PIC-et Radio: How to Send AX.25 UI Frames Using Inexpensive PIC Microprocessors

by John 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 implement AX.25 UI frames using inexpensive PIC microcontrollers. It is designedprimarily for those who wish to implement packet radio UI beacons for point to multipoint communications. The article assumes some knowledge ofprogramming concepts and PIC microprocessors. It also discusses the limitations that must be overcome in order to build a completely PIC based terminal node controller.

Keywords: AX 25, UI Frames, PIC Microprocessors

Introduction

Over the past several years there has been a growing number of applications that utilize packet radio communication via UI (unconnected) frames. These applications include the PACSAT broadcast protocol, APRS transmissions, and my own HamWeb file transfer protocol. All of these share the advantage that they provide communication of data from one source to many recipients simultaneously.

It is my view that we have only begun to exploit the possibilities of broadcast data in amateur radio applications. Furthermore, the efficiency of sending data from one source to many recipients is such that it can often overcome the limitations imposed by the relatively low data rates common in amateur radio applications. In short, broadcast protocols can breathe new life into otherwise "slow" data streams of 1200 baud or less. I believe the day will soon arrive that at 1200 baud, UI frames will be the most common means of transmitting amateur packet data.

Currently the principal means of communicating UI frames is to use a standard packet radio terminal node controller. While the price of these had fallen to the $100 range, by using small inexpensive microprocessors for these applications, the cost can be reduced even further.

This paper provides the basic information that is needed to construct AX.25 UI frames using inexpensive, easy to program Peripheral Interface Controller (PIC) chips produced by Microchip Inc. This paper assumes that the reader has some familiarity with PIC chips, how they work and how to program them. For more information on these subjects, see my recent article in QST.[1] PIC microcontrollers are available in single quantities for around $6. In larger quantities they are even less expensive. Using PIC microcontrollers, it is possible to build a device that can take serial data as an input and send it out as AX.25 UI frames using a PIC microcontroller for less than $20.

This paper is intended to be generic in nature. While a large variety of PIC microcontrollers are available, this article does not refer to any one particularly, instead the concepts described here apply to at least the entire series of Microchip midrange (series 16) controllers. Further, I did not want to limit this discussion to any one particular programming language. All of the examples presented in this article are in the C language. I have used this language because I think that the code is much

easier to read and understand than is assembly language (the most common means of programming PIC chips). But the basic constructions seen here can be applied in any language (as long as there is an appropriate PIC compiler available).

A Sample Data Stream

For the purpose of this discussion, assume that you want to broadcast the following simple packet:

W2FS -4> CQ, RELAY: Test

This is a simple frame where the source is W2FS-4, the destination is "CQ", there is one digipeter called “RELAY” and the text to be transmitted is “Test”. I selected this frame because it is a fairly simple packet, but includes most of the features that you would want to incorporate in a more complex UI frame. In a real world situation, the source of this data could be almost anything... a GPS data stream, an automated weather station, a DX Cluster, a highway traffic information server or perhaps even a generic file server like HamWeb.

What you see above is the packet as it would appear on your TNC. There is actually somewhat more to it than this. The AX.25 protocol divides the packet into seven sections as follows:

1. Flag (s)
2. Address
3. Control
4. Protocol Identifier (PID)
5. Information
6. Frame Check Sequence
7. Flag (s)

Most of the information below describing these fields is a summary of what appears in the AX.25 protocol description.* I have simplified this material to only describe what you need to know to send UI frames.

Flags

The flags are simply the hex value 7E (01111110 in binary) sent over and over again when no information is being transmitted. For example, when you set the TXdelay on your TNC to some value, it sends flags (7E's) over and over again for that period. These flags provide the receiver with a clear indication of when one packet has ended and the next is beginning. Thus, there must be at least one flag between two adjacent packets.

Address

The address field contains the destination (CQ, in this case) the source (W2FS-4 in this case), and up to eight digipeters (only RELAY, in this case). Each of “callsigns” in the address field must contain

exactly 7 characters, six for the callsign and 1 for the SSID. If a callsign is less than 6 characters long, it must be padded with blanks. In addition, the receiving station must have some way to determine when the address field has ended (since it could have anywhere from 0 to 8 digipeters in it). This is handled by shifting each of the bits one position to the left so that a 0 appears as the least significant bit. This bit is then set to a one for the SSID (seventh byte) or the last callsign in the address field. For example, the destination callsign would be encoded as follows:

Character	HEX Value from ASCII Table	Shifted Hex Value
C	43 (01000011)	86 (10000110)
Q	51 (01010001)	A2 (10100010)
Space	20 (00100000)	40 (01000000)
Space	20 (00100000)	40 (01000000)~
Space	20 (00100000)	40 (0 1000000)
Space	20 (00100000)	40 (0 1000000)

The seventh byte, the SSID, is a bit more tricky. Use the following bit pattern: 011 SSSSx where SSSS is the SSID (in binary) and x is a 0 if this is not the last callsign in the address field and a 1 if it is the last callsign in the address field. Since the destination address in this case has no SSID (that is, the SSID =0) and it is not the last callsign, the value should be: 01100000 or 60 in hex.

