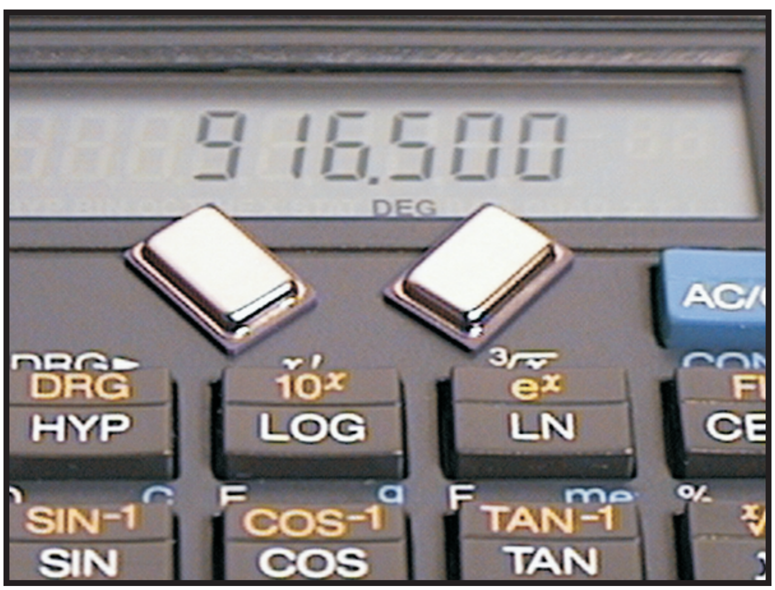
Discontinued

ASH Transceiver *Software* Designer's Guide

Updated 2002.08.07



RFM®

ASH Transceiver Software Designer's Guide

Drawings

1 Introduction

1.1 Why Can't I Just Use a UART?

Why can't I just use a UART and a couple of transistors to invert the TX and RX data signals to and from your ASH transceiver and get my application on the air? Well, you can if you don't need maximum performance and you make the necessary provisions in your software for the characteristics of radio communications. But, you are going to leave a lot of performance on the table. A radio link is a type of communication channel, and it has specific properties and characteristics, just as an ordinary phone line is another type of communication channel with its own properties and characteristics. To get usable data communications over your phone line, you place a modem between your PC's UART and the phone line. And to get good performance from your ASH radio link, you are going to need to put something more than a couple of transistors between the UART and the transceiver.

1.2 The Radio Channel – Magic and Imperfect

Radio is magic. It allows commands, data, messages, voice, pictures and other information to be conveyed with no physical or visible connection. A radio wave can penetrate most materials, and it can get around most barriers it cannot directly penetrate. It is arguably the most useful electronic communication channel so far discovered.

But from a software developer's point of view, a radio channel has some aggravating properties and characteristics. The good news is there are strategies for dealing with them.

1.2.1 Modeling a radio system

Figure 1.2.1 is a block diagram of a radio system. The antenna on the transmitter launches energy into the RF channel, and the antenna on the receiver retrieves some of the energy and amplifies it back to a useful level. No big deal, right? Well its no small deal either.

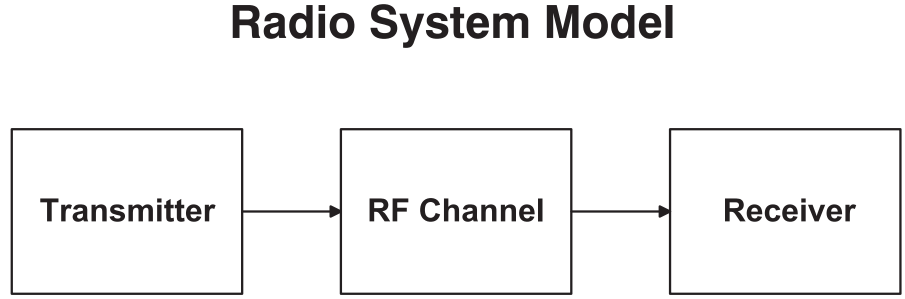


Figure 1.2.1

Receiver Signal Processing

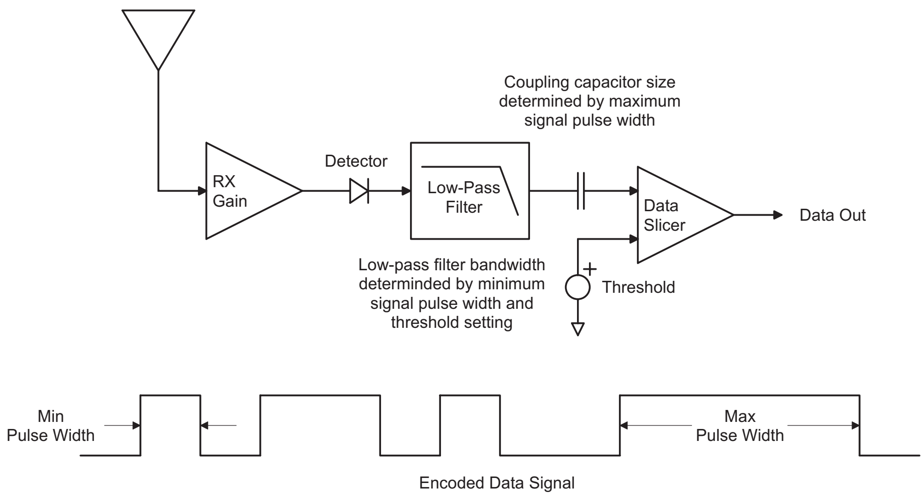


Figure 1.2.2

1.2.2 Data rate and bandwidth

Figure 1.2.2 is a generic block diagram of an RF receiver. This is where most of the action takes place in a radio communication system. There are two filters in this block diagram that you need to know about before you start writing code. The low-pass filter limits the rate that data can be sent through the radio system. And it also has a major impact on the range of the system. As you probably guessed, there is a trade-off here. For a fixed amount of transmitter power, you can transmit farther if you transmit at a lower data rate. The coupling capacitor in the block diagram creates a high-pass filter (in other words, your signal is AC coupled). You have to choose a data rate and use a data encoding scheme that lets your information flow successfully through these two filters. And if you get this right, these filters will greatly contribute to the overall performance of your system.

It is best to think in terms of the most narrow pulse (or most narrow gap) in your encoded signal, which must match the bandwidth of the low-pass filter, and the widest pulse in your encoded signal (or the widest gap), which must correctly match the time constant formed by the coupling capacitor and its associated circuitry. It is the minimum and maximum pulse widths (and gaps) in the encoded data that must be “in tune” with the filters in the receiver – not the underlying data rate.

1.2.3 Noise and interference

Unlicensed radio regulations, such as FCC regulation 15.249, limit the amount of RF power you can transmit to roughly 0.001% of the power dissipated in a 25 watt light bulb. But you only need to capture about 0.00000002% of this transmitted power level to receive properly encoded data at 2000 bps under typical conditions. Using decent antennas chest-high above the ground, this equates to more than one-eighth of a mile of range outdoors and much farther if one or both ends of the system are elevated.

There is a limit on how weak an RF signal can get and still convey information. This limit is due to electrical noise. One source of noise is everywhere present on the surface of the earth and is due to thermally-generated random electrical voltages and currents. Any device with electrical resistance becomes a source of this noise. Two other noise contributors are important in RF communications – semiconductor noise and attenuation. Semiconductor devices such as RF amplifiers contain noise generation mechanisms in addition to resistive thermal noise. Also, any component that attenuates a signal and is a thermal noise generator itself reduces the signal-to-noise ratio by the amount of the attenuation. An RF filter is an example of this type of component.

A signal transmitted through a radio system will be at its lowest power level when it reaches the first amplifier stage in the receiver. The noise added to the signal at this point places an upper limit on the signal-to-noise ratio that can be achieved by the receiver (for a given low-pass filter bandwidth). A good antenna helps keep the signal-to-noise ratio up by delivering more signal power. In addition, using a low-loss RF filter between the antenna and the first amplifier helps keep the signal-to-noise ratio up by minimizing signal attenuation. Using RF IC technology with low inherent RF semiconductor noise minimizes the amount of noise that is added to the signal beyond the ever-present resistive thermal noise. And yes, there are software tricks to take maximum advantage of whatever signal-to-noise ratio the hardware guys get for you.

Figure 1.2.3.1 shows the probability distribution, or histogram, of the noise voltage you would see at the base-band output of the ASH transceiver (R_{LPF} = 330 K). Notice that the noise has a Gaussian probability distribution. About 70% of the time the noise voltage will be between ±9 mV, or one standard deviation. Occasionally, noise spikes will reach ±18 mV, or two standard deviations. On rare occasions, spikes will reach ±27 mV, and on very rare occasions noise spikes will reach ±36 mV or more. So every now and then a noise spike or “pop” will occur that is strong enough to corrupt even a strong received signal. This characteristic of thermal noise (and thermal-like semiconductor noise) means that no RF channel can be perfectly error free. You have to plan for data transmission errors when designing your software.

From DC to frequencies much higher than RF, thermal noise exhibits a flat power spectrum. The power spectrum of semiconductor noise can also be considered flat across the RF bandwidth of a typical receiver. If you halve the bandwidth of the low-pass filter in a receiver, you halve the thermal noise power that comes through it. This is why you can transmit longer distances at a lower data rate. It allows you to reduce the bandwidth of

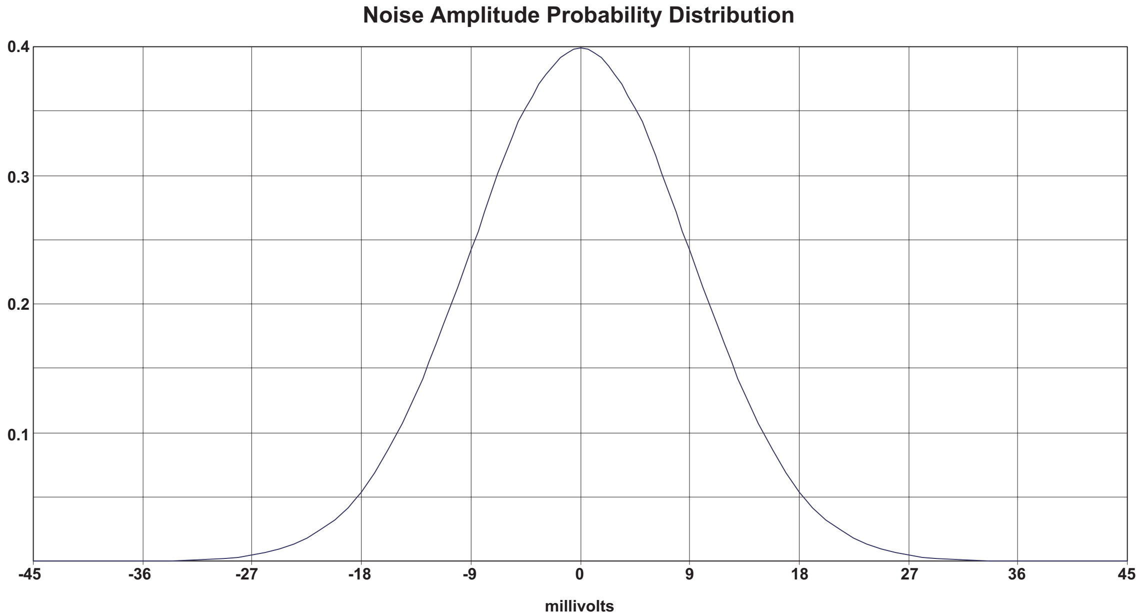


Figure 1.2.3.1

the low-pass filter so less noise gets through. You can then successfully recover data from a weaker received signal.

Lets go back and look at Figure 1.2.2 again. The job of the data slicer is to convert the signal that comes through the low-pass filter and coupling capacitor back into a data stream. And when everything is set up properly, the data slicer will output almost perfect data from an input signal distorted with so much noise that it is hard to tell there is a signal there at all. For the time being, assume the threshold voltage to the data slicer is zero. In this case, anytime the signal applied to the data slicer is zero volts or less, the data slicer will output a logic 0. Anytime the signal is greater than zero volts, the data slicer will output a logic 1. Through software techniques, you can assure that the signal reaching the data slicer swings symmetrically about 0 volts. Noise spikes, either positive or negative, that are slightly less than one half of the peak-to-peak voltage of the desired signal will not appear as spikes in the data output. The ability to recover almost perfect data from a signal with a lot of added noise is one of the main reasons that digital has overtaken analog as the primary format for transmitting information.

In the way of a preview, look at Figures 1.2.3.2, 1.2.3.3, 1.2.3.4 and 1.2.3.5, which are simulations of a radio system with various amounts of noise added to the signal. The top trace in Figure 1.2.3.2 is the signal seen at the input to the data slicer.

The horizontal line through this signal is the slicing level. Notice that the signal droops down as it starts from left to right, so that is swinging symmetrically around the slicing level by about the fifth vertical grid line. This is the transient response of the base-band coupling capacitor, and its associated circuitry, as it starts blocking the DC component of the received signal. The steady 1-0-1-0… bit pattern seen to the left of the fifth grid line is a training preamble. It sets up the slicing symmetry. To the right of the fifth grid line there is a 12 bit start symbol and then the encoded message bits, etc. You will notice that

