbyte-2) {                                //print the text (not including fcs bytes)
         putc(bte[L]);
         L++;
    1                                                     //end of while
    printf( 'W');                                         //add carriage return/line feed at the end
}
```

The variable L is used to walk through the data. The AX.25 protocol calls for the destination callsign to be included first followed by the source callsign. Note in the printout these are placed in the opposite order, since this is the de facto standard on the monitor function of most TNCs. Thus a packet with the text "Test" addressed to APRS from W2FS-2 via RELAY will show up here as:

W2FS-2 > APRS, RELAY: Test

The AX.25 protocol requires that the callsigns in the data contain exactly seven bytes (padded with spaces if necessary) and be left shifted by one bit. The code above undoes these changes. The SSID values (such as bte[131) must be anded with 7F to eliminate the "command/response" bit that is also carried on this byte. There are also two “reserved” bits on this byte that are supposed to be set to one. Conveniently (and perhaps by design) they result in the addition 30 (in hexidecimal) to the SSID value. That is, an SSID of 1 is appears as hex 31 after it is anded with 7F. This is convenient because the hex value 31 is produces an ASCII value of “1” when it is printed out.[1] The control, PID and FCS bytes are not printed.

Conclusion

To put all the pieces together, it is necessary to set up an infinite loop that causes the PIC continuously loop through these functions. In pseudocode it would look something like this:

```
Initialize I/O pins and set modem to receive
While (TRUE) {
        Look for the first flag
        Check for additional flags until you have the first byte of data
        Continue receiving data until you reach the terminating flag
        Check the FCS
        Route output to the serial port if the FCS checks
}
```

It seemed almost magical when I first burned this code into a 16F84 and watched the output flow across my screen. What once seemed to require so much hardware can now be accomplished with less than $12 worth of integrated circuits.

[1] The same trick works for all the digits 0 through 9. If one wanted to worry about decoding SSID’s greater than 9, some additional coding would be required.