The next callsign is the source callsign, which in this case is W2FS-4. Using the rules established above we get the following string of bytes for this callsign:

W	2	F	S	Space	Space	SSID = 4
AE	64	8C	A6	40	40	68 (01101000)

There is only one digipeter so it is necessary to set the least significant bit of the SSID of that callsign to 1 :

R	E	L	A	Y	Space	SSID = 0
A4	8A	98	82	B2	40	61 (01100001)

Since UI frames are generally broadcast and not directed at any station in particular, the destination space is often wasted with something fairly meaningless (like CQ or APRS). The APRS MIC-E protocol capitalized on this by actually encoding useful position information within the destination address.

Control and PID

Since we are only doing UI frames, the control and PID bytes are pretty simple. Always use the hex value 3F for the control byte and the value FO for the PID byte.

Information

In this case the information being conveyed is simply the word “Test”. This consists of 4 hex bytes as follows:

54 65 73 74

The only thing to be careful about here is that these values are NOT shifted to the left as the address bytes are.

Frame Check Sequence

I was rather disappointed with the description of the Frame Check Sequence that is contained in the otherwise excellent AX.25 Protocol definition published by the ARRL. Here is what it says:

The flame-check sequence (FCS) is a sixteen bit number calculated by both the sender and receiver of a frame. It is used to insure that the frame was not corrupted by the medium used to get the frame from the sender to the receiver. It shall be calculated in accordance with IS0 3309 (HDLC) Recommendations.3

Fortunately there is considerable information on the Internet concerning the calculation of this value, which is more often referred to as a “CRC” than an FCS. In addition there is an excellent article in the September, 1986 issue of Byte Magazine on this subject.[4] Rather than review the theory of doing CRC calculations, I will provide a relatively simple mechanism for calculating this value, implemented in assembler and C. This code is included below.

Putting it All Together

Aside from the flags and the FCS (which will be calculated as we go along), our test packet can now be implemented as an array of 27 hex bytes as follows:

C	Q	SP	SP	SP	Sp	SSID=O	W	2	F	S	**Sp**	**Sp**	SSID=4
86	A2	40	40	40	40	60	AE	64	8C	A6	40	40	68

R	E	L	A	Y	Sp	SSID=O	Control	PID	T	e	S	t
A4	8A	98	82	B2	40	61	3F	FO	54	65	73	74

Or, as an initialized C array:

```
byte SendData[27] = {0x86, 0xA2, 0x40, 0x40, 0x40, 0x40, 0x60, 0xAE, 0x64, 0x8C, 0xA6,
                     0x40, 0x40, 0x68, 0xA4, 0x8A, 0x98, 0x82, 0xB2, 0x40, 0x61, 0x3F,
                     0xF0, 0x54, 0x65, 0x73, 0x74}
```

Doing it with a PIC Microcontroller

So aside from putting a few flags on the ends and adding the FCS, one simply needs to transmit this set of bytes in order to transmit the UI frame. There are a number of ways that this can be done. First, it is possible to get the PIC itself to actually send the data by simulating a sine wave with something called a resistor ladder. The theory is as follows. You take four (or more) output pins from the PIC and connect four (or more) different value resistors to them. You connect the other ends of these resistors together. Then you step through the output pins to get a rising and falling voltage that simulates a sine wave. This method is documented in a Microchip application note.'

A second method is to feed a data stream out of the PIC to a modem chip and have the modem chip generate the tones. There are a number of tone producing chips that will work for this application. Traditionally, packet TNCs have used the Texas Instruments TCM3 105. Because this chip is no longer being produced and is getting harder to find and more expensive, I didn't want to start down this road. Instead, I've been experimenting with the MX-COM MX614 chip, which cost about five dollars. It works quite well for this application. It appears that the Philips PCD33 1 1/ PCD33 12 chips will also work in this application and cost around $1.50, but I've not had occasion to use them yet.

Even using a modem chip to generate the tones, you can't simply send serial data out a PIC pin and expect it generate packets that can be decoded by other TNCs. There are three reasons for this:

1. Serial data contains start and stop bits. These are not used in packet radio transmissions.

2. The AX.25 protocol specifies that if five consecutive l's are received in a row, except in a flag byte, a zero should be added after the string of five ones. This is referred to as "bit-stuffing". If you are constructing the AX.25 frames yourself, you are responsible for bit stuffing.