Signal Reception with No Noise

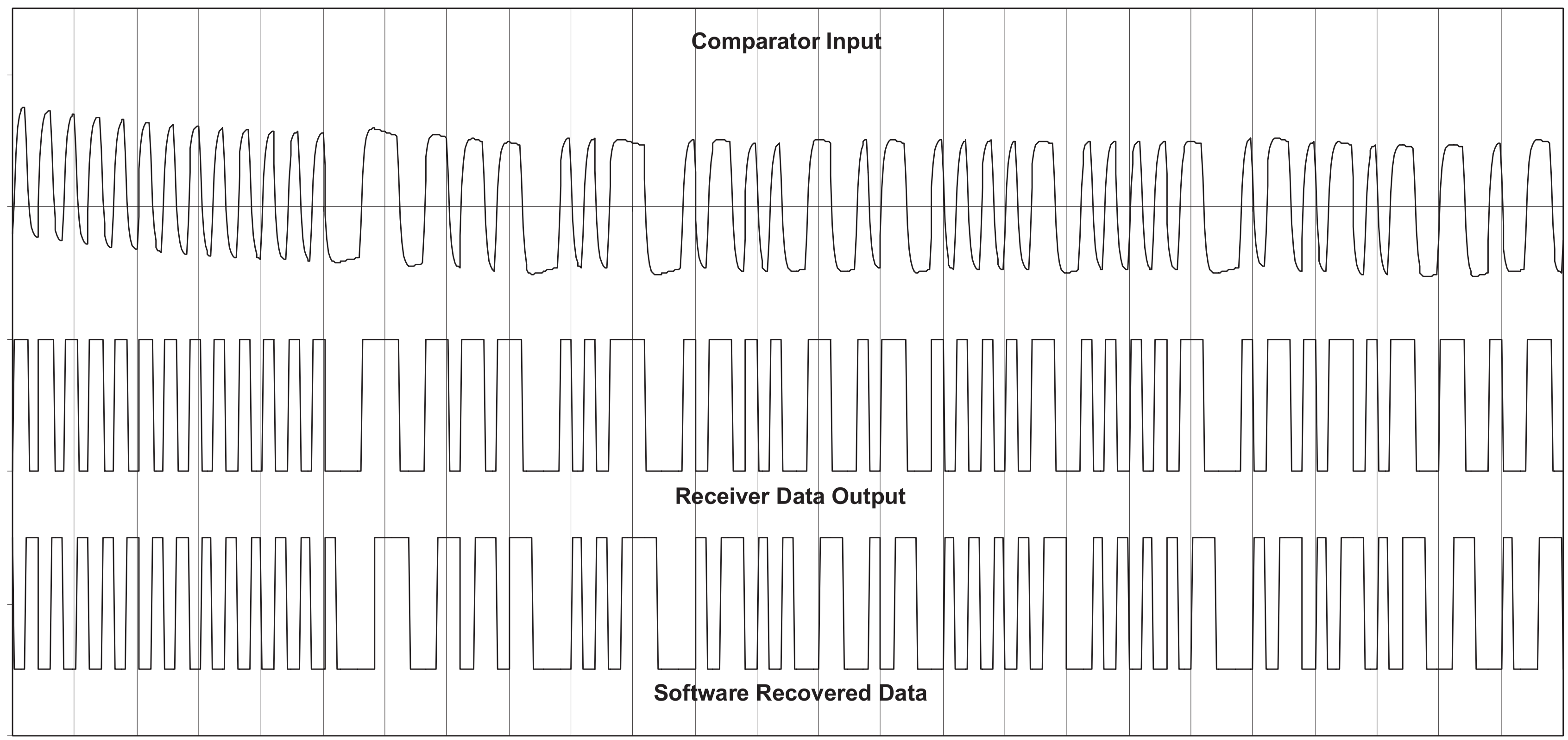


Figure 1.2.3.2

Signal Reception with Moderate Noise

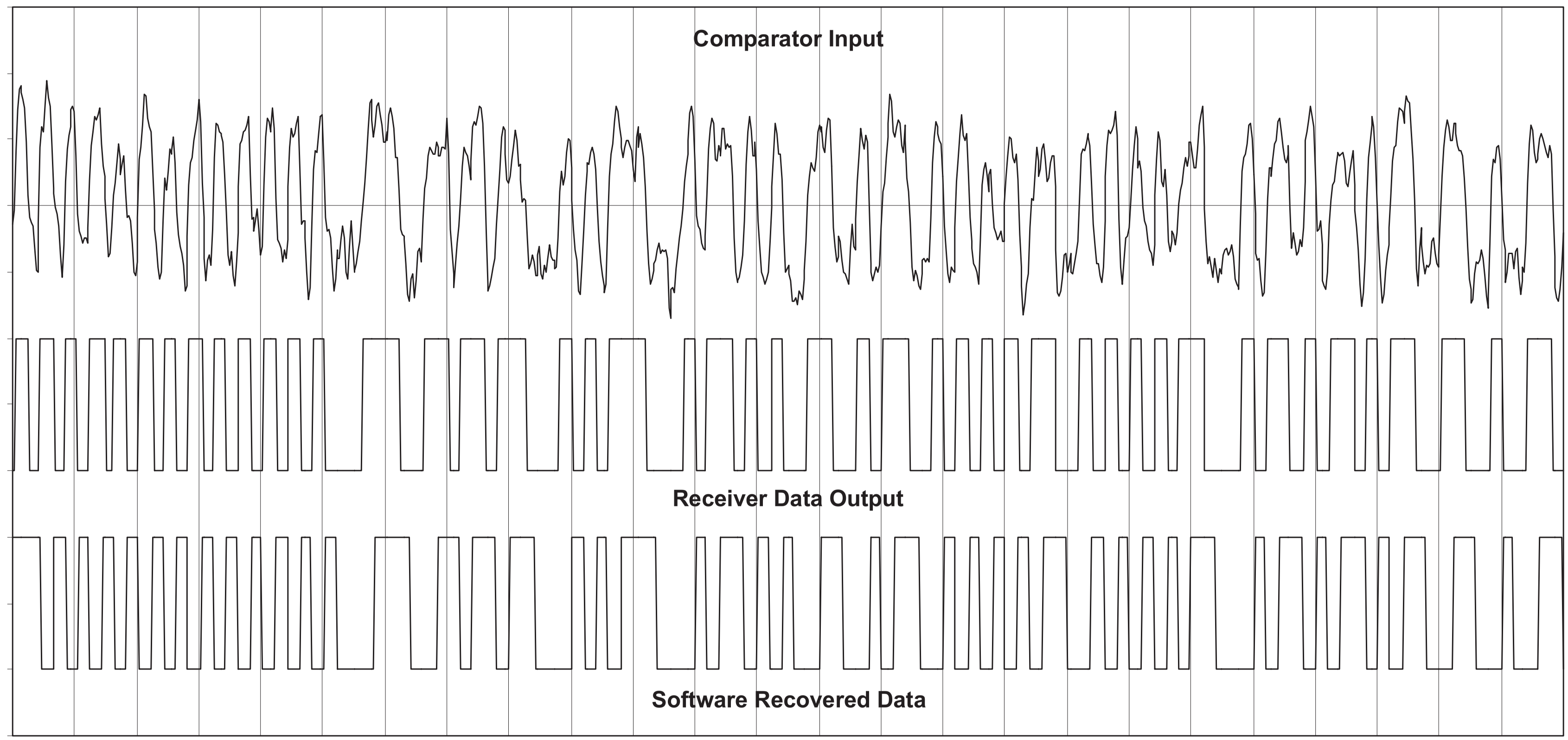


Figure 1.2.3.3

Signal Reception with Heavy Noise

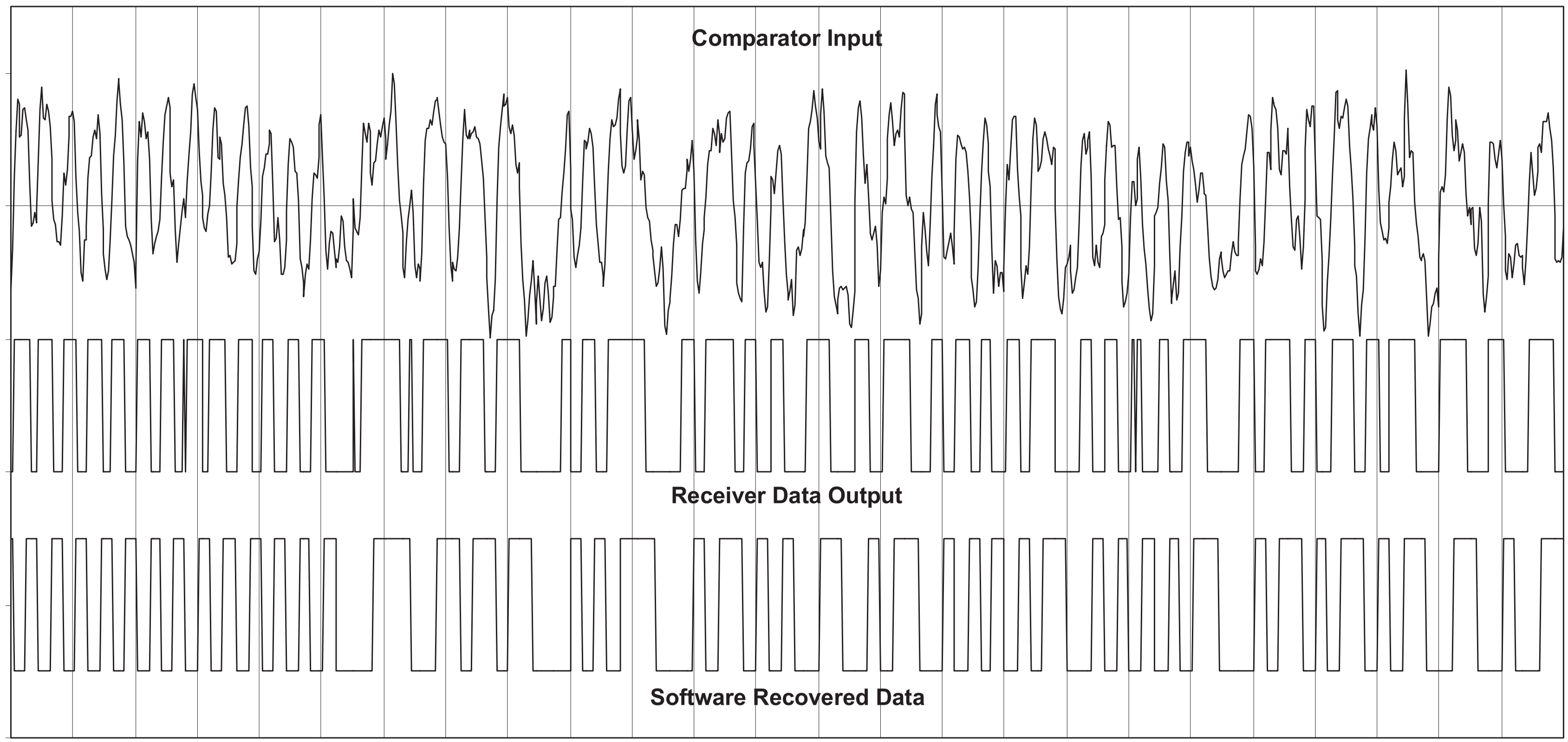


Figure 1.2.3.4

Reception with Heavy Noise (expanded scale)

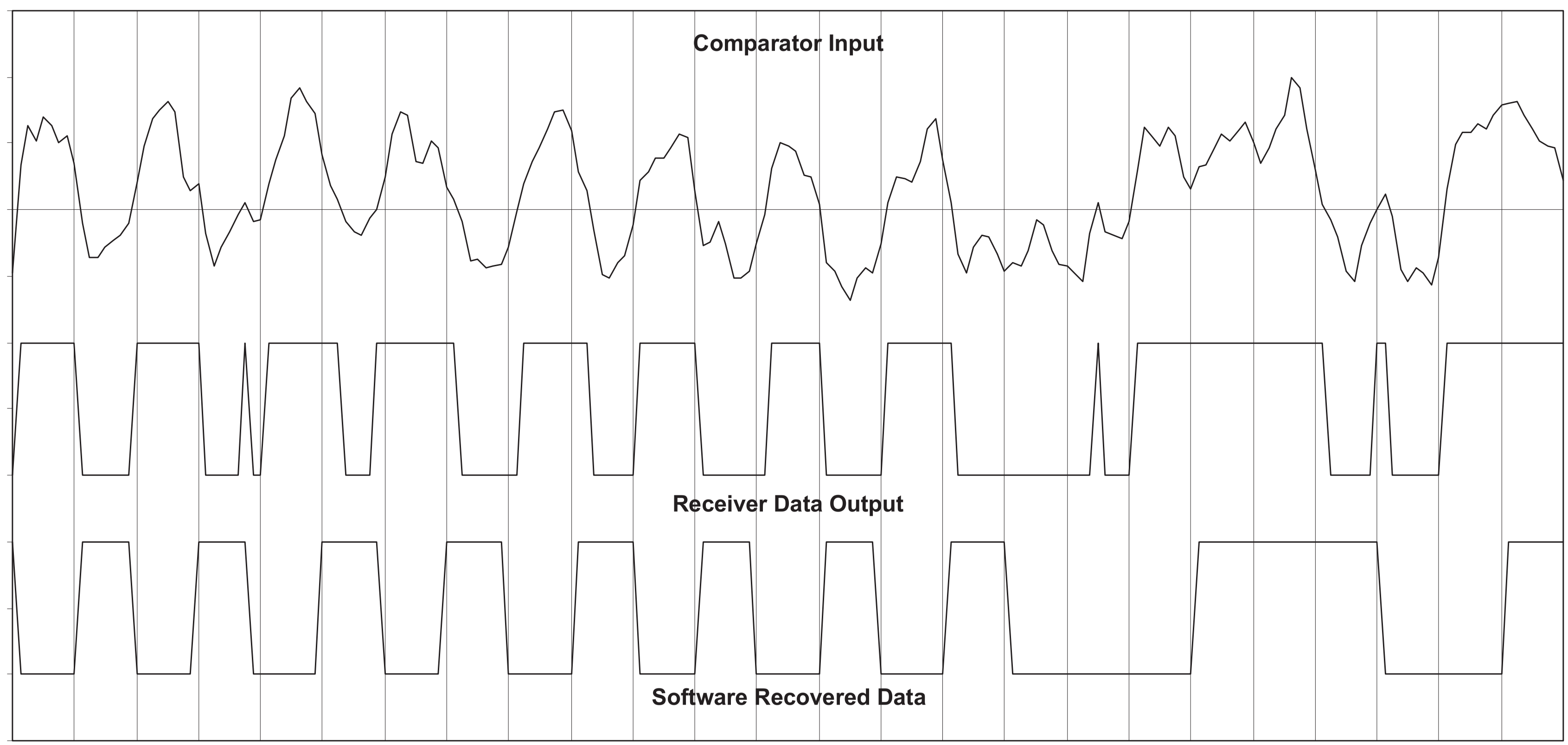


Figure 1.2.3.5

the signal has been “rounded off” so that the 1-0-1-0… bit sequences almost look sinusoidal. This shaping effect is due to the low-pass filter. If you set the bandwidth of the filter too low for a given data rate, it will start seriously reducing the amplitude of these 1-0-1-0… bit sequences and/or smearing them into each other.

The output of the data slicer is the middle trace, and the output of the software recovery subroutine is the bottom trace. Notice that the bottom trace is shifted to the right one bit period. This is because the software “studies” the receiver data output for a complete bit period before estimating the bit value. It will soon become apparent why this is done.

Figure 1.2.3.3 shows the same signal with a moderate amount of noise added. You now have to look at the top trace carefully to see the data pattern (look right at the slicing level). The middle trace shows the output of the data slicer, which has recovered the data accurately other than for some jitter in the width of the bits. The data recovered by the software matches the middle trace again, shifted one bit period to the right.

Figure 1.2.3.4 shows the signal with heavy noise added. The data pattern has become even more obscure in the top trace. With this much noise, the output from the data slicer shows occasional errors. Note that the software subroutine has been able to overcome these errors by deciding the most likely bit value at the end of each bit period. Figure 1.2.3.5 is a section of 1.2.3.4 on an expanded scale to show more bit-by-bit detail.

Interference is defined as an unwanted RF signal radiated by another system (RF or digital). Like noise, interference that is not too strong can be eliminated by the data slicer and/or software subroutine. Of course, the data has to be encoded so that it swings symmetrically around the slicing level to get maximum noise and interference rejection.

1.2.4 Indoor RF propagation

It is intuitive that the farther away from a transmitter you get, the less power you can capture from it with your receiver. This is what you would see in free space, far away from the ground and other physical objects. But on the ground, and especially indoors, you will find that the signal strength varies up and down rapidly as the distance between the transmitter and the receiver is steadily increased. The reason this happens is both good news and bad news. It turns out that the radio waves from the transmitter antenna are taking many different paths to the receiver antenna. Radio waves strongly reflect off the ground and off metal surfaces as light reflects off a mirror. And radio waves will also partially reflect off non-metallic walls, etc. as light does off a window pane. The good news is that all this bouncing around allows radio waves to diffuse around barriers they cannot directly penetrate. The bad news is that all the bouncing around makes the RF power you receive vary rapidly (flutter) as you move around and hit small reception “dead spots”. You can even see reception flutter if you stand still and other people, vehicles, etc. move nearby. Any radio system that operates near the ground (mobile phones, wireless microphones, broadcast radios in cars, etc.) must deal with this multi-path flutter problem. And yes, it is a consideration when you start writing your code.

Studies on indoor propagation show that you will find only a few spots in a room that have really bad reception, and these severe “dead spots” tend to occupy a very small space. Mild dead spots are far more common, and you will also find some places where reception is especially good. As a rule of thumb, you need 100 times more transmitted power indoors than in free space to get adequate reception at comparable distances. This is called a 20 dB fading margin, and it provides about 99% coverage indoors. If you are in a severe dead spot at UHF frequencies, moving just an inch or two gets you out of it.

When you look at a professional wireless microphone, you will notice that the base unit is equipped with a “rabbit ear” antenna. Actually, there are two separate antennas and two separate receivers in the wireless microphone base unit, with the antennas at right angles to each other. This arrangement provides diversity reception, which greatly mitigates the dead spot problem indoors. Since the paths between the two base station antennas and the microphone are different, it is unlikely that the microphone will hit a dead spot for both antennas at the same time. Mobile phone base stations also use diversity reception as do many other radio systems, including a number of ASH transceiver systems.

1.2.5 Regulatory considerations

Systems based on ASH transceiver technology operate under various low power, unlicensed UHF radio regulations. From a software point of view, the main differences in these regulations are the maximum power you are allowed to transmit, and the allowed transmitter duty cycle. European regulations (ETSI) allow the most transmitted power, American regulations are in the middle, and Japan allows the least transmitted power. At lower power levels, you have to transmit at a low data rate to get a useful amount of range. At higher power levels you have more flexibility.

Duty cycle refers to the percentage of time each transmitter in your system can be on. Some regulations, such as FCC 15.249 place no restrictions on duty cycle. Some bands in Europe also have no current duty cycle limit - for example, the 433.92 MHz band. Other bands in Europe do have a duty cycle limit. At 868.35 MHz, the duty cycle limit is 36 seconds in any 60 minute interval. Duty cycle requirements influence the choice of band to operate in, and the design of your software. RFM’s web site has links to many radio regulatory sites. Be sure to thoroughly familiarize yourself with the regulations in each geographical market for your product. We have seen cases where a customer had to redo a well-engineered system to accommodate a regulatory subtlety.

2 Key Software Design Issues

There are at least four key issues to consider in designing ASH transceiver software. You may identify others depending on the specifics of your product’s application. It is worth giving it some thought before you start designing your code.

2.1 Fail-Safe System Design

Most unlicensed UHF radio systems operate with few interference problems. However, these systems operate on shared radio channels, so interference can occur at any time and at any place. Products that incorporate unlicensed UHF radio technology must be designed so that *a loss of communications due to radio interference or any other reason will not create a dangerous situation, damage equipment or property, or cause loss of valuable data*. The single most important consideration in designing a product that uses unlicensed radio technology is safety.

2.2 Message Encoding for Robust RF Transmission

Look at Figure 1.2.2 again, and note the threshold input to the data slicer. When you set the threshold voltage to a value greater than zero you move the slicing level up. This provides a noise squelching action. Compare Figures 2.2.1 and 2.2.2. In Figure 2.2.1, the threshold is set to zero. With no signal present, noise is continuously present at the receiver data output, and at the output of the software data recovery routine. Software downstream of the data recovery subroutine has to be able to distinguish between noise and a desired signal. Figure 2.2.2 shows the effect of adding a moderate threshold. Notice that just a few noise spikes appear at the receiver data output and no noise spikes come out of the software data recovery routine (it could still happen occasionally). As we raise the threshold more, even fewer noise spikes will appear at the receiver data output. Don't expect to eliminate all noise spikes – noise amplitude has that Gaussian probability distribution we discussed earlier. Even using a very heavy threshold, you have to plan for noise spikes now and then, as well as strong bursts of interference.

As you raise the threshold from zero, you reduce the receiver's sensitivity to desired signals, and you make it more vulnerable to propagation flutter. If you need all the range and system robustness possible, you will want to use little or no threshold. On the other hand, using a threshold can reduce the amount of work your software has to do on data recovery. This allows you to support a higher data rate with the same processing power, or reduce average processor current consumption in applications where this is critical. If you decide to use an ordinary UART on the radio side, a strong threshold is a must. Also, some remote control decoder chips will not tolerate much noise.

The ASH transceiver is equipped with two thresholds, DS1 and DS2. DS1 works basically as shown in Figures 1.2.2, 2.2.1, and 2.2.2. DS2 is used in conjunction with DS1 and its primary job is to support high data rate transmissions. The details on how to adjust these thresholds are given in the ASH Transceiver Designer's Guide, Sections 2.7.1 and 2.7.2.

Your message encoding strategy and several adjustments on the ASH transceiver depend on whether you use a threshold, and on how strongly the threshold is set. Let's start with the "no threshold" case, which offers the best potential performance. Referring to Figure 1.2.3.2, we start the transmission with a 1-0-1-0… training preamble. This preamble needs to be long enough to establish good signal slicing symmetry at the input to the

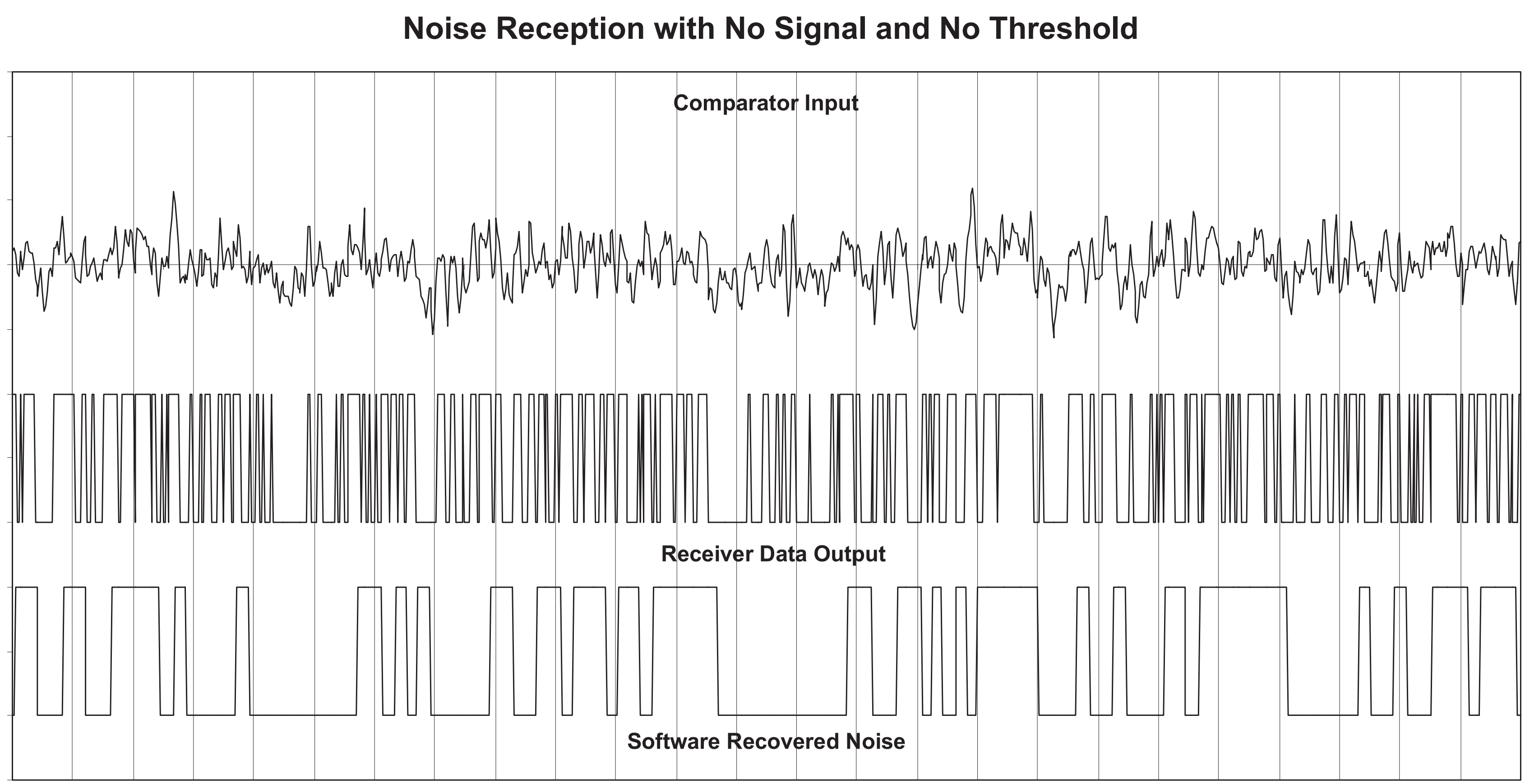

Noise Reception with No Signal and No Threshold
Comparator Input
Receiver Data Output
Software Recovered Noise

Figure 2.2.1

Noise Reception with No Signal and Moderate Threshold

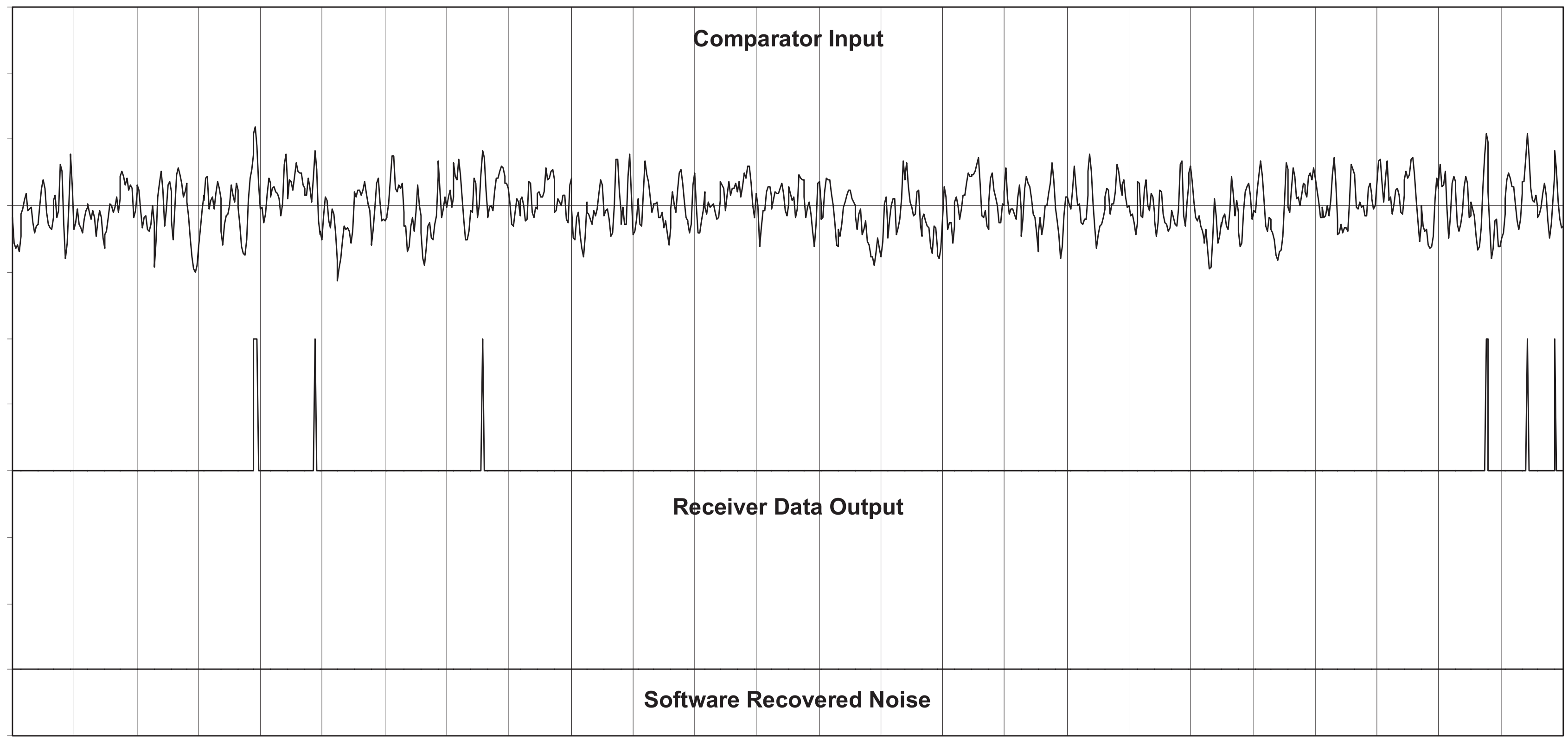


Figure 2.2.2

comparator. The preamble is followed by a specific pattern of bits that will not occur anywhere else in the message. This pattern is often called a "sync vector", and makes it possible to distinguish data from noise with high reliability (the sync vector is 12 bits in this example). The balance of the message consists of encoded data and error detection bits.

The purpose of encoding your data is to maintain good slicing symmetry at the input to the comparator. This is called DC-balanced encoding. Look at Figure 1.2.3.2 again. There are five bit periods between each vertical grid line. Notice that you will not find more than three 1 or 0 bits in a row in the data shown, and that there are always six ones and six zeros in any sequence of 12 bits. This is because each message byte has been encoded as 12 bits, always with six ones and six zeros, and with no more than four bits of the same type in a row for any combination of adjacent encoded characters. This is one type of coding that maintains good dynamic DC balance, and is similar to techniques used in fiber-optic data transmissions. Another popular encoding scheme is Manchester encoding, which encodes each 1 bit in the message as a 1-0 bit sequence, and each 0 bit in the message as a 0-1 bit sequence. Both 12-bit encoding and Manchester encoding work well. Manchester encoding has a maximum of two bits of the same type in a row, but requires 16 bits to encode a byte. 12-bit encoding can have up to 4 bits of the same type in a row, and requires, of course, 12 bits to encode a byte. By the way, your start vector should also be dynamically DC balanced in most cases.

The data rate and the encoding scheme you use affects two adjustments on the ASH transceiver (or vice versa). The most narrow pulse or gap in your encoded data sets the low-pass filter bandwidth. For the two encoding schemes we have discussed, this is one encoded bit period. Once you know the bit period, Section 2.5 in the ASH Transceiver Designer's Guide explains how to set the low-pass filter bandwidth. The widest pulse or gap in your encoded data sets the value of the coupling capacitor. Once you know the maximum number of 1 bits or 0 bits that can occur in a row, you know the width of the maximum pulse or gap that can occur in your encoded data. Section 2.6 in the ASH Transceiver Designer's Guide explains how to determine the coupling capacitor value and the required training preamble length from the maximum pulse or gap width.

Trying to send data without encoding is generally a disaster. Without a threshold, any long sequence of 1's or 0's in your data will charge or discharge the coupling capacitor, unbalancing the symmetry of the signal into the data slicer and ruining the noise rejection performance.

When you use one of the data encoding schemes discussed above with no slicer threshold, the coupling-capacitor transient response automatically adjusts the slicing symmetry as variations occur in received signal strength. This greatly improves system robustness to signal flutter. You usually want to make the coupling-capacitor value no larger than needed, so that fast signal fluctuations can be followed.

Let's now consider message encoding schemes and ASH transceiver adjustments when a threshold is used. Again, a threshold trades-off sensitivity and flutter robustness for less noise in the no-signal condition. If you are using a strong threshold, you may decide you do not need a training preamble or start vector (this depends on the way you design your code). But if you are using AGC and/or data slicer DS2 in your ASH transceiver, you will need at least one 1-0-1-0… preamble byte for training these hardware functions. The threshold in DS1 has a built-in hysteresis. When the input voltage to the data slicer exceeds the threshold level, DS1 will output a logic 1, and it will continue to output a logic 1 until the input voltage swings below zero. The DC-balanced data encoding methods already discussed work satisfactorily with the DS1 hysteresis. Again, once you know the bit period of your encoded data, Section 2.5 in the ASH Transceiver Designer's Guide explains how to set the low-pass filter bandwidth. Note that a larger bandwidth is recommended for the same bit period when a threshold is used. Using the coupling capacitor value as determined in Section 2.6 of the ASH Transceiver Designer's Guide is a good default choice. When you use a threshold, 1 bits tend to drop out of weak and/or fluttering signals at the data slicer. Message patterns that contain a few less 1 bits than 0 bits work somewhat better with a strong threshold than classical DC-balanced codes. In some cases you may work with encoder and decoder chips designed to send command codes. Some of these chips send code messages with short preambles and relatively large gaps between the messages. These chips often work better if you use a moderate threshold and a relatively large coupling capacitor, so it is worth doing some experimenting.

2.3 Clock and Data Recovery

The clock and data recovery techniques used at the receiver are critical to overall system performance. Even at moderate signal-to-noise ratios, the output of the data slicer will exhibit some jitter in the position of the logic transitions. At lower signal-to-noise ratios, the jitter will become more severe and spikes of noise will start to appear at the data slicer output, as shown in Figure1.2.3.5. The better your clock and data recovery techniques can handle edge jitter and occasional noise spikes, the more robust your radio link will be. There is some good news about edge jitter due to Gaussian noise. The average position of the logic transitions are in the same place as the noise-free case. This allows you to use a phase-locked loop (PLL) that hones in on the average position of the data edges for clock recovery. Once your clock recovery PLL is lined up, you can use the logic state at the middle of each bit period, or the dominant logic state across each bit period as your recovered bit value. Testing mid-bit works best when the low-pass filter is well-matched to the data rate. On the other hand, determining the dominant logic state across a bit period can improve performance when the low-pass filter is not so well matched. The dominant logic state is often determined using an "integrate and dump" algorithm, which is a type of averaging filter itself.

It is possible to use simple data recovery techniques for less demanding applications (close operating range so the signal-to-noise ratio is high). The standard protocol software that comes in the DR1200-DK, DR1201-DK and DR1300-DK Virtual Wire® Development Kits uses a simplified data recovery technique to achieve air transmission rates of 22.5 kbps with a modest microcontroller. And yes, ordinary UARTs are being used successfully in non-demanding applications. But a word of caution. It appears the UARTs built into some microcontroller chips really don't like even moderate edge jitter. If you are considering using a built-in UART on the radio side, do some testing before you commit your design to that direction.

About now you may be wondering if anybody builds an “RF UART”, which is designed for low signal-to-noise ratio applications. The IC1000 discussed below is one example of this concept.

2.4 Communication Protocols

So far, we have discussed message encoding techniques for robust RF data transmission, and clock and data recovery techniques that can work with some noise-induced edge jitter and occasional noise spikes. Even so, transmission errors and drop outs will occur. The main job of your communication protocol is to achieve near-perfect communications over an imperfect RF communication channel, or to alarm you when a communication problem occurs. And channel sharing is often another requirement.

A protocol is a set of standard structures and procedures for communicating digital information. A complete protocol is often visualized as a stack of structures and procedures that are very specific to the communication hardware and channel characteristics at the bottom, and more general-purpose and/or application oriented at the top.

Packet-based protocols are widely used for digital RF communications (and for sending data on many other types of communications channels.) Even simple command transmissions usually employ a packet-style data structure.

2.4.1 Digital command transmissions

In addition to ASH transceivers, RFM’s second-generation ASH radio product line includes transmitter and receiver derivatives for one-way RF communications. Most one-way command applications are actually two-way; RF in one direction and audible or visual in the other direction. For example, you press the “open” button until you see the garage door or gate start moving. The data encoding and data recovery techniques discussed above can be used to build a robust one-way RF communications system. But often, off-the-shelf command encoder and decoder ICs are used. Among the most popular are the Microchip KeeLoq™ ICs. Figure 2.4.1 shows RFM’s suggested application circuit for second-generation ASH receivers driving KeeLoq™ decoders. You can usually derive enough information from the data sheets of other encoder and decoder ICs to calculate the component values to use with second-generation ASH receivers. The calculations are the same as discussed in the ASH Transceiver Designer’s Guide.

There is a growing trend to replace one-way RF communication links with two-way links for added system integrity. This is especially true for one-way RF communication links that are not activated by the user. Wireless home security systems are one example.

ASH Receiver Application Circuit
KeeLoq Configuration

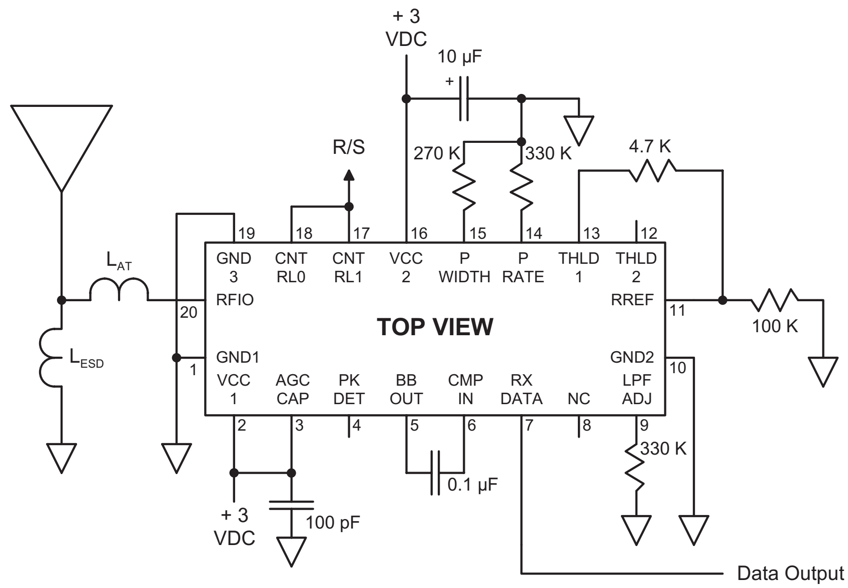


Figure 2.4.1

2.4.2 Data transmissions using packet protocols

A packet structure generally includes a training preamble, start symbol, routing information (to/from, etc.) packet ID, all or part of a message, and error detection bits. Other information may be included depending on the protocol. Communications between nodes in a packet-based system may be uncoordinated (talk when you want to) or coordinated (talk only when it is your turn). In the case of uncoordinated transmissions, packet collisions are possible. Theorists note that the collision problem limits the throughput of an uncoordinated channel to about 18% of its steady one-way capacity. Coordinated transmissions have higher potential throughput but are more complex to code. Many applications that use ASH radio technology transmit relatively infrequently, so uncoordinated transmissions work very successfully.

In both uncoordinated and coordinated systems, transmission errors can and will occur. An acknowledgment (ACK) transmission back to the sending node is used to confirm that the destination node has received the packet error free. Error-detection bits are added to a packet so the destination node can determine if the packet was received accurately. Simple parity checks or checksums are not considered strong enough for error checking RF transmissions. The error-detection bits added to the end of a packet are often called a frame check sequence (FCS). An FCS is usually 16 to 24 bits long, and is generated using a cyclic redundancy code (CRC) method. IBM developed such a code many years ago for their X.25 protocol and it is still widely used for RF packet transmissions. The ISO3309

Standard details the generation of this error detection code, and it is used in the protocol code example below.

It is time to bring up the real challenge in designing and writing protocol software. Events can happen in any sequence, and data coming into the protocol software can be corrupted in any bit or in every bit (remember, short packets work best on a low signal-to-noise radio channel). It is worth doing a careful “what if” study relevant to your protocol and your application before doing the detailed design and coding of your software. Consider how you can force unlikely sequences of events in your testing. Thorough front end planning can avoid a lot of downstream problems.

3 IC1000 “Radio UART”

RFM has introduced the IC1000 to support fast-track product development cycles using ASH radio technology. The IC1000 implements the clock and data recovery tasks that often constitute a lot of the learning curve in your first RF protocol project. The IC1000 is designed to operate with no threshold, which is the key to good system sensitivity.

3.1 IC1000 Description

The IC1000 is implemented in an industrial temperature range PIC12LC508A-04I\SN microcontroller using internal clocking. Nominal operating current is 450 µA, consistent with the low operating current emphasis of the second-generation ASH radio product line. The IC1000 is provided in a miniature eight-pin SMT package.

3.2 IC1000 Application

A typical IC1000 application is shown in Figure 3.2.1. The data (slicer) output from the second-generation ASH transceiver is buffered by an inverting buffer and is applied to Pin 3 of the IC1000 and the Data In pin of the host microprocessor. When the IC1000 detects the presence of a specific start-of-data pulse sequence, it outputs a Start Detect pulse on Pin 2. This pulse is applied to an interrupt pin on the host processor. The IC1000 generates data clocking (data valid) pulses in the middle of each following bit period using an oversampled clock extraction method. The IC1000 is designed to tolerate continuous input noise while searching for a start-of-data pulse sequence.

The IC1000 supports four data rates - 2400, 4800, 9600, and 19200 bits per second (bps). The data rate is selected by setting the logic input levels to Pin 6 (Speed 1) and Pin 7 (Speed 0). Please refer to the IC1000 data sheet for additional information.