3. Packet radio uses a modulation scheme called NRZI (Non-Return to Zero, Inverted). This means that the ones and zeros are not represented by high and low states (or tones). Rather, a zero is represented by a change in tone (if it was high, it goes low, if it was low, it goes high) while a one is represented by no change in tone. Together with bit-stuffing, this ensures that there will be a tone change at least every five bits, if not more often (except for flags). This helps the transmit and receive timing stay in sync.[6]

In order to implement this in a PIC microcontroller, therefore, you must take the incoming datastream, calculate the FCS, add the flags and route the stream of data to a subroutine that handles the transmission of the data taking into account the proper timing of the bits, bit stuffing, and NRZI. It's a tall order, but it can be easily handled by a $6 PIC chip!

On to the Code

Here is some stripped down C code that will send our sample array as an AX.25 UI frame. This code is written specifically for use with the PIC C compiler made by CCS, Inc. It is the least expensive C compiler currently available ($99). As noted above, the same logic flow could be applied to sending UI frames using assembler. An excellent assembler (MPASM) is available from Microchip, Inc. free of charge. Starting with an overview, the following function will send the packet that is contained in the array SendData.

```
void SendPacket(void) {

  fcslo=fcshi=0xFF;      //The 2 FCS Bytes are initialized to FF
  stuff = 0;             //The variable stuff counts the number of l's in a row. When it gets to 5
                         // it is time to stuff a 0.

  output_high(PTT);    //Turns on the microcontroller Pin that controlls the PTT line.

  flag = TRUE;           //The variable flag is true if you are transmitted flags (7E's) false otherwise.
  fcsflag = FALSE;       //The variable fcsflag is true if you are transmitting FCS bytes, false otherwise.

  for (i=0;i<20;i++) (SendByte(0x7E));          //Sends flag bytes. Adjust length for txdelay
                                                //each flag takes approx 6.7 ms
  flag = FALSE;                                 //done sending flags
  for(i=0;i<27;i++)  (SendByte(SendData[i]));   //send the packet bytes

  fcsflag = TRUE;                 //about to send the FCS bytes
  fcslo =fcslo^0xff;              //must XOR them with FF before sending
  fcshi = fcshi^0xff;
  SendByte(fcslo);                //send the low byte of fcs
  SendByte(fcshi);                //send the high byte of fcs
  fcsflag = FALSE                 //done sending FCS
  flag = TRUE;                    //about to send flags
  SendByte(0x7e);                 // Send a flag to end packet
  output_low(PTT);                //unkey PTT
 }
```

At the heart of the SendPacket function is the SendByte function which is called to send each of the 27 bytes in the SendData array. Here is the SendByte function:

```
void SendByte (byte inbyte) {
  int k, bt;
  for (k=0;k<8;k++) {                                   //do the following for each of the 8 bits in the byte
    bt = inbyte & 0x01;                                 //strip off the rightmost bit of the byte to be sent (inbyte)
    if ((fcsflag = = FALSE) & (flag = = false)) (fcsbit(bt));   //do FCS calc, but only if this
                                                        //is not a flag or fcs byte
    if (bt = = 0) (flipout());                                  // if this bit is a zero, flip the output state
      else {                                            //otherwise if it is a 1, do the following:
        stuff++;                                        //increment the count of consequtive l's
        if ((flag = = FALSE) & (stuff = = 5)){          //stuff an extra 0, if 5 l's in a row
                delay_us(850);                          //introduces a delay that creates 1200 baud
                flipout();                              //flip the output state to stuff a 0
        }//end of if
      }//end of else
```

```
    inbyte = inbyte<< 1;                    //go to the next bit in the byte
    delay_us(850);                          //introduces a delay that creates 1200 baud
  }//end of for
}/end of SendByte
```

Note that for each byte, the data is transmitted least significant bit first (that is from right to left, rather than from left to right). The function delay us is a routine shipped with the CCS C compiler. It is supposed to create a delay of 850 microseconds. You might think that this is too long a period a time since 1200 baud would normally require that each bit last exactly 833 milliseconds (1 sec/1200). The CCS timing routine is not exactly accurate, however. Experimentation revealed that a value for the CCS function of 850 resulting in timing that is correct for 1200 baud.