4 Example Data Link Layer Protocol

The data link protocol discussed below is tuned for high-sensitivity, low data rate requirements. The protocol code is designed to run on the ATMEL AT89C2051 microcontroller used in the DR1200-DK/DR1200A-DK Series Virtual Wire® Development Kits. The “A” version kits (DR1200A-DK, etc.) ship with this software and require no hardware

Typical IC1000 Application

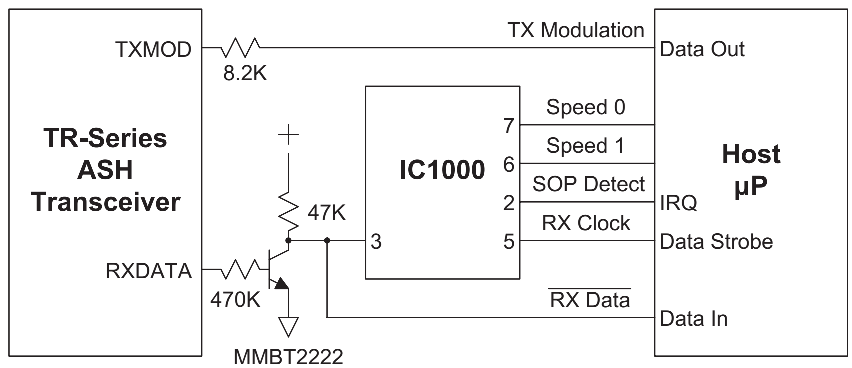


Figure 3.2.1

modifications. It is necessary to replace the radio boards used in the standard kits with "A" version radio boards before using this code, or to modify the standard radio boards as detailed below. Figure 4.1 shows the circuit modification used between the ASH transceiver base-band output, Pin 5, and the comparator (data-slicer) input, Pin 6. Figure 4.2 shows how these components are installed and their values. This modification reduces the

ASH Transceiver Application Circuit
Low Data Rate OOK

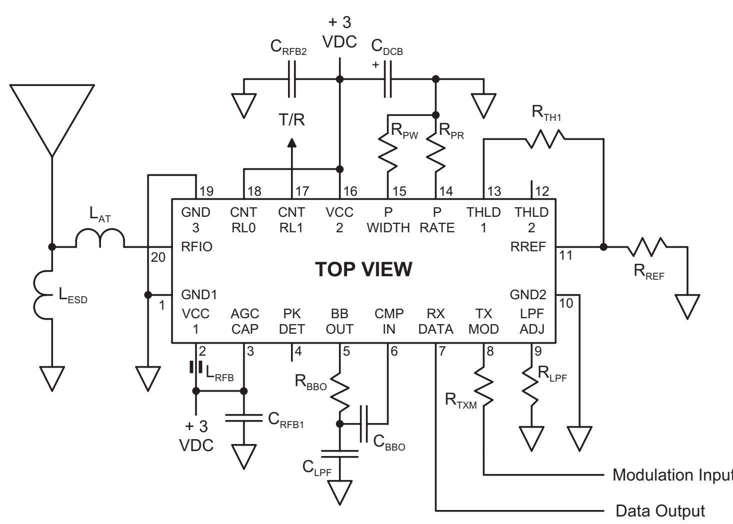


Figure 4.1

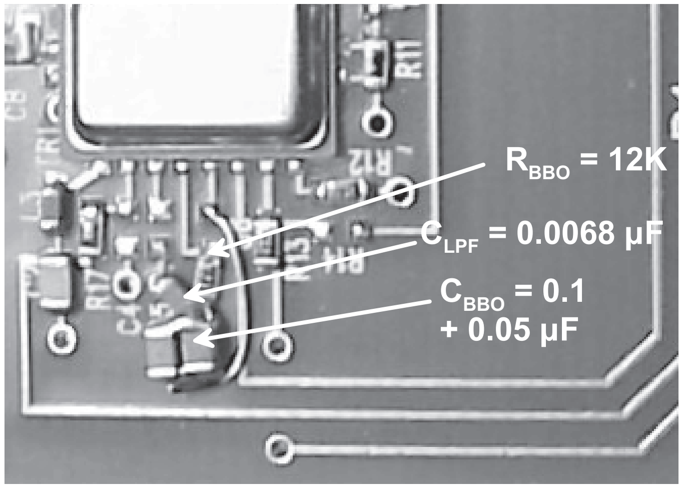


Figure 4.2

noise bandwidth of the receiver. In addition, R9 on the DR1200, DR1201 and DR1300 radio boards should be changed to a zero-ohm jumper (no DS1 threshold). R12 should be changed to 330 K on all three radio boards. Note that the DR1200A, DR1201A and DR1300A already incorporate these modifications.

4.1 Link Layer Protocol Source Code

The link layer protocol is implemented in 8051 assembly language and the source, DK200A.ASM (RFM P/N SW0012.V01), is compatible with the popular TASM 3.01 shareware assembler. You can get TASM 3.01 at www.rehn.org/YAM51/files.shtml.

By the way, this "A" link layer protocol uses the programming pins differently than the protocol supplied in the standard development kits. See Picture 4.3. Placing a jumper next to the "dot" end (ID0) enables the AutoSend mode (do this on one protocol board only). Placing a jumper at the far end (ID3) strips the packet framing and header characters off

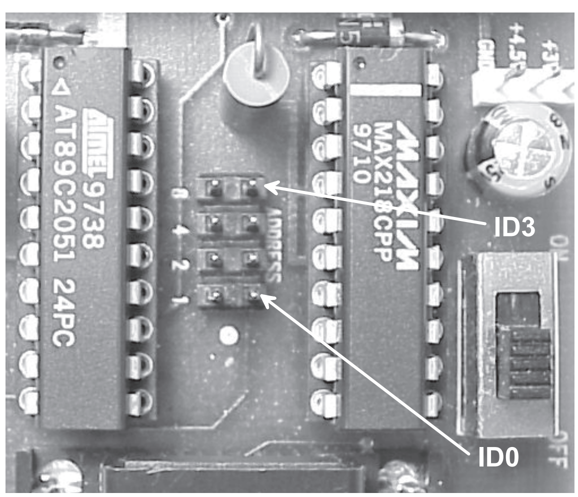


Figure 4.3

received packets. This can be handy for driving small serial printers, etc. You do not use jumpers to set the FROM address with this protocol.

Details of the packet and byte structures used by the protocol are shown in Figure 4.4. The host-protocol packet structure begins and ends with a 0C0H framing character (FEND) that cannot be used elsewhere in the packet. For example, you cannot use 0C0H in the TO/FROM address byte. This will otherwise not be a problem using seven-bit ASCII message characters. Eight-bit data can be sent using seven-bit ASCII characters to represent numerical values, or a framing character substitution scheme like the one used in the Internet SLIP protocol can be employed. The framing character helps deal with the "non real time" nature of serial ports on your typical PC. The host-protocol packet structure within the frame includes the TO/FROM address byte, with the high nibble the TO address and the low nibble the FROM address. The ID byte indicates which packet this is. Each packet can hold up to 24 additional message bytes. As mentioned, short packets should be used on radio channels.

Framing characters are not needed in the transmitted packet structure as the protocol is real time on the radio side. The transmitted packet structure beings with a 1-0-1-0… preamble which establishes good signal slicing symmetry at the input to the radio comparator and then trains the clock and data recovery processes in the software. The preamble is followed by a 12-bit start symbol that provides good discrimination to random noise patterns. The number of bytes in the packet (beyond the start symbol), the TO/FROM address, packet ID, message bytes and FCS then follow. The start symbol and all bytes following are 12-bit encoded for good dynamic DC balance.

Packet and Byte Structure Details

Host-Protocol Packet Structure:

FEND	TO/FROM	ID	Message	FEND

Transmitted Packet Structure:

Preamble	Start Symbol	# Bytes	TO/FROM	ID	Message	FCS

Host-Protocol ACK/NAK Structure:

FEND	TO/FROM	IDS	FEND

Transmitted ACK Structure:

Preamble	Start Symbol	69	TO/FROM	ID	FCS

TO/FROM Byte Detail:

TO Nibble	FROM Nibble

IDS Byte Detail:

ACK/NAK Bit	3 ID Bits	4 Retry # Bits

Figure 4.4

ACK and NAK packets contain an IDS byte which is detailed in Figure 4.4. The most significant bit in this byte is set to 1 for an ACK or 0 for a NAK. The next three bits are the packet ID, and the lower nibble of the byte holds the retry number for the ACK.

On power up the program is initialized by a call to the `setup` subroutine. The program then begins running in the `main` loop. The `tick` subroutine is called every 104.18 microseconds through `t_isr`, the interrupt service routine for timer T0. The `tick` subroutine always runs, and provides support for data reception, data transmission and event timing. The `tick` subroutine has a number of operating modes, controlled by the state of several flags.

Most of the time, `tick` will call `pll`, the receiver clock and data recovery subroutine. The `pll` subroutine uses two simple but effective signal processing techniques for accurately recovering bits from a data input steam with edge jitter and occasional noise spikes. The first signal processing technique is PLL clock alignment and the second technique is integrate-and-dump (I&D) bit estimation.

Register `R2` acts as a modulo 0 to 159 ramp counter that wraps on overflow about every 8 sampling ticks, (one bit period). This provides an 500 microsecond bit period, which equates to a nominal RF data rate of 2000 bits per second. Unless an edge occurs in the incoming bit stream, the ramp is incremented by 12.5% on each tick. If an edge occurs (change of logic state between ticks), the ramp is incremented 6.875% if the ramp value is below 80, or is incremented 18.125% if the ramp value is equal to or greater than 80. This causes the ramp period to gradually slide either backward or forward into alignment with the average bit period of the incoming data. After alignment, the position of the ramp can only change ±5.625% on each incoming data edge. Moderate edge jitter and occasional noise spikes will not seriously affect the ramp's alignment with the incoming data. Note that a preamble is needed to train the PLL (slide it into alignment).

Once the ramp is aligned, the I&D bit estimate becomes meaningful. The count in buffer `RXID` is incremented on each tick within a bit period if input sample `RXSMP` is a logic 1. At the end of the bit period (`R2` overflow wrap), the incoming bit is estimated to be a 0 if the count is four or less, or a 1 if the count is five or more. `RXID` is then cleared (dumped) in preparation for the next bit estimate. Integrate-and-dump estimation provides additional noise filtering by effectively averaging the value of the input samples within a bit period.

Once a bit value is determined, subroutine `pll` either inputs it into a 12-bit buffer (lower nibble of `RXBH` plus `RXBL`) used to detect the message start symbol, or adds it to buffer `RXBB`, which collects six-bit half symbols from the incoming encoded message. Flag `SOPFLG` controls which of these actions are taken.

You will notice that `tick` samples the RX input pin near the start of the subroutine, and when transmitting, outputs a TX bit sample as one of the first tasks. This helps minimize changes in the delay between timer T0 activating `t_isr` and these input/output events. If these activities are placed further down in the `tick` code or in the `pll` subroutine, an effect similar to adding extra noise-induced jitter can occur as different branches are taken through the code.

In addition to supporting data reception and transmission, the `tick` subroutine runs several timer functions. One timer provides a time-out for partial messages arriving from the host. The AutoSend timer and the transmit retry timer are also part of the `tick` subroutine.

The other interrupt service routine used by the protocol software is `s_isr`, which supports serial port interrupts by calling `srio`. The function of `srio` is to provide priority reception of messages from the host. An acknowledgment back to the host confirms the serial interrupt was enabled and the protocol received the host's message.

As mentioned, the code starts running in the `main` loop. A number of subroutines can be called from this loop, depending on the state of their associated control flags. Here are these subroutines and what they do:

The `do_as` subroutine automatically transmits a "Hello" test message paced by a timer in `tick`. This AutoSend function is activated by a call from `setup` if a jumper is detected across the pins near the "dot" end on the protocol board, as discussed above.

The `do_rt` subroutine retransmits a message if an ACK has not been received. Retransmissions are paced by a timer in `tick`. The timer is randomly loaded with one of eight different delays, which helps reduce the possibility of repeated collisions between two nodes trying to transmit a message at the same time. The protocol will attempt to transmit a message up to eight times. The `do_rt` subroutine manages attempts two through eight as needed.

The `aksnd` subroutine sends an ACK/NAK message back to the protocol's host to indicate the outcome of attempting to transmit a message. When called directly from the `main` subroutine, it sends a NAK message. When called from `do_rx`, it sends an ACK.

The `rxsop` subroutine detects the message start symbol (SOP) by comparing the bit pattern in the 12-bit correlation buffer updated by `pll` to the start symbol pattern. When the SOP pattern is detected, `rxsop` modifies flag states and clears buffers in preparation for receiving the encoded message. As mentioned, this protocol uses 12-bit encoding to achieve dynamic DC balance. The start symbol is not one of the 12-bit symbols used in the encoding table, but it is also DC balanced.

The `do_rx` subroutine receives and decodes the incoming message, tests the FCS for message accuracy, returns an ACK to the sender if it has received an error-free data message for this node, sends an ACK message to the host if it has received an ACK message for this node, and sends an error-free data message to the host if the message is for this node. These tasks are done by calling subroutines from `do_rx`. Here are these subroutines and what they do:

The `rxmsg` subroutine receives each six-bit half symbol from `pll` and converts it to a decoded nibble using the `smbl` table near the end of the listing. Decoded nibbles are assembled into bytes and added to the received message buffer. When all the message is received, control is returned to `do_rx`. If a message length overflow occurs, `rxmsg` fakes a short message that will fail the FCS test.

The `rxfcs` subroutine tests the message for errors by recalculating the FCS with the transmitted FCS bits included in the calculation. If there are no errors, the received FCS calculation will equal 0F0B8H. The `rxfcs` subroutine uses calls to `b_rfcs` and `a_rfcs` to do the FCS calculation and to test the results.

The `acktx` subroutine determines if the received message is an ACK for a packet (ID) being transmitted from this node. If so, `acktx` idles transmission attempts and signals `rxmsg` to send an ACK message to the host by setting flag states.

When called from `rxsmg`, `aksnd` sends an ACK message to the host. Notice that when `aksnd` is called from `main`, it sends a NAK message.

The `ackrx` subroutine transmits an ACK message back to the sending node when it receives a valid data message from the sending node addressed to it. The subroutines used by `ackrx` are "borrowed" from the transmit side of the protocol and will be discussed later.

The `rxsnd` subroutine sends a received data message to the host, provided the message is for its node and has passed the FCS test.

The `rxrst` subroutine resets flags and initializes buffers in preparation for receiving the next packet.

The first byte of a packet sent from the host triggers the serial interrupt service routine `t_isr` which calls subroutine `srio`. The serial interrupt is disabled and the `do_tx` subroutine is called. This subroutine takes in the message from the host, computes the FCS, turns the transmitter on, sends the preamble and start symbol, encodes and sends the message, and turns the transmitter off. The `do_tx` subroutine accomplishes these actions by calling other subroutines. Here are these transmit subroutines and what they do:

The `txget` subroutine receives the message from the host and loads it into the transmit message buffer. Provisions are made in `txget` to exit on a null message (just two FENDs), time-out on partial messages, or send the first part of an incoming message that is overflowing in length. Since the serial interrupt service routine is disabled from time-to-time, a short packet transfer acknowledgment message (PAC) is sent back to the host to confirm the protocol has the message and is attempting to transmit it. No PAC is sent on a null message or a time-out as there is nothing to send.

The txfcs subroutine calculates the FCS that will be used for error detection at the receive end. It uses calls to b_tfcs and a_tfcs to do the FCS calculation and to add the results to the message.

The txpre subroutine turns on the transmitter and after a short delay sends the preamble and start symbol using the data in the tstrt table near the end of the listing. Note that txpre is supported by tick to provide sample-by-sample bit transmission.

The txmsg subroutine encodes the message bytes as 12-bit symbols and transmits them in cooperation with tick. This subroutine uses the smbl table to encode each nibble in each message byte into six bits.

The txrst subroutine can either reset to send the same message again or can reset to receive a new message from the host, based on flag states.

The do_tx subroutine receives a message from the host and attempts to transmit it once. Additional transmit attempts are done by do_rt, which is called from main as needed. The do_rt subroutine uses most of the same subroutines as do_tx. The do_as subroutine can also be called from main to provide the AutoSend test transmission and it also uses many of the same subroutines as do_tx. And as mentioned earlier, ackrx uses several of these subroutines to transmit an ACK back for a received message.

4.2 Terminal Program Source

V110T30C.FRM is the Visual Basic source code for the companion terminal program to DK200A.ASM. After initializing flags, variables, etc., the form window is shown and the program starts making periodic calls to the Timer1_Timer "heartbeat" subroutine. The Xfer subroutine provides time-outs for PAC, ACK or NAK messages expected back from the protocol. Xfer is also handy for reminding you to turn on the power switch or put fresh batteries in the protocol board. The PC's serial input buffer is set up for polling (no interrupts) and is serviced by calling RxPtk from Timer1_Timer. The terminal program also has an AutoSend subroutine, ASPkt, that is called from Timer1_Timer when AutoSend is active. (No, you are not supposed to use the AutoSend feature in the protocol and the host program at the same time.) Here is a listing of the terminal program subroutines and what they do:

RxPkt is called from Timer1_Timer when bytes are found in the serial port input buffer. RxPkt calls two other subroutines, InCom and ShowPkt.

InCom collects bytes from the serial port input buffer for a period of time set by the InDel! variable. These bytes are added to the end of the RPkt$ string variable, which acts as byte FIFO.

`ShowPkt` is then called to display or otherwise process the bytes in `RPkt$`. The outer `Do, Loop Until (J = 0)` structure takes advantage of the framing characters to separate individual packets in `RPkt$`. This avoids the need for reading the PC's serial port input buffer at precise times which you probably can't do anyway. As each packet is removed from the left side of `RPkt$`, it is checked to see if it is a one-character PAC (0FFH character), a two-character ACK or NAK, or a data message of three or more characters. Flags `TFlag`, `ANFlag`, `NAFlag` and `TNFlag` are reset by `ShowPkt` as appropriate and are used by the `Xfer` monitoring subroutine to confirm messages are flowing back from the protocol in a timely manner. The `NAFlag` enables the next AutoSend transmission. The `ShwACK` flag selects either to display inbound messages (and PID Skips) only, or inbound messages plus PAC, ACK/NAK, TO/FROM and ID information.

`Text1_KeyPress` is used to build messages for transmission. Editing is limited to backspacing, and the message is sent by pressing the Enter key or entering the 240th character.

`SndPkt` breaks the message into packets, adds the framing characters, the TO/FROM address and the ID number to each packet and sends them out. `SndPkt` sets the `TFlag` and `ANFlag` flags and clears the value of several variables. NxtPkt is a small subroutine used by `SndPkt` that picks a new ID number for each packet.

`Xfer` monitors the elapsed time from when a packet is sent out (to the protocol) and a PAC is received back, and the elapsed time from when a packet is sent out and an ACK or NAK is received back. `Xfer` will display error messages and reset control flags and other variables through `ReSetTX` if these elapsed times get too long.

`ASPkt` automatically sends test packets using the `NxtPkt` and `SndPkt` subroutines. It is paced by the state of the `NAFlag`.

`GetPkt` is a small subroutine that supplies `ASPkt` with a message. Until the first message is typed in, `GetPkt` provides a default message. It otherwise provides the last message typed in.

`LenTrap` clears a text window when 32,000 bytes of text have accumulated in it.

The remaining subroutines in the terminal program are classical event procedures related to mouse clicks on the terminal program window. Most of these relate to the Menu bar.

The three top level choices on the Menu bar are *File*, *Edit* and *View*. Under *File* you can choose to *Exit* the terminal program. Under *Edit*, the next level of choices are the *To Address* and the *From Address*. Under the *To Address* you can choose *Nodes 1*, *2*, *3*, or *4*, with *Node 2* the default. Under the *From Address* you can choose *Nodes 1*, *2*, *3*, or *4*, again with *Node 2* the default.

Under *View* you can choose *Clear* (screen), *Show RX Dups, Show ACK/NAK*, and *AutoSend*, as discussed earlier. The status bar and its embedded progress bar at the bottom of the form monitors outbound packets even when *Show ACK/NAK* is not enabled.

4.3 Variations and Options

In most real world applications, `s_isr`, `srio`, `txget`, `rxsnd` and `aksnd` would be replaced with resident application subroutines. Your real-world application is left as a homework assignment. Test, test, test!

Another pair of programs are provided for your experimentation. DK110K.ASM is a simplified "shell" protocol that transmits a message received from the host (once) and sends any message received with a valid FCS to the host. PAC/ACK/NAK handshaking between the host and the protocol and between protocol nodes is not implemented. Also, no TO/FROM address filtering is provided at the protocol level. This gives you the flexibility to add these types of features either to the protocol or the terminal program yourself. Terminal Program V110T05B.FRM works with DK110K.ASM and provides a simple implementation of ACK/NAK handshaking at the host level. Of course, DK110K.ASM is not intended to work with V110T30C.FRM and DK200A.ASM is not intended to work with V110T05B.FRM.

4.4 Test Results

Laboratory tests show that a 916.5 MHz ASH radio system using the example software achieves a bit-error-rate between 10^{-4} and 10^{-3} at a received signal level of -101 dBm using pulse modulation (or -107 dBm using 100% amplitude modulation). Open-field range tests using commercial half-wave dipole antennas (Astron Antenna Model AXH9NSMS) demonstrate good performance chest-high at distances of one-eighth mile or more.

5 Source Code Listings

5.1 DK200A.ASM

```
;  DK200A.ASM 2002.07.31 @ 20:00 CST
;  See RFM Virtual Wire(r) Development Kit Warranty & License for terms of use
;  Experimental software - NO representation is
;  made that this software is suitable for any purpose
;  Copyright(c) 2000 - 2002, RF Monolithics, Inc.
;  AT89C2051 assembler source code file (TASM 3.01 assembler)
;  Low signal-to-noise protocol for RFM ASH transceiver
;  Integrate & dump PLL (I&D) - 62.40 us tick

          .NOLIST
          #INCLUDE "8051.H"         ;  tasm 8051 include file
          .LIST

;  constants:

ITMOD     .EQU      022H            ;  set timers 0 and 1 to mode 2
ITICK     .EQU      141             ;  set timer T0 for 62.40 us tick
ISMOD     .EQU      080H            ;  SMOD = 1 in PCON
IBAUD     .EQU      0FAH            ;  19.2 kbps @ 22.1184 MHz, SMOD = 1
ISCON     .EQU      050H            ;  UART mode 1

RMPT      .EQU      159             ;  PLL ramp top value (modulo 0 to 159)
RMPW      .EQU      159             ;  PLL ramp reset (wrap) value
RMPS      .EQU      80              ;  PLL ramp switch value
RMPI      .EQU      20              ;  PLL ramp increment value
RMPA      .EQU      29              ;  PLL 5.625% advance increment value (20 + 9)
RMPR      .EQU      11              ;  PLL 5.625% retard increment value (20 - 9)

AKMB      .EQU      03EH            ;  ACK message buffer start address
TXMB      .EQU      043H            ;  TX message buffer start address
TFTX      .EQU      044H            ;  TO/FROM TX message buffer address
IDTX      .EQU      045H            ;  packet ID TX message buffer address
RXMB      .EQU      061H            ;  RX message buffer start address
TFRX      .EQU      062H            ;  TO/FROM RX message buffer address
IDRX      .EQU      063H            ;  packet ID RX message buffer address
FEND      .EQU      0C0H            ;  FEND framing character (192)
SOPL      .EQU      08AH            ;  SOP low correlator pattern
SOPH      .EQU      0B3H            ;  SOP high correlator pattern
TXR0      .EQU      026H            ;  TX retry timer count

FCSS      .EQU      0FFH            ;  FCS seed
FCSH      .EQU      084H            ;  FCS high XOR mask
FCSL      .EQU      08H             ;  FCS low XOR mask
FCVH      .EQU      0F0H            ;  FCS valid high byte pattern
FCVL      .EQU      0B8H            ;  FCS valid low byte pattern

;  stack:  08H - 021H (26 bytes)

;  bit labels:

WBFLG     .EQU      010H            ;  warm boot flag (future use)
PLLON     .EQU      011H            ;  RX PLL control flag
RXISM     .EQU      012H            ;  RX inverted input sample
RXSMP     .EQU      013H            ;  RX input sample
LRXSM     .EQU      014H            ;  last RX input sample
RXBIT     .EQU      015H            ;  RX input bit
RXBFLG    .EQU      016H            ;  RX input bit flag
SOPFLG    .EQU      017H            ;  SOP detect flag
RXSFLG    .EQU      018H            ;  RX symbol flag
RM        .EQU      019H            ;  RX FCS message bit
OKFLG     .EQU      01AH            ;  RX FCS OK flag

SIFLG     .EQU      01BH            ;  serial in active flag
TSFLG     .EQU      01CH            ;  output TX sample flag
TXBIT     .EQU      01DH            ;  TX message bit
TM        .EQU      01EH            ;  TX FCS message bit
TXFLG     .EQU      01FH            ;  TX active flag
TMFLG     .EQU      020H            ;  TX message flag
TOFLG     .EQU      021H            ;  get message time out flag

AMFLG     .EQU      022H            ;  AutoSend message flag
ASFLG     .EQU      023H            ;  AutoSend active flag
ANFLG     .EQU      024H            ;  ACK/NAK status flag
```

```
SAFLG     .EQU      025H          ;  send ACK/NAK flag
NHFLG     .EQU      026H          ;  no RX FEND/header flag

SFLG1     .EQU      027H          ;  spare flag 1
SFLG2     .EQU      028H          ;  spare flag 2
SFLG3     .EQU      029H          ;  spare flag 3
SFLG4     .EQU      02AH          ;  spare flag 4
SFLG5     .EQU      02BH          ;  spare flag 5
SFLG6     .EQU      02CH          ;  spare flag 6
SFLG7     .EQU      02DH          ;  spare flag 7
SFLG8     .EQU      02EH          ;  spare flag 8
SFLG9     .EQU      02FH          ;  spare flag 9

;  register usage:

;  R0                                RX data pointer
;  R1                                TX data pointer
;  R2                                PLL ramp buffer
;  R3                                RX FCS buffer A
;  R4                                not used
;  R5                                TX FCS buffer A
;  R6                                TX FCS buffer B
;  R7                                RX FCS buffer B

;  byte labels:

BOOT      .EQU      022H          ;  1st byte of flags

RXID      .EQU      026H          ;  RX integrate & dump buffer
RXBL      .EQU      027H          ;  RX low buffer, SOP correlator, etc.
RXBH      .EQU      028H          ;  RX high buffer, SOP correlator, etc.
RXBB      .EQU      029H          ;  RX symbol decode byte buffer
RMDC      .EQU      02AH          ;  RX symbol decode loop counter
RMBIC     .EQU      02BH          ;  RX symbol decode index pointer
RMBYC     .EQU      02CH          ;  RX message byte counter
RMFCS     .EQU      02DH          ;  RX FCS byte buffer
RMSBC     .EQU      02EH          ;  RX symbol bit counter
RMLPC     .EQU      02FH          ;  RX message loop counter
RMFCC     .EQU      030H          ;  RX message FCS counter, etc.

TMFCC     .EQU      031H          ;  TX timer & loop counter
TXSMC     .EQU      032H          ;  TX output sample counter
TMBIC     .EQU      033H          ;  TX message bit counter
TMBYT     .EQU      034H          ;  TX message byte buffer
TMBYC     .EQU      035H          ;  TX message byte counter
TXSL      .EQU      036H          ;  TX message symbol low buffer
TXSH      .EQU      037H          ;  TX message symbol high buffer
TMFCS     .EQU      038H          ;  TX FCS byte buffer
TXTL      .EQU      039H          ;  TX timer low byte
TXTH      .EQU      03AH          ;  TX timer high byte
TXCNT     .EQU      03BH          ;  TX retry counter
IDBUF     .EQU      03CH          ;  packet ID buffer
TFBUF     .EQU      03DH          ;  TO/FROM address buffer

;  I/O pins:

MAX       .EQU      P1.6          ;  Maxim 218 power (on = 1)

RXPIN     .EQU      P3.2          ;  RX input pin (inverted data)
TXPIN     .EQU      P3.3          ;  TX output pin (on = 1)
PTT       .EQU      P1.7          ;  transmit enable (TX = 0)

PCRCV     .EQU      P3.7          ;  PC (host) input LED (on = 0)
RFRCV     .EQU      P3.5          ;  RX FCS OK LED (on = 0)
RXI       .EQU      P3.4          ;  RX activity LED (on = 0)

ID0       .EQU      P1.2          ;  jumper input bit 0 (dot end)
ID1       .EQU      P1.3          ;  jumper input bit 1
ID2       .EQU      P1.4          ;  jumper input bit 2
ID3       .EQU      P1.5          ;  jumper input bit 3

;  start of code:

          .ORG      00H           ;  hardware reset
          SETB      WBFLG         ;  set warm boot flag
reset:    AJMP      start         ;  jump to start

          .ORG      0BH           ;  timer 0 interrupt vector
t_isr:    ACALL     tick          ;  sampling tick subroutine
          RETI                    ;  interrupt done
```

```
         .ORG      023H            ;  serial interrupt vector
s_isr:   ACALL     srio            ;  serial I/O subroutine
         CLR       TI              ;  clear TI (byte sent) flag
         CLR       RI              ;  clear RI (byte received) flag
         RETI                      ;  interrupt done

         .ORG      040H            ;  above interrupt code space
start:   ACALL     setup           ;  initialization code

main:    JNB       AMFLG,mn0       ;  skip if AutoSend idle
         CLR       PCRCV           ;  else turn PCRCV LED on
         ACALL     do_as           ;  do AutoSend
         SETB      PCRCV           ;  turn PCRCV LED off
         AJMP      mn1             ;  and jump to RX SOP detect
mn0:     JNB       TMFLG,mn1       ;  skip if TX message idle
         CLR       PCRCV           ;  else turn PCRCV LED on
         ACALL     do_rt           ;  do TX retry
         SETB      PCRCV           ;  turn PCRCV LED off
mn1:     JNB       SAFLG,mn2       ;  skip if send ACK/NAK flag reset
         ACALL     aksnd           ;  else send NAK to host
mn2:     ACALL     rxsop           ;  do RX SOP detect
         JNB       SOPFLG,main     ;  if not SOP loop to main
         ACALL     do_rx           ;  else do RX message
mn_d:    AJMP      main            ;  and loop to main

do_rx:   CLR       ES              ;  deactivate serial interrupts
         ACALL     rxmsg           ;  decode RX message
         CLR       PLLON           ;  idle RX PLL
         ACALL     rxfcs           ;  test RX message FCS
         JNB       OKFLG,rx2       ;  reset if FCS error
         JNB       TXFLG,rx0       ;  skip if send TX idle
         ACALL     acktx           ;  if TX ACK, set send ACK flag
         JNB       SAFLG,rx0       ;  skip if send ACK/NAK flag reset
         ACALL     aksnd           ;  else send ACK message to host
         AJMP      rx2             ;  and jump to reset RX
rx0:     JB        ASFLG,rx1       ;  don't ACK AutoSend
         ACALL     ackrx           ;  ACK RX message
rx1:     ACALL     rxsnd           ;  send RX message to host
rx2:     ACALL     rxrst           ;  reset for next RX message
         SETB      PLLON           ;  enable RX PLL
         CLR       TI              ;  clear TI flag
         CLR       RI              ;  clear RI flag
         SETB      ES              ;  activate serial interrupts
rx_d:    RET                       ;  RX done

tick:    PUSH      PSW             ;  push status
         PUSH      ACC             ;  push accumulator
         MOV       C,RXPIN         ;  read RX input pin
         MOV       RXISM,C         ;  store as inverted RX sample
         JNB       TSFLG,tic0      ;  skip if TX sample out idle
         MOV       A,TXSMC         ;  else get sample count
         JZ        tic0            ;  skip if 0
         MOV       C,TXBIT         ;  else load TX bit
         MOV       TXPIN,C         ;  into TX output pin
         DEC       TXSMC           ;  decrement sample count
tic0:    JNB       PLLON,tic1      ;  skip if PLL idle
         ACALL     pll             ;  else run RX PLL
tic1:    JNB       TOFLG,tic2      ;  skip if get message timeout idle
         INC       TMFCC           ;  else bump timeout counter
         MOV       A,TMFCC         ;  get counter
         CJNE      A,#50,tic2      ;  skip if counter <> 50 (5.2 ms)
         CLR       TOFLG           ;  else reset time out flag
         MOV       TMFCC,#0        ;  reset counter
tic2:    INC       TXTL            ;  bump TX timer low
         MOV       A,TXTL          ;  load TX timer low
         JNZ       tick_d          ;  done if no rollover
         JNB       ASFLG,tic3      ;  skip if AutoSend idle
         DJNZ      TXTH,tick_d     ;  decrement TXTH, done if <> 0
         SETB      AMFLG           ;  else set AM message flag
         MOV       TXTL,#0         ;  clear TX delay low
         MOV       TXTH,#TXR0      ;  reload TX delay high
         AJMP      tick_d          ;  and jump to tic6
tic3:    JNB       TXFLG,tick_d    ;  skip if TX idle
         DJNZ      TXTH,tick_d     ;  decrement TXTH, done if <> 0
         SETB      TMFLG           ;  else set TM message flag
         MOV       DPTR,#delay     ;  point to delay table
         MOV       A,TL1           ;  get random table offset
         ANL       A,#07H          ;  mask out upper 5 bits
         MOVC      A,@A+DPTR       ;  load byte from table
         MOV       TXTH,A          ;  into TX delay high
         MOV       TXTL,#0         ;  clear TX delay low
```

```
        MOV     A,TXCNT         ; load retry count
        CJNE    A,#9,tick_d     ; if <> 9 jump to tick_d
        CLR     TMFLG           ; else reset send TX message
        CLR     ANFLG           ; reset ACK/NAK flag (NAK)
        SETB    SAFLG           ; set send ACK/NAK flag
        CLR     TXFLG           ; reset TX active flag
tick_d: POP     ACC             ; pop accumulator
        POP     PSW             ; pop status
        RET                     ; tick done

pll:    MOV     C,RXSMP         ; load RX sample
        MOV     LRXSM,C         ; into last RX sample
        MOV     C,RXISM         ; get inverted RX sample
        CPL     C               ; invert sample
        MOV     RXSMP,C         ; and store RX sample
        JNC     pll0            ; if <> 1 jump to pll0
        INC     RXID            ; else increment I&D
pll0:   JNB     LRXSM,pll1      ; if last sample 1
        CPL     C               ; invert current sample
pll1:   JNC     pll4            ; if no edge jump to pll4
        MOV     A,R2            ; else get PLL value
        CLR     C               ; clear borrow
        SUBB    A,#RMPS         ; subtract ramp switch value
        JC      pll3            ; if < 0 then retard PLL
pll2:   MOV     A,R2            ; else get PLL value
        ADD     A,#RMPA         ; add (RMPI + 5.625%)
        MOV     R2,A            ; store PLL value
        AJMP    pll5            ; and jump to pll5
pll3:   MOV     A,R2            ; get PLL value
        ADD     A,#RMPR         ; add (RMPI - 5.625%)
        MOV     R2,A            ; store PLL value
        AJMP    pll5            ; and jump to pll5
pll4:   MOV     A,R2            ; get PLL value
        ADD     A,#RMPI         ; add ramp increment
        MOV     R2,A            ; store new PLL value
pll5:   CLR     C               ; clear borrow
        MOV     A,R2            ; get PLL ramp value
        SUBB    A,#RMPT         ; subtract ramp top
        JC      pllD            ; if < 0 don't wrap
pll6:   MOV     A,R2            ; else get PLL value
        CLR     C               ; clear borrow
        SUBB    A,#RMPW         ; subtract reset value
        MOV     R2,A            ; and store result
        CLR     C               ; clear borrow
        MOV     A,RXID          ; get I&D buffer
        SUBB    A,#5            ; subtract 5
        JNC     pll7            ; if I&D count => 5 jump to pll7
        CLR     RXBIT           ; else RX bit = 0 for I&D count < 5
        SETB    RXBFLG          ; set new RX bit flag
        MOV     RXID,#0         ; clear the I&D buffer
        AJMP    pll8            ; and jump to pll8
pll7:   SETB    RXBIT           ; RX bit = 1 for I&D count => 5
        SETB    RXBFLG          ; set new RX bit flag
        MOV     RXID,#0         ; clear the I&D buffer
pll8:   JB      SOPFLG,pllA     ; skip after SOP detect
        MOV     A,RXBH          ; else get RXBH
        CLR     C               ; clear carry
        RRC     A               ; rotate right
        JNB     RXBIT,pll9      ; if bit = 0 jump to pll9
        SETB    ACC.7           ; else set 7th bit
pll9:   MOV     RXBH,A          ; store RXBH
        MOV     A,RXBL          ; get RXBL
        RRC     A               ; shift and pull in carry
        MOV     RXBL,A          ; store RXBL
        AJMP    pll_d           ; done for now
pllA:   MOV     A,RXBL          ; get RXBL
        CLR     C               ; clear carry
        RRC     A               ; shift right
        JNB     RXBIT,pllB      ; if bit = 0 jump to pllB
        SETB    ACC.5           ; else set 5th bit
pllB:   MOV     RXBL,A          ; store RXBL
        INC     RMSBC           ; bump bit counter
        MOV     A,RMSBC         ; get counter
        CJNE    A,#6,pllC       ; if <> 6 jump to pllC
        MOV     RXBB,RXBL       ; else get symbol
        MOV     RMSBC,#0        ; reset counter
        SETB    RXSFLG          ; set symbol flag