The flipout function changes the state on the output pin when a zero is being sent. It is as follows:

```
void flipout(){                             //flips the state of output pin a_1
        stuff = 0;                          //since this is a 0, reset the stuff counter
        if (! bit_test(port_a, 1)) (output_high(pin_a1));  //if the state of the pin was low, make it high.
                else (output_low(pin_al));  //if the state of the pin was high make it low
}
```

Finally, we need the routine that actually calculates the FCS. The FCS consists of two bytes, which I have called fcslo and fcshi. These are both initially set to FF. In this example the FCS will be calculated on a bit by bit basis. Algorithms exist that can calculate the FCS either a bit at a time or a byte at a time. Here is the calculation routine:

```
void crcbit(byte tbyte) {
#asm
  B C F  03,0
  RRF  fcshi,F                  // rotates the entire 16 bits
  RRF  fcslo,F                  // to the right
#endasm
  if (((status & 0x0 1)^(tbyte)) ==0x0 1) {
      fcshi = fcshi^0x84;
      fcslo = fcslo^0x08;
    }
}
```

The function parameter tbyte is either the byte 0000 or the byte 0001 corresponding to the value of the bit that is currently being transmitted. I have used three assembly language instructions in the beginning of this function (between #asm and #endasm) because there is no simple means of rotating a 16 bit value in the CCS C implementation. These three assembly language instructions simply move the 16 bit value one place to the right, with the previous least significant bit being placed in bit 0 of the status register in the PIC chip. The next line (the line that begins with if) performs an exclusive or (XOR) on this bit from the status register and the bit that is being transmitted (tbyte). If the result is equal to 1, the FCS is XOR-ed with the hex value 8408. If the result is equal to 0, this

latter step is not performed. Either way, the new value of the FCS is preserved in fcshi and fcslo. This procedure may seem rather arcane, but it does work. For a discussion of the theoretical reasons behind this procedure see the 1986 Byte Magazine article.

A PIC-based TNC?

From the above discussion it is clear that it is not all that difficult to send AX.25 frames using a PIC chip. There are a wide range of beacon type applications where such a device could be very useful. To go one step further, does this mean we could build a PIC-based TNC for very little money? Unfortunately this is not a trivial matter. My near term goal is to find a way to build a 1200 baud transmit module that will take a continuous data stream, convert it into packets and transmit it. A similar module on the other end of the link would undo the process. Using this mechanism you could transmit virtually any serial data stream from one point to many points using existing amateur radio transceivers without conventional TNCs. My intermediate term goal is to build a full duplex stand-alone 1200 baud (and then 9600 baud) KISS TNC using these inexpensive chips. If this could be accomplished it would have a myriad of applications including very inexpensive 9600 baud amateur satellite modems.

There are a number of hurdles to be overcome before any of this is possible. Starting on the transmit side, the basic problem is that the device must receive data via a serial link at the same time that it is transmitting data over the radio. The beacon style device discussed in this article takes existing data that it obtained from whatever source and transmits it using AX.25. For the purposes of this device, it is assumed that the data stream is not continuous. If the input data stream is continuous, however, while it is transmitting the first packet, it must also be accumulating data for the second packet over the serial link. Some buffering would also be required since there is not a bit for bit correspondence between the serial data stream (which includes start and stop bits) and the radio data stream (which includes no start and stop bits, but does include addresses, PID and control bytes, the FCS, etc.).

The receive side may actually be a bit more difficult. This is because the incoming packet must be received in its entirety in order to calculate the FCS before it is forwarded out the serial port since packets with incorrect FCSs should be discarded. Thus there must be enough buffer space to hold the entire packet. Most packet applications, including all of the amateur digital satellites, are limited to PACLENs of 255 characters. There are PIC microprocessors with enough on board memory to handle this available in the $10 range. However, some protocols are now using packet lengths in excess of 1K. No PIC contains this much on board storage, so some external SRAM would be required. This, in turn would involve using a PIC with a substantial number of I/O pins (for both the data and address lines).

Conclusion

Contrary to popular opinion, the most significant limitations in packet radio today are not technological. Amateur radio operators are only beginning to scratch the surface of the range of things that can be accomplished with the technology that is already available. In addition to the quest for faster speeds, we should also focus on new applications that can be developed with the slower

speed digital technologies that can piggyback on conventional FM radio channels. One key to doing this is to develop extremely inexpensive packet radio interfaces. PIC chips can provide a means of doing this. .

---

[1] Hansen, John A., " Using PIC Microcontrollers in Amateur Radio Projects" (QST, October, 1998).

[2] Fox, Terry L. *AX.25 Amateur Packet-Radio Link-layer Protocol, Version* 2.0 (Newington, Ct: ARRL, 1984).

[3] *ibid,* p.4.

[4] Morse, Greg "Calculating CRCs by Bits and Bytes" (Byte, Sept 1986, pp. 115-124).

[5] Microchip Application Note 655, "D/A Conversion Using PWM and R-2R Ladders to Generate Sine and DTMF Waveforms" .
http://www.microchip.com/Download/Appnote/Category/16CXX/00655a.pdf

[6] For a more complete description see: McDermott, Tom, *Wireless Digital Communications: Design and Theory.* (Tucson:
TAPR, 1966) pp. 121-126.