pllC:   AJMP    pll_d           ; done for now
pllD:   CLR     RXBFLG          ; clear RXBFLG
pll_d:  RET                     ; PLL done
```

```
rxsop:    JNB     RXBFLG,sop_d   ;  done if no RX bit flag
          CLR     RXBFLG         ;  else clear RX bit flag
          MOV     A,RXBL         ;  get low RX buffer
          CJNE    A,#SOPL,sop_d  ;  done if <> SOPL
          MOV     A,RXBH         ;  else get high RX buffer
          CJNE    A,#SOPH,sop_d  ;  done if <> SOPH
          CLR     A              ;  else clear A
          MOV     RXBL,A         ;  clear RX low buffer
          MOV     RXBH,A         ;  clear RX high buffer
          MOV     RMSBC,A        ;  clear RX symbol bit counter
          CLR     RXSFLG         ;  clear RX symbol flag
          SETB    SOPFLG         ;  set SOP detected flag
          CLR     RXI            ;  RXI LED on
sop_d:    RET                    ;  SOP detect done

rxmsg:    JNB     RXSFLG,rxmsg   ;  wait for RX symbol flag
          CLR     RXSFLG         ;  clear RX symbol flag
rxm1:     MOV     DPTR,#smbl     ;  point to RX symbol decode table
          MOV     RMDC,#16       ;  16 symbol decode table entries
          MOV     RMBIC,#0       ;  index into symbol table
rxm2:     MOV     A,RMBIC        ;  load index into A
          MOVC    A,@A+DPTR      ;  get table entry
          XRL     A,RXBB         ;  XOR to compare with RXBB
          JZ      rxm3           ;  exit loop with decoded nibble
          INC     RMBIC          ;  else bump index
          DJNZ    RMDC,rxm2      ;  and try to decode again
rxm3:     MOV     A,RMBIC        ;  get decoded nibble
          SWAP    A              ;  swap to high nibble
          MOV     RXBH,A         ;  into RXBH (low nibble is high)
rxm4:     JNB     RXSFLG,rxm4    ;  wait for symbol flag
          CLR     RXSFLG         ;  clear flag
rxm5:     MOV     DPTR,#smbl     ;  point to symbol decode table
          MOV     RMDC,#16       ;  16 symbol decode table entries
          MOV     RMBIC,#0       ;  reset symbol table index
rxm6:     MOV     A,RMBIC        ;  load index into A
          MOVC    A,@A+DPTR      ;  get table entry
          XRL     A,RXBB         ;  XOR to compare with RXBB
          JZ      rxm7           ;  exit loop with decoded nibble
          INC     RMBIC          ;  else bump index
          DJNZ    RMDC,rxm6      ;  and try to decode again
rxm7:     MOV     A,RMBIC        ;  get decoded nibble
          ORL     A,RXBH         ;  add RXBH low
          SWAP    A              ;  nibbles now in right order
          MOV     RXBH,A         ;  store in RXBH
          MOV     @R0,RXBH       ;  and store in RX message buffer
          CJNE    R0,#RXMB,rxm8  ;  skip if not 1st message byte
          MOV     A,RXBH         ;  else get 1st byte
          ANL     A,#63          ;  mask upper 2 bits
          MOV     RMBYC,A        ;  load message byte counter
          MOV     RMFCC,A        ;  and RX message loop counter
          CLR     C              ;  clear borrow
          SUBB    A,#30          ;  compare number of bytes to 30
          JC      rxm8           ;  skip if < 30
          MOV     RMBYC,#4       ;  else force byte counter to 4
          MOV     RMFCC,#4       ;  and force loop counter to 4
rxm8:     INC     R0             ;  bump pointer
          DJNZ    RMFCC,rxmsg    ;  if <> 0 get another byte
          MOV     R0,#RXMB       ;  reset RX message pointer
          SETB    RXI            ;  turn LED off
rxm_d:    RET                    ;  RX message done

rxfcs:    MOV     RMFCC,RMBYC    ;  move byte count to loop counter
rxf0:     MOV     RMFCS,@R0      ;  get next message byte
          INC     R0             ;  bump pointer
          ACALL   b_rfcs         ;  build FCS
          DJNZ    RMFCC,rxf0     ;  loop for next byte
          ACALL   a_rfcs         ;  test FCS
rxf_d:    RET                    ;  RX FCS done

acktx:    MOV     A,RXMB         ;  get 1st RX byte
          ANL     A,#64          ;  mask ACK bit
          CJNE    A,#64,atx_d    ;  done if <> ACK
          MOV     A,TFBUF        ;  else get TX TO/FROM
          SWAP    A              ;  swap for FROM/TO
          CJNE    A,TFRX,atx_d   ;  done if <> RX TO/FROM
          MOV     A,IDBUF        ;  else get TX packet ID
          CJNE    A,IDRX,atx_d   ;  done if <> TX ID
          SETB    ANFLG          ;  else set ACK/NAK flag (ACK)
          SETB    SAFLG          ;  set send ACK/NAK message flag
          CLR     TXFLG          ;  clear TX active flag
atx_d:    RET                    ;  ACK TX done
```

```
ackrx:    MOV       A,TFBUF          ;  get local TO/FROM address
          ANL       A,#15            ;  mask to get local FROM address
          MOV       B,A              ;  store FROM address
          MOV       A,TFRX           ;  get T/F address from RX buffer
          SWAP      A                ;  swap - FROM/TO
          ANL       A,#15            ;  mask to get TO address
          CJNE      A,B,arx0         ;  done if not to this node
          MOV       R1,#AKMB         ;  load ACK pointer
          MOV       @R1,#69          ;  ACK bit + 5 bytes
          MOV       TMFCS,#69        ;  load TX message FCS byte
          ACALL     b_tfcs           ;  and build FCS
          INC       R1               ;  bump pointer
          MOV       A,TFRX           ;  get TO/FROM byte
          SWAP      A                ;  swap TO/FROM addresses
          MOV       @R1,A            ;  add to ACK buffer
          MOV       TMFCS,A          ;  load TX message FCS byte
          ACALL     b_tfcs           ;  and build FCS
          INC       R1               ;  bump pointer
          MOV       A,IDRX           ;  get packet ID byte
          MOV       @R1,A            ;  add ID to ACK message
          MOV       TMFCS,A          ;  load TX message FCS byte
          ACALL     b_tfcs           ;  and build FCS
          INC       R1               ;  bump pointer
          ACALL     a_tfcs           ;  add FCS
          MOV       R1,#AKMB         ;  reset ACK pointer
          PUSH      TMBYC            ;  push TX message TMBYC
          MOV       TMBYC,#5         ;  5 bytes in ACK
          ACALL     txpre            ;  send TX preamble
          ACALL     txmsg            ;  send TX message
          CLR       A                ;  reset for next TX
          MOV       TMBYT,A          ;  clear TX message byte
          MOV       TXSMC,A          ;  clear TX out count
          MOV       TXSL,A           ;  clear TX symbol low
          MOV       TXSH,A           ;  clear TX symbol high
          MOV       R1,#TXMB         ;  point R1 to message start
          POP       TMBYC            ;  restore TX message TMBYC
arx0:     SETB      RFRCV            ;  turn FCS LED off
arx_d:    RET                        ;  RX ACK done (rxsnd sets ES)

rxsnd:    CLR       PCRCV            ;  turn PC LED on
          MOV       A,TFBUF          ;  get local TO/FROM address
          ANL       A,#15            ;  mask to get local FROM address
          MOV       B,A              ;  store FROM address
          MOV       A,TFRX           ;  get T/F address from RX buffer
          SWAP      A                ;  swap - FROM/TO
          ANL       A,#15            ;  mask to get TO address
          CJNE      A,B,rxs4         ;  if <> don't send to host
          DEC       RMBYC            ;  don't send
          DEC       RMBYC            ;  the 2 FCS bytes
          MOV       R0,#RXMB         ;  reset RX message pointer
          MOV       @R0,#FEND        ;  replace # bytes with 1st FEND
          JNB       NHFLG,rxs0       ;  skip if no FEND/header flag reset
          INC       R0               ;  bump past FEND
          DEC       RMBYC            ;  decrement byte count
          INC       R0               ;  bump past TO/FROM
          DEC       RMBYC            ;  decrement byte count
          INC       R0               ;  bump past ID
          DEC       RMBYC            ;  decrement byte count
rxs0:     CLR       TI               ;  clear TI flag
rxs1:     MOV       SBUF,@R0         ;  send byte
rxs2:     JNB       TI,rxs2          ;  wait until byte sent
          CLR       TI               ;  clear TI flag
          INC       R0               ;  bump pointer
          DJNZ      RMBYC,rxs1       ;  loop to echo message
          JB        NHFLG,rxs4       ;  skip if no FEND/header flag set
          MOV       SBUF,#FEND       ;  add 2nd FEND
rxs3:     JNB       TI,rxs3          ;  wait until byte sent
          CLR       TI               ;  clear TI flag
rxs4:     SETB      RFRCV            ;  turn FCS LED off
          SETB      PCRCV            ;  turn PC LED off
rxs_d:    RET                        ;  send RX message done

aksnd:    CLR       ES               ;  disable serial interrupts
          CLR       PCRCV            ;  turn PC LED on
          CLR       SAFLG            ;  reset send ACK/NAK flag
          CLR       TXFLG            ;  reset TX active flag
          MOV       A,IDBUF          ;  get local ID
          ANL       A,#7             ;  mask unused bits
          SWAP      A                ;  swap ID to upper IDS nibble
          ADD       A,TXCNT          ;  add retry count to IDS
          JNB       ANFLG,aks0       ;  skip if NAK
```

```
        ADD     A,#128          ; else set ACK bit
aks0:   MOV     B,A             ; hold IDS in B
        MOV     A,TFBUF         ; get local TO/FROM
        SWAP    A               ; switch TO and FROM
        CLR     TI              ; clear TI flag
        MOV     SBUF,#FEND      ; send 1st FEND
aks1:   JNB     TI,aks1         ; wait until byte sent
        CLR     TI              ; clear TI flag
        MOV     SBUF,A          ; send TO/FROM
aks2:   JNB     TI,aks2         ; wait until byte sent
        CLR     TI              ; clear TI flag
        MOV     SBUF,B          ; send IDS
aks3:   JNB     TI,aks3         ; wait until byte sent
        CLR     TI              ; clear TI flag
        MOV     SBUF,#FEND      ; send 2nd FEND
aks4:   JNB     TI,aks4         ; wait until byte sent
        ACALL   txrst           ; reset TX state
        SETB    RFRCV           ; turn FCS LED off
        SETB    PCRCV           ; turn PC LED off
        CLR     TI              ; clear TI flag
        CLR     RI              ; clear RI flag
        SETB    ES              ; enable serial interrupts
aks_d:  RET                     ; send ACK message done

rxrst:  CLR     A               ; clear A
        MOV     RXBH,A          ; clear buffer
        MOV     RXBL,A          ; clear buffer
        MOV     RXBB,A          ; clear buffer
        MOV     RMBYC,A         ; clear RX byte count
        MOV     RMFCC,A         ; clear loop counter
        MOV     R0,#RXMB        ; point R0 to message start
        CLR     OKFLG           ; clear FCS OK flag
        CLR     SOPFLG          ; enable SOP test
        SETB    RXI             ; turn RXI LED off
rxr_d:  RET                     ; RX reset done

b_rfcs: MOV     RMLPC,#8        ; load loop count of 8
brf0:   CLR     C               ; clear carry bit
        MOV     A,RMFCS         ; load RX message byte
        RRC     A               ; shift lsb into carry
        MOV     RMFCS,A         ; store shifted message byte
        MOV     RM,C            ; load RM with lsb
        CLR     C               ; clear carry bit
        MOV     A,R3            ; load high FCS byte
        RRC     A               ; shift right
        MOV     R3,A            ; store shifted high FCS
        MOV     A,R7            ; load low FCS byte
        RRC     A               ; shift and pull in bit for FCS high
        MOV     R7,A            ; store shifted low FCS
        JNB     RM,brf1         ; if lsb of low FCS = 0, jump to brf1
        CPL     C               ; else complement carry bit
brf1:   JNC     brf2            ; if RM XOR (low FCS lsb) = 0 jump to brf2
        MOV     A,R3            ; else load high FCS
        XRL     A,#FCSH         ; and XOR with high FCS poly
        MOV     R3,A            ; store high FCS
        MOV     A,R7            ; load low FCS
        XRL     A,#FCSL         ; XOR with low FCS poly
        MOV     R7,A            ; store low FCS
brf2:   DJNZ    RMLPC,brf0      ; loop through bits in message byte
brfcs_d: RET                    ; done this pass

a_rfcs: MOV     A,R3            ; load FCS high
        XRL     A,#FCVH         ; compare with 0F0H
        JNZ     arf0            ; if <> 0 jump to arf0
        MOV     A,R7            ; load FCS low
        XRL     A,#FCVL         ; else compare with 0B8H
        JNZ     arf0            ; if <> 0 jump to arf0
        CLR     RFRCV           ; else turn FCS LED on
        SETB    OKFLG           ; set FCS OK flag
arf0:   MOV     R3,#FCSS        ; reseed FCS high
        MOV     R7,#FCSS        ; reseed FCS low
arfcs_d: RET                    ; RX FCS done

srio:   PUSH    PSW             ; save
        PUSH    ACC             ; environment
        JNB     TI,sr_0         ; skip if not TI flag
        CLR     TI              ; else clear TI flag
sr_0:   JNB     RI,sr_1         ; skip if not RI flag
        CLR     RI              ; and clear RI flag
        JNB     SIFLG,sr_1      ; skip if serial in inactive
        CLR     PCRCV           ; else turn PC LED on
```

```
         ACALL    do_tx          ;  get & transmit message from host
         SETB     PCRCV          ;  turn PC LED off
sr_1:    POP      ACC            ;  restore
         POP      PSW            ;  environment
         RET                     ;  serial in done

do_as:   CLR      PLLON          ;  idle RX PLL
         ACALL    hello2         ;  get AutoSend message
         ACALL    txfcs          ;  build and add FCS
         ACALL    txpre          ;  send TX preamble
         ACALL    txmsg          ;  send TX message
         ACALL    txrst          ;  reset TX
         SETB     PLLON          ;  enable RX PLL
         RET                     ;  TX message done

do_tx:   ACALL    txget          ;  get TX message from host
         JNB      TXFLG,do1      ;  skip if send TX idle
         CLR      PLLON          ;  else idle RX PLL
         ACALL    txfcs          ;  build and add FCS
         ACALL    txpre          ;  send TX preamble
         ACALL    txmsg          ;  send TX message
         INC      TXCNT          ;  increment TX count
do1:     ACALL    txrst          ;  reset TX
         SETB     PLLON          ;  enable RX PLL
         RET                     ;  TX message done

do_rt:   CLR      PLLON          ;  idle RX PLL
         ACALL    txpre          ;  send TX preamble
         ACALL    txmsg          ;  send TX message
         INC      TXCNT          ;  increment TX count
         ACALL    txrst          ;  reset TX
         SETB     PLLON          ;  enable RX PLL
         RET                     ;  TX message done

txget:   MOV      A,SBUF         ;  get byte
         MOV      TMBYT,A        ;  copy to TMBYT
         XRL      A,#FEND        ;  compare to FEND
         JZ       txg0           ;  if FEND jump to txg0
         AJMP     txg_d          ;  else done
txg0:    MOV      @R1,TMBYT      ;  store 1st FEND
         INC      TMBYC          ;  bump TX byte counter
txg1:    MOV      TMFCC,#0       ;  reset timeout counter
         SETB     TOFLG          ;  set timeout flag
         CLR      RI             ;  clear RI flag
txg2:    JNB      TOFLG,txg3     ;  if TOFLG reset jump to txg3
         JNB      RI,txg2        ;  else loop until next byte
         CLR      RI             ;  clear RI flag
         CLR      TOFLG          ;  clear TOFLG
         AJMP     txg4           ;  and jump to txg4
txg3:    MOV      TMBYC,#2       ;  look like null message
         AJMP     txg6           ;  and jump to txg6
txg4:    MOV      A,SBUF         ;  get byte
         MOV      TMBYT,A        ;  copy to TMBYT
         INC      TMBYC          ;  bump byte counter
         INC      R1             ;  bump pointer R1
         MOV      @R1,TMBYT      ;  store byte
         MOV      A,TMBYC        ;  load counter
         CLR      C              ;  clear carry
         SUBB     A,#28          ;  test for 28 bytes
         JZ       txg5           ;  if 28 handle overflow at txg5
         MOV      A,TMBYT        ;  else load byte
         CJNE     A,#FEND,txg1   ;  if <> FEND loop to txg1
         AJMP     txg6           ;  else jump to txg6 on 2nd FEND
txg5:    MOV      @R1,#FEND      ;  force 2nd FEND
txg6:    MOV      R1,#TXMB       ;  reset TX message pointer
         MOV      A,TMBYC        ;  get byte count
         CJNE     A,#2,txg7      ;  if <> 2 jump to txg7
         MOV      TMBYC,#0       ;  else reset byte counter
         AJMP     txg_d          ;  jump to txg_d
txg7:    CLR      SIFLG          ;  idle serial in
         CLR      TOFLG          ;  clear timeout flag
         SETB     TXFLG          ;  set TX active flag
         MOV      TFBUF,TFTX     ;  update local TO/FROM buffer
         MOV      IDBUF,IDTX     ;  update local ID buffer
         CLR      TI             ;  clear TI flag
         MOV      SBUF,#FEND     ;  send 1st FEND
txg8:    JNB      TI,txg8        ;  wait until byte sent
         CLR      TI             ;  clear TI flag
         MOV      SBUF,#255      ;  send PAK byte
txg9:    JNB      TI,txg9        ;  wait until byte sent
         CLR      TI             ;  clear TI flag
```

```
        MOV     SBUF,#FEND      ;  send 2nd FEND
txgA:   JNB     TI,txgA         ;  wait until byte sent
        CLR     TI              ;  clear TI flag
txg_d:  RET                     ;  get TX message done

txfcs:  INC     TMBYC           ;  # bytes including FCS
        MOV     @R1,TMBYC       ;  replace 1st FEND with # bytes
        MOV     TMFCC,TMBYC     ;  move byte count to loop counter
        DEC     TMFCC           ;  loop count is 2 less
        DEC     TMFCC           ;  than # bytes including FCS
txf0:   MOV     TMFCS,@R1       ;  get next message byte
        INC     R1              ;  bump pointer
        ACALL   b_tfcs          ;  build FCS
        DJNZ    TMFCC,txf0      ;  loop for next byte
        ACALL   a_tfcs          ;  add FCS
        MOV     R1,#TXMB        ;  reset TX message pointer
        JB      ASFLG,txf1      ;  skip if AutoSend
        MOV     DPTR,#delay     ;  point to delay table
        MOV     A,TL1           ;  get random table offset
        ANL     A,#07H          ;  mask upper 5 bits
        MOVC    A,@A+DPTR       ;  load table byte
        MOV     TXTH,A          ;  into TX delay high
        AJMP    txf2            ;  skip AutoSend delay
txf1:   MOV     TXTH,#TXR0      ;  load AutoSend delay
txf2:   MOV     TXTL,#0         ;  clear TX delay low
        SETB    TMFLG           ;  set TX message flag
txf_d:  RET                     ;  TX FCS done

txpre:  CLR     PTT             ;  turn PTT on
        MOV     B,#200          ;  load PTT delay count
txp0:   DJNZ    B,txp0          ;  loop to delay
txp1:   MOV     DPTR,#tstrt     ;  point to TX start table
        MOV     B,#0            ;  clear B
        MOV     A,B             ;  B holds table offset
        MOVC    A,@A+DPTR       ;  load table entry
        MOV     TMBYT,A         ;  into TMBYT
        MOV     TMBIC,#4        ;  load bit count
        MOV     TXSMC,#0        ;  clear sample count
        SETB    TSFLG           ;  turn TX sample out on
txp2:   MOV     A,TXSMC         ;  get sample count
        JNZ     txp2            ;  loop until sample count 0
        MOV     A,TMBIC         ;  get bit count
        JNZ     txp3            ;  if <> 0 jump to txp3
        MOV     A,B             ;  else get current offset (0 to 11)
        CLR     C               ;  clear carry
        SUBB    A,#11           ;  subtract ending offset
        JZ      txp_d           ;  if 0  done
        INC     B               ;  else bump byte count
        MOV     A,B             ;  get count/offset
        MOVC    A,@A+DPTR       ;  load table entry
        MOV     TMBYT,A         ;  into TMBYT
        MOV     TMBIC,#4        ;  reload bit count
txp3:   MOV     A,TMBYT         ;  get TX message byte
        CLR     C               ;  clear carry
        RRC     A               ;  shift right into carry
        MOV     TXBIT,C         ;  load next bit
        MOV     TMBYT,A         ;  store shifted message byte
        DEC     TMBIC           ;  decrement bit count
        MOV     TXSMC,#8        ;  reload sample count
        AJMP    txp2            ;  loop again
txp_d:  RET                     ;  TX preamble done

txmsg:  MOV     B,#1            ;  count 1st byte sent
        MOV     A,@R1           ;  get 1st TX message byte
        MOV     TMBYT,A         ;  into TMBYT
        MOV     DPTR,#smbl      ;  point to symbol table
        ANL     A,#0FH          ;  clean offset
        MOVC    A,@A+DPTR       ;  get 6-bit symbol
        MOV     TXSL,A          ;  move to TXSL
        MOV     A,TMBYT         ;  get TMBYT
        SWAP    A               ;  swap nibbles
        ANL     A,#0FH          ;  clean offset
        MOVC    A,@A+DPTR       ;  get 6-bit symbol
        MOV     TXSH,A          ;  move to TXSH
        MOV     TMBIC,#12       ;  set bit count to 12
        MOV     TXSMC,#0        ;  clear sample count
txm0:   MOV     A,TXSMC         ;  get sample count
        JNZ     txm0            ;  loop until sample count 0
        MOV     A,TMBIC         ;  get bit count
        CLR     C               ;  clear carry
        SUBB    A,#7            ;  subtract 7
```

```
        JNC     txm1            ;  if => 7 jump to txm1
        MOV     A,TMBIC         ;  else get bit count
        JNZ     txm2            ;  if > 0 jump to txm2
        MOV     A,B             ;  else get current byte number
        CLR     C               ;  clear carry
        SUBB    A,TMBYC         ;  subtract TX message byte count
        JZ      txm3            ;  if 0  done
        INC     R1              ;  else bump byte pointer
        INC     B               ;  and bump byte counter
        MOV     A,@R1           ;  get next byte
        MOV     TMBYT,A         ;  into TMBYT
        MOV     DPTR,#smbl      ;  point to symbol table
        ANL     A,#0FH          ;  offset
        MOVC    A,@A+DPTR       ;  get 6-bit symbol
        MOV     TXSL,A          ;  move to TXSL
        MOV     A,TMBYT         ;  get TMBYT
        SWAP    A               ;  swap nibbles
        MOV     DPTR,#smbl      ;  point to symbol table
        ANL     A,#0FH          ;  clean offset
        MOVC    A,@A+DPTR       ;  get 6-bit symbol
        MOV     TXSH,A          ;  move to TXSH
        MOV     TMBIC,#12       ;  set bit count to 12
txm1:   MOV     A,TXSL          ;  get low TX symbol
        CLR     C               ;  clear carry
        RRC     A               ;  shift right into carry
        MOV     TXBIT,C         ;  load next bit
        MOV     TXSL,A          ;  store shifted message byte
        DEC     TMBIC           ;  decrement bit count
        MOV     TXSMC,#8        ;  reload sample count
        AJMP    txm0            ;  loop again
txm2:   MOV     A,TXSH          ;  get high TX symbol
        CLR     C               ;  clear carry
        RRC     A               ;  shift right into carry
        MOV     TXBIT,C         ;  load next bit
        MOV     TXSH,A          ;  store shifted message byte
        DEC     TMBIC           ;  decrement bit count
        MOV     TXSMC,#8        ;  reload sample count
        AJMP    txm0            ;  loop again
txm3:   CLR     TSFLG           ;  clear TX sample out flag
        CLR     TXPIN           ;  clear TX out pin
        SETB    PTT             ;  turn PTT off
txm_d:  RET                     ;  TX message done

txrst:  CLR     TMFLG           ;  clear TX message flag
        CLR     AMFLG           ;  clear AutoSend message flag
        CLR     A               ;  reset for next TX
        MOV     TMBYT,A         ;  clear TX message byte
        MOV     TMFCC,A         ;  clear TX FCS count
        MOV     TXSMC,A         ;  clear TX out count
        MOV     TXSL,A          ;  clear TX symbol low
        MOV     TXSH,A          ;  clear TX symbol high
        MOV     R1,#TXMB        ;  point R1 to message start
        JB      ASFLG,txr_d     ;  skip if in AutoSend
        JB      TXFLG,txr_d     ;  skip if send TX active
        MOV     TMBYC,A         ;  reset TX message byte count
        MOV     TXCNT,A         ;  reset TX retry count
        MOV     TXTL,A          ;  clear TX timer low
        MOV     TXTH,A          ;  clear TX timer high
        SETB    SIFLG           ;  enable serial in
txr_d:  RET                     ;  TX reset done

b_tfcs: MOV     B,#8            ;  load loop count of 8
btf0:   CLR     C               ;  clear carry bit
        MOV     A,TMFCS         ;  load TX message byte
        RRC     A               ;  shift lsb into carry
        MOV     TMFCS,A         ;  store shifted message byte
        MOV     TM,C            ;  load TM with lsb
        CLR     C               ;  clear carry bit
        MOV     A,R5            ;  load high FCS byte
        RRC     A               ;  shift right
        MOV     R5,A            ;  store shifted high FCS
        MOV     A,R6            ;  load low FCS byte
        RRC     A               ;  shift and pull in bit for FCS high
        MOV     R6,A            ;  store shifted low FCS
        JNB     TM,btf1         ;  if lsb of low FCS = 0, jump to btf1
        CPL     C               ;  else complement carry bit
btf1:   JNC     btf2            ;  if TM XOR (low FCS lsb) = 0 jump to btf2
        MOV     A,R5            ;  else load high FCS
        XRL     A,#FCSH         ;  and XOR with high FCS poly
        MOV     R5,A            ;  store high FCS
        MOV     A,R6            ;  load low FCS
```

```
          XRL       A,#FCSL         ;  XOR with low FCS poly
          MOV       R6,A            ;  store low FCS
btf2:     DJNZ      B,btf0          ;  loop through bits in message byte
btfcs_d:  RET                       ;  done this pass

a_tfcs:   MOV       A,R6            ;  load FCS (high/low switch)
          CPL       A               ;  1's complement
          MOV       @R1,A           ;  store at end of TX message
          INC       R1              ;  increment TX message byte pointer
          MOV       A,R5            ;  load FCS (high/low switch)
          CPL       A               ;  1's complement
          MOV       @R1,A           ;  store at end of TX message
          MOV       R5,#FCSS        ;  reseed FCS high
          MOV       R6,#FCSS        ;  reseed FCS low
atfcs_d:  RET                       ;  add TX FCS done

setup:    CLR       EA              ;  disable interrupts
          SETB      PTT             ;  turn PTT off
          CLR       TXPIN           ;  turn TX modulation off
tick_su:  MOV       TMOD,#ITMOD     ;  set timers T0 and T1 to mode 2
          CLR       TR0             ;  stop timer T0
          CLR       TF0             ;  clear T0 overflow
          MOV       TH0,#ITICK      ;  load count for 62.40 us tick
          MOV       TL0,#ITICK      ;  load count for 62.40 us tick
          SETB      TR0             ;  start timer T0
          SETB      ET0             ;  unmask T0 interrupt
uart_su:  SETB      MAX             ;  power up Maxim RS232 converter
          CLR       TR1             ;  stop timer T1
          CLR       TF1             ;  clear T1 overflow
          MOV       TH1,#IBAUD      ;  load baud rate count
          MOV       TL1,#IBAUD      ;  load baud rate count
          MOV       PCON,#ISMOD     ;  SMOD = 1 for baud rate @ 22.1184 MHz
          SETB      TR1             ;  start baud rate timer T1
          MOV       SCON,#ISCON     ;  enable UART mode 1
          MOV       A,SBUF          ;  clear out UART RX buffer
          CLR       A               ;  clear A
          CLR       RI              ;  clear RI (byte received) flag
          CLR       TI              ;  clear TI (byte sent) flag
          ACALL     hello           ;  send start up message
          ACALL     initr           ;  initialize TX & RX
          MOV       TXTH,#TXR0      ;  load default AutoSend delay
          SETB      SIFLG           ;  set serial in flag active
          MOV       C,ID3           ;  read ID3
          JC        as_set          ;  skip if no ID3 jumper
          SETB      NHFLG           ;  else set no FEND/header flag
as_set:   MOV       C,ID0           ;  read ID0
          JC        ser_on          ;  skip if no ID0 jumper
          ACALL     hello2          ;  else do AutoSend
ser_on:   SETB      ES              ;  enable serial ISR
isr_on:   SETB      EA              ;  enable interrupts
          SETB      PLLON           ;  activate RX PLL
setup_d:  RET                       ;  setup done

initr:    ANL       BOOT,#1         ;  warm boot (don't reset WBFLG)
          MOV       R0,#35          ;  starting here
          MOV       B,#93           ;  for 93 bytes
          CLR       A               ;  clear A
clr_r:    MOV       @R0,A           ;  clear RAM
          INC       R0              ;  bump RAM pointer
          DJNZ      B,clr_r         ;  loop again
          MOV       R0,#RXMB        ;  load RX buffer pointer
          MOV       R1,#TXMB        ;  load TX buffer pointer
          MOV       R2,A            ;  clear R2
          MOV       R3,#FCSS        ;  seed R3
          MOV       R5,#FCSS        ;  seed R5
          MOV       R6,#FCSS        ;  seed R6
          MOV       R7,#FCSS        ;  seed R7
          MOV       TFBUF,#34       ;  initialize TO/FROM 2 & 2
          MOV       IDBUF,#3        ;  initialize ID = 3
          CLR       SOPFLG          ;  clear SOPFLG
          SETB      PT0             ;  tick is 1st priority
ini_d:    RET                       ;  done

hello:    MOV       DPTR,#table     ;  point to table
          MOV       B,#13           ;  load loop count in B
          MOV       R7,#0           ;  R7 has 1st table entry
snd_h:    MOV       A,R7            ;  move table offset into A
          MOVC      A,@A+DPTR       ;  load table byte
          CLR       TI              ;  clear TI flag
          MOV       SBUF,A          ;  send byte
```

```
nxt_tx:   JNB       TI,nxt_tx      ;  wait until sent
          INC       R7             ;  bump index
          DJNZ      B,snd_h        ;  loop to send message
hello_d:  RET                      ;  done

hello2:   MOV       DPTR,#tbl_2    ;  point to table 2
          MOV       R1,#TXMB       ;  reset TX buffer pointer
          MOV       B,#10          ;  loop count for 9 bytes
          MOV       TMBYC,#0       ;  offset for 1st table entry
snd_h2:   MOV       A,TMBYC        ;  move table offset into A
          MOVC      A,@A+DPTR      ;  load table byte
          MOV       @R1,A          ;  into TX buffer
          INC       TMBYC          ;  increment TMBYC
          INC       R1             ;  increment R1
          DJNZ      B,snd_h2       ;  loop to load message
          MOV       R1,#TXMB       ;  reset TX pointer
          CLR       SIFLG          ;  reset serial input
          SETB      ASFLG          ;  set AutoSend flag
helo2_d   RET

;  tables:

tstrt:    .BYTE     10             ;  preamble/SOP table
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     8              ;  table data
          .BYTE     3              ;  table data
          .BYTE     11             ;  table data

smbl:     .BYTE     13             ;  4-to-6 bit table
          .BYTE     14             ;  table data
          .BYTE     19             ;  table data
          .BYTE     21             ;  table data
          .BYTE     22             ;  table data
          .BYTE     25             ;  table data
          .BYTE     26             ;  table data
          .BYTE     28             ;  table data
          .BYTE     35             ;  table data
          .BYTE     37             ;  table data
          .BYTE     38             ;  table data
          .BYTE     41             ;  table data
          .BYTE     42             ;  table data
          .BYTE     44             ;  table data
          .BYTE     50             ;  table data
          .BYTE     52             ;  table data
          .BYTE     00             ;  overflow

delay:    .BYTE     020H           ;  0.50 second
          .BYTE     044H           ;  1.10 second
          .BYTE     032H           ;  0.80 second
          .BYTE     058H           ;  1.40 second
          .BYTE     028H           ;  0.65 second
          .BYTE     04EH           ;  1.25 second
          .BYTE     03CH           ;  0.95 second
          .BYTE     062H           ;  1.55 second

table:    .BYTE     192            ;  start up message
          .BYTE     34             ;  table data
          .BYTE     3              ;  table data
          .BYTE     'D'            ;  table data
          .BYTE     'K'            ;  table data
          .BYTE     '2'            ;  table data
          .BYTE     '0'            ;  table data
          .BYTE     '0'            ;  table data
          .BYTE     'A'            ;  table data
          .BYTE     ':'            ;  table data
          .BYTE     ' '            ;  table data
          .BYTE     ' '            ;  table data
          .BYTE     192            ;  table data

tbl_2:    .BYTE     192            ;  table data
          .BYTE     34             ;  table data
          .BYTE     3              ;  table data
          .BYTE     'H'            ;  table data
          .BYTE     'e'            ;  table data
```

```
        .BYTE     'l'           ;  table data
        .BYTE     'l'           ;  table data
        .BYTE     'o'           ;  table data
        .BYTE     ' '           ;  table data
        .BYTE     192           ;  table data

        .END                    ;  end of source code
```

5.2 V110T30C.FRM

```
VERSION 5.00
Object = "{648A5603-2C6E-101B-82B6-000000000014}#1.1#0"; "MSCOMM32.OCX"
Object = "{F9043C88-F6F2-101A-A3C9-08002B2F49FB}#1.2#0"; "COMDLG32.OCX"
Object = "{831FDD16-0C5C-11D2-A9FC-0000F8754DA1}#2.0#0"; "MSCOMCTL.OCX"
Begin VB.Form Form1
   Caption         =   "V110T30C Terminal Program for DK200A Protocol - 2002.08.07 Rev"
   ClientHeight    =   5235
   ClientLeft      =   225
   ClientTop       =   630
   ClientWidth     =   7785
   LinkTopic       =   "Form1"
   MaxButton       =   0   'False
   ScaleHeight     =   5951.697
   ScaleMode       =   0  'User
   ScaleWidth      =   7905
   Begin MSComctlLib.ProgressBar ProgressBar1
      Height          =   251
      Left            =   1162
      TabIndex        =   3
      Top             =   4934
      Width           =   4875
      _ExtentX        =   8599
      _ExtentY        =   450
      _Version        =   393216
      Appearance      =   0
      Scrolling       =   1
   End
   Begin MSComctlLib.StatusBar StatusBar1
      Align           =   2  'Align Bottom
      Height          =   375
      Left            =   0
      TabIndex        =   2
      Top             =   4860
      Width           =   7785
      _ExtentX        =   13732
      _ExtentY        =   661
      _Version        =   393216
      BeginProperty Panels {8E3867A5-8586-11D1-B16A-00C0F0283628}
         NumPanels       =   4
         BeginProperty Panel1 {8E3867AB-8586-11D1-B16A-00C0F0283628}
            Alignment       =   1
            Bevel           =   0
            Object.Width    =   148
            MinWidth        =   148
         EndProperty
         BeginProperty Panel2 {8E3867AB-8586-11D1-B16A-00C0F0283628}
            Alignment       =   1
            Object.Width    =   1737
            MinWidth        =   1737
            Text            =   "TX Buffer"
            TextSave        =   "TX Buffer"
         EndProperty
         BeginProperty Panel3 {8E3867AB-8586-11D1-B16A-00C0F0283628}
            Object.Width    =   8755
            MinWidth        =   8755
         EndProperty
         BeginProperty Panel4 {8E3867AB-8586-11D1-B16A-00C0F0283628}
            Alignment       =   1
            Text            =   "Keyboard"
            TextSave        =   "Keyboard"
         EndProperty
      EndProperty
   End
   Begin MSComDlg.CommonDialog CommonDialog1
      Left            =   240
      Top             =   4320
      _ExtentX        =   688
      _ExtentY        =   688
```

```
    _Version        =   393216
 End
 Begin VB.TextBox Text2
    Height          =   2323
    Left            =   148
    Locked          =   -1  'True
    MultiLine       =   -1  'True
    ScrollBars      =   2  'Vertical
    TabIndex        =   1
    Top             =   0
    Width           =   7460
 End
 Begin VB.Timer Timer1
    Left            =   720
    Top             =   4320
 End
 Begin MSCommLib.MSComm MSComm1
    Left            =   1200
    Top             =   4320
    _ExtentX        =   794
    _ExtentY        =   794
    _Version        =   393216
    DTREnable       =   -1  'True
 End
 Begin VB.TextBox Text1
    Height          =   2323
    Left            =   120
    MultiLine       =   -1  'True
    ScrollBars      =   2  'Vertical
    TabIndex        =   0
    Top             =   2513
    Width           =   7460
 End
 Begin VB.Menu mnuFile
    Caption         =   "&File"
    Begin VB.Menu mnuExit
       Caption         =   "E&xit"
    End
 End
 Begin VB.Menu mnuEdit
    Caption         =   "&Edit"
    Begin VB.Menu mnuToAdr
       Caption         =   "To Address"
       Begin VB.Menu mnuTN1
          Caption         =   "Node 1"
       End
       Begin VB.Menu mnuTN2
          Caption         =   "Node 2"
          Checked         =   -1  'True
       End
       Begin VB.Menu mnuTN3
          Caption         =   "Node 3"
       End
       Begin VB.Menu mnuTN4
          Caption         =   "Node 4"
       End
    End
    Begin VB.Menu mnuFrmAdr
       Caption         =   "From Address"
       Begin VB.Menu mnuFN1
          Caption         =   "Node 1"
       End
       Begin VB.Menu mnuFN2
          Caption         =   "Node 2"
          Checked         =   -1  'True
       End
       Begin VB.Menu mnuFN3
          Caption         =   "Node 3"
       End
       Begin VB.Menu mnuFN4
          Caption         =   "Node 4"
       End
    End
 End
 Begin VB.Menu mnuView
    Caption         =   "&View"
    Begin VB.Menu mnuClear
       Caption         =   "&Clear"
    End
```

```
      Begin VB.Menu mnuDups
         Caption         =   "Show RX &Dups"
         Checked         =   -1  'True
      End
      Begin VB.Menu mnuShw
         Caption         =   "&Show ACK/NAK"
         Checked         =   -1  'True
      End
      Begin VB.Menu mnuAutoSnd
         Caption         =   "&AutoSend"
      End
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False

' V110T30C.FRM, 2002.08.07 @ 08:00 CDT
' See RFM Virtual Wire(r) Development Kit Warranty & License for terms of use
' Tutorial software - NO representation is made that this software
' is suitable for any purpose
' Copyright(c) 2000-2002, RF Monolithics, Inc.
' For experimental use with the RFM DR1200A-DK and DR1201A-DK
' and DR1300A-DK ASH Transceiver Virtual Wire(R) Development Kits
' For protocol software version DK200A.ASM
' Check www.rfm.com for latest software updates
' Compiled in Microsoft Visual Basic 6.0

' global variables:
  Dim ComData$                                            ' com input string
  Dim ComTime!                                            ' com input reference time
  Dim KeyIn$                                              ' keystroke input buffer
  Dim TXFlag As Integer                                   ' send TX message flag
  Dim TNFlag As Integer                                   ' send next TX packet flag
  Dim TPkt$                                               ' keyboard input string
  Dim TSPkt$                                              ' SLIP encoded input string
  Dim TXPkt$                                              ' transmit message string
  Dim SPkt$                                               ' transmit packet string
  Dim TFlag As Integer                                    ' packet transfer flag
  Dim ANFlag As Integer                                   ' ACK/NAK flag
  Dim TCnt As Integer                                     ' TX timeout counter
  Dim XCnt As Integer                                     ' TX transfer retry counter
  Dim Temp$                                               ' temp string buffer
  Dim Temp1$                                              ' temp1 string buffer
  Dim FRM  As Integer                                     ' RX From address
  Dim ID As Integer                                       ' RX packet ID
  Dim DupFltr As Integer                                  ' duplicate RX filter flag
  Dim PID(15) As Integer                                  ' packet ID array (dup/skip detector)
  Dim DpSkp As Integer                                    ' dup/skip status
  Dim pSLIP As Integer                                    ' SLIP pointer
  Dim G As Integer                                        ' ID compare
  Dim I As Integer                                        ' general purpose index/counter
  Dim K As Integer                                        ' SLIP encoded packet length
  Dim N As Integer                                        ' keyboard byte counter
  Dim P As Integer                                        ' TX packet ID #, 1 - 7
  Dim FEND$                                               ' SLIP framing character
  Dim ESC$                                                ' SLIP escape character
  Dim TFEND$                                              ' SLIP transpose frame
  Dim TESC$                                               ' SLIP transpose escape
  Dim PktHdr$                                             ' packet header
  Dim J As Integer                                        ' FEND$ string position
  Dim Q As Integer                                        ' RPkt$ length
  Dim RPkt$                                               ' RX message FIFO string
  Dim R2Pkt$                                              ' RX message display string
  Dim ASFlag As Integer                                   ' AutoSend enable flag
  Dim NAFlag As Integer                                   ' AutoSend next message flag
  Dim InDel!                                              ' delay for com input
  Dim PCnt As Integer                                     ' packet TX tries counter
  Dim ShwACK As Integer                                   ' show ACK/NAK flag
  Dim TNode As Integer                                    ' To node numeric value
  Dim FNode As Integer                                    ' From node numeric value
  Dim TF As Integer                                       ' To/From node numeric value
  Dim ASStr$                                              ' AutoSend string

Private Sub Form_Load()

' initialize variables:
  ComData$ = ""                                           ' clear string
  ComTime! = 0                                            ' clear reference time
```

```
  KeyIn$ = ""                                                 ' clear keystroke buffer
  TXFlag = 0                                                  ' clear TX message flag
  TNFlag = 0                                                  ' clear next TX packet flag
  TPkt$ = ""                                                  ' clear TX packet string
  TSPkt$ = ""                                                 ' clear SLIP encoded string
  TXPkt$ = ""                                                 ' clear TX message string
  SPkt$ = ""                                                  ' clear send packet string
  TFlag = 0                                                   ' clear transfer flag
  ANFlag = 0                                                  ' clear ACK/NAK flag
  TCnt = 0                                                    ' clear TX timeout counter
  XCnt = 0                                                    ' clear transfer counter
  Temp$ = ""                                                  ' clear temp string buffer
  Temp1$ = ""                                                 ' clear 2nd temp string buffer
  FRM = 0                                                     ' set RX From to 0
  ID = 0                                                      ' set RX packet ID to 0
  DupFltr = 0                                                 ' clear duplicate filter
  pSLIP = 0                                                   ' clear SLIP pointer
  G = 0                                                       ' clear ID compare
  I = 0                                                       ' clear index/counter
  K = 0                                                       ' clear SLIP packet length
  N = 0                                                       ' clear keyboard byte counter
  P = 3                                                       ' set packet ID to 3
  FEND$ = Chr$(192)                                           ' initialize SLIP framing character
  ESC$ = Chr$(219)                                            ' initialize SLIP escape character
  TFEND$ = Chr$(220)                                          ' initialize SLIP transpose frame
  TESC$ = Chr$(221)                                           ' initialize SLIP transpose escape
  PktHdr$ = Chr$(34)                                          ' set To/From default = 2/2
  J = 0                                                       ' clear string position
  Q = 0                                                       ' clear string length
  RPkt$ = ""                                                  ' clear RX FIFO string
  R2Pkt$ = ""                                                 ' clear RX display string
  ASFlag = 0                                                  ' clear AutoSend flag
  NAFlag = 0                                                  ' clear next AutoSend flag
  PCnt = 0                                                    ' clear TX tries counter
  ShwACK = 1                                                  ' set show ACK/NAK flag
  TNode = 2                                                   ' set To node default = 2
  FNode = 2                                                   ' set From node default = 2
  TF = 34                                                     ' set TF default = 34
  For B = 0 To 15
    PID(B) = -1                                               ' set PID array elements = -1
  Next B

  ASStr$ = "**Auto Test Message**" & vbCrLf                   ' default AutoSend message

  Form1.Left = (Screen.Width - Form1.Width) / 2               ' center form left-right
  Form1.Top = (Screen.Height - Form1.Height) / 2              ' center form top-bottom
  Text1.BackColor = QBColor(0)                                ' black background
  Text1.ForeColor = QBColor(15)                               ' white letters
  Text1.FontSize = 10                                         ' 10 point font
  Text2.BackColor = QBColor(0)                                ' black background
  Text2.ForeColor = QBColor(15)                               ' white letters
  Text2.FontSize = 10                                         ' 10 point font

  MSComm1.CommPort = 1                                        ' initialize com port
  MSComm1.Settings = "19200,N,8,1"                            ' at 19.2 kbps
  MSComm1.RThreshold = 0                                      ' poll only, no interrupts
  MSComm1.InputLen = 0                                        ' read all bytes
  MSComm1.PortOpen = True                                     ' open com port
  InDel! = 0.1                                                ' initialize get com delay at 100 ms

  StatusBar1.Panels(4).Text = "Keyboard Active"               ' keyboard active status message
  ProgressBar1.Min = 0                                        ' progress bar min number of TX bytes
  ProgressBar1.Max = 240                                      ' progress bar max number of TX bytes

  Show                                                        ' show form
  Text1.Text = "**TX Message Window**" & vbCrLf               ' 1st line of TX start up message
  Text1.Text = Text1.Text & "**Set for Node 2 & 2**" _
  & vbCrLf & vbCrLf                                           ' 2nd line of TX start up message
  Text1.SelStart = Len(Text1.Text)                            ' put cursor at end of text
  Text2.Text = "**RX Message Window**" & vbCrLf               ' RX start up message
  Text2.SelStart = Len(Text2.Text)                            ' put cursor at end of text

  Randomize                                                   ' initialize random # generator

  Timer1.Interval = 300                                       ' 300 ms timer interval
  Timer1.Enabled = True                                       ' start timer

End Sub
```

```
Private Sub Timer1_Timer()
  If ANFlag = 1 Then                                    ' if ACK/NAK flag set
    Call Xfer                                           ' call Xfer (detect switch OFF, etc.)
  End If
  If MSComm1.InBufferCount > 0 Then                     ' if com input buffer has bytes
    Call RxPkt                                          ' call RxPkt
  End If
  If TXFlag = 1 Then                                    ' if TX message flag set
    If TNFlag = 1 Then                                  ' and next TX packet flag set
      Call SndPkt                                       ' call SndPkt
    End If
  End If
  If ASFlag = 1 Then                                    ' if AutoSend flag set
    If TXFlag = 0 Then                                  ' and TX message flag clear
      Call ASPkt                                        ' call AutoSend
    End If
  End If
End Sub

Public Sub RxPkt()
  Call InCom                                            ' InCom gets RX message bytes
  Call ShowPkt                                          ' ShowPkt shows RX message bytes
End Sub

Public Sub InCom()
  On Error Resume Next                                  ' set up error handler
  ComTime! = Timer                                      ' get current time
  Do Until Abs(Timer - ComTime!) > InDel!               ' get bytes for InDel! interval
    Do While MSComm1.InBufferCount > 0                  ' while bytes are in com buffer
     ComData$ = ComData$ & MSComm1.Input                ' put them in ComData$
    Loop
  Loop
End Sub

Public Sub ShowPkt()
  RPkt$ = RPkt$ & ComData$                              ' add ComData$ bytes to RPkt$ FIFO
  ComData$ = ""                                         ' and clear ComData$
  Do                                                    ' do until FEND$s are gone
    Q = Len(RPkt$)                                      ' Q is RPkt$ packet length
    J = InStr(1, RPkt$, FEND$)                          ' find position of next FEND$
    If (J < 2) Then                                     ' if FEND$ is in the first position
      RPkt$ = Right$(RPkt$, (Q - J))                    ' just delete it
    Else                                                ' else
      R2Pkt$ = Left$(RPkt$, (J - 1))                    ' R2Pkt$ what's left of this FEND$
      RPkt$ = Right$(RPkt$, (Q - J))                    ' RPkt$ what's right of this FEND$
      If Len(R2Pkt$) = 1 Then                           ' only PAC is a 1 byte message
        If (R2Pkt$ = Chr$(255)) Then                    ' if PAC byte
          TFlag = 0                                     ' reset transfer flag
          If ShwACK = 1 Then                            ' if show ACK/NAK flag set
            Call LenTrap                                ' manage textbox memory
            Text1.SelStart = Len(Text1.Text)            ' put cursor at end of text
            Text1.SelText = "<Xfer on try " _
            & Str(XCnt + 1) & "> "                      ' show try number for transfer
          End If
          R2Pkt$ = ""                                   ' and clear R2Pkt$
        End If
      ElseIf Len(R2Pkt$) = 2 Then                       ' only ACK/NAK are 2 byte messages
        ANFlag = 0                                      ' reset ACK/NAK flag
        NAFlag = 0                                      ' reset next AutoSend flag
        TNFlag = 1                                      ' set next TX packet flag
        Temp$ = Str((Asc(Left$(R2Pkt$, 1)) And &HF))    ' get From address
        Temp1$ = Str((Int(Asc(Mid$(R2Pkt$, 2, 1)) / 16)) _
        And &H7)                                        ' get packet ID number
        If (Asc(Right$(R2Pkt$, 1)) And &H80) = 128 Then ' if ACK bit set
          PCnt = (Asc(Right$(R2Pkt$, 1)) And &HF)       ' get ACK retry number
          If ShwACK = 1 Then                            ' if show ACK/NAK flag set
            Call LenTrap                                ' manage textbox memory
            Text1.SelStart = Len(Text1.Text)            ' put cursor at end of text
            Text1.SelText = "<ACK from N" _
            & Temp$ & " : P" & Temp1$ & " on " _
            & Str(PCnt) & ">" & vbCrLf                  ' show ACK From, ID and retry number
          End If
          R2Pkt$ = ""                                   ' and clear R2Pkt$
        Else
          If ShwACK = 1 Then                            ' if show ACK/NAK flag set
            Call LenTrap                                ' manage textbox memory
            Text1.SelStart = Len(Text1.Text)            ' put cursor to end of text
            Text1.SelText = "<NAK from N" _
            & Temp$ & " : P" & Temp1$ & ">" & vbCrLf    ' show NAK received
          End If
```

```
        R2Pkt$ = ""                                   ' and clear R2Pkt$
      End If
    ElseIf Len(R2Pkt$) > 2 Then                       ' other messages are > 2 bytes
      Do                                              ' decode FEND$ escape sequences
        pSLIP = InStr(R2Pkt$, (ESC$ & TFEND$))        ' find position of next ESC$ & TFEND$
        If pSLIP <> 0 Then                            ' if (ESC$ & TFEND$) present
          K = Len(R2Pkt$)
          If K >= (pSLIP + 2) Then                    ' if escape sequence not last bytes
            R2Pkt$ = Left$(R2Pkt$, (pSLIP - 1)) & FEND$ _
            & Mid$(R2Pkt$, (pSLIP + 2))               ' replace escape sequence with FEND$
          Else
            R2Pkt$ = Left$(R2Pkt$, (pSLIP - 1)) & FEND$   ' else replace with FEND$ at end
          End If
        Else
          Exit Do                                     ' else done
        End If
      Loop
      Do                                              ' decode ESC$ escape sequences
        pSLIP = InStr(R2Pkt$, (ESC$ & TESC$))         ' find position of next ESC$ & TESC$
        If pSLIP <> 0 Then                            ' if (ESC$ & TESC$) string(s) present
          I = Len(R2Pkt$)
          If I >= (pSLIP + 2) Then                    ' if escape sequence not last bytes
            R2Pkt$ = Left$(R2Pkt$, (pSLIP - 1)) & ESC$ _
            & Mid$(R2Pkt$, (pSLIP + 2))               ' replace escape sequence with ESC$
          Else
            R2Pkt$ = Left$(R2Pkt$, (pSLIP - 1)) & ESC$    ' else replace with ESC$ at end
          End If
        Else
          Exit Do                                     ' else done
        End If
      Loop
      FRM = Asc(Left$(R2Pkt$, 1)) And &HF             ' get RX packet From address
      ID = Asc(Mid$(R2Pkt$, 2, 1)) And &H7            ' get RX packet ID
      Call ChkPkt                                     ' check packet for skip/dup
      If DpSkp <> 0 Or DupFltr = 0 Then               ' if not dup or dup filter off
        If ShwACK = 1 Then                            ' if show ACK/NAK flag set
          Temp$ = Str(FRM)                            ' make From address string
          Temp1$ = Str(ID)                            ' make packet ID string
          R2Pkt$ = Right$(R2Pkt$, (Len(R2Pkt$) - 2))  ' strip off TO/FROM and ID bytes
          If Right$(R2Pkt$, 2) = vbCrLf Then          ' check for vbCrLf
            R2Pkt$ = Left$(R2Pkt$, (Len(R2Pkt$) - 2)) ' remove vbCrLf if present
          ElseIf Right$(R2Pkt$, 1) = Chr$(13) Then    ' also check for a trailing Cr
            R2Pkt$ = Left$(R2Pkt$, (Len(R2Pkt$) - 1)) ' remove Cr if present
          End If
          If Left$(R2Pkt$, 1) = Chr$(10) Then         ' check for a leading Lf
            R2Pkt$ = Right$(R2Pkt$, (Len(R2Pkt$) - 1))    ' remove Lf if present
          End If
          Call LenTrap                                ' manage textbox memory
          If DpSkp = 1 Then                           ' if skipped packet(s) detected
            Text2.SelStart = Len(Text2.Text)          ' put cursor at end of text
            Text2.SelText = " [PID Skip] "            ' show where skip(s) occurred
          End If
          Text2.SelStart = Len(Text2.Text)            ' put cursor at end of text
          Text2.SelText = R2Pkt$ & "  <from N" _
          & Temp$ & " : P" & Temp1$ & ">" & vbCrLf    ' show message, From, ID, new line
          R2Pkt$ = ""                                 ' and clear R2Pkt$
        Else
          R2Pkt$ = Right$(R2Pkt$, (Len(R2Pkt$) - 2))  ' else strip off TO/FROM and ID bytes
          Call LenTrap                                ' manage textbox memory
          If DpSkp = 1 Then                           ' if skipped packet(s) detected
            Text2.SelStart = Len(Text2.Text)          ' put cursor at end of text
            Text2.SelText = " [PID Skip] "            ' show where skip(s) occurred
          End If
          Text2.SelStart = Len(Text2.Text)            ' put cursor at end of text
          Text2.SelText = R2Pkt$                      ' show message
          R2Pkt$ = ""                                 ' and clear R2Pkt$
        End If
      End If
    End If
  End If
Loop Until (J = 0)                                    ' done when there are no more FEND$s
End Sub

Public Sub ChkPkt()
  G = PID(FRM)                                        ' G is last stored ID
  If G = -1 Then                                      ' if -1 it's the first check
    DpSkp = -1                                        ' so signal no skip/dup
  ElseIf G = ID Then                                  ' else if G = ID it's a dup
    DpSkp = 0                                         ' signal dup
  Else                                                ' else if G <> to ID
    G = G + 1                                         ' increment G
```

```
    If G > 7 Then                                      ` if greater than 7
      G = 0                                            ` reset to 0
    End If
    If G = ID Then                                     ` if updated G = ID
      DpSkp = -1                                       ` signal no skip/dup
    Else
      DpSkp = 1                                        ` else signal skip
    End If
  End If
  PID(FRM) = ID                                        ` store current PID for next check
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
  If TXFlag = 0 Then                                   ` if TX message flag reset
    KeyIn$ = Chr$(KeyAscii)                            ` convert keystroke to character
    If KeyIn$ = Chr$(8) Then                           ` if it is a backspace from keyboard
      If N > 0 Then                                    ` and if keyboard byte counter > 0
        TPkt$ = Left$(TPkt$, (N - 1))                  ` trim right end of packet
        N = N - 1                                      ` back up byte counter
      End If
    ElseIf KeyIn$ = Chr$(13) Then                      ` else if it is a Cr
      TPkt$ = TPkt$ & vbCrLf                           ` add vbCrLf to TX packet
      ASStr$ = TPkt$                                   ` update AutoSend string
      N = 0                                            ` reset keyboard byte counter
      TXFlag = 1                                       ` set TX message flag
      TNFlag = 1                                       ` set next TX packet flag
      StatusBar1.Panels(4).Text = “Keyboard Locked”    ` show keyboard locked
    Else
      TPkt$ = TPkt$ & KeyIn$                           ` else add byte to TX packet
      N = N + 1                                        ` increment byte counter
    End If
    If (N = 238) Then                                  ` if keyboard byte counter is 238
      TPkt$ = TPkt$ & vbCrLf                           ` add vbCrLf to TX message
      ASStr$ = TPkt$                                   ` update AutoSend string
      Text1.SelStart = Len(Text1.Text)                 ` place cursor at end
      Text1.SelText = KeyIn$ & vbCrLf                  ` show key input and vbCrLf
      KeyAscii = 0                                     ` block double key display
      N = 0                                            ` reset keyboard byte counter
      TXFlag = 1                                       ` set TX message flag
      TNFlag = 1                                       ` set next TX packet flag
      StatusBar1.Panels(4).Text = “Keyboard Locked”    ` show keyboard locked
    End If
    Call LenTrap                                       ` manage textbox memory
  Else
    KeyAscii = 0                                       ` block keystroke if TX flag set
  End If
End Sub

Public Sub SndPkt()
  If TNFlag = 1 Then                                   ` if next TX packet flag set
    If TPkt$ <> “” Then                                ` if TPkt$ has new bytes
      L = Len(TPkt$)                                   ` get number of bytes in TPkt
      For I = 1 To L                                   ` for each byte in TPkt$
        Temp$ = Mid$(TPkt$, I, 1)                      ` load byte in Temp$
        If Temp$ = FEND$ Then                          ` if byte in Temp$ is a FEND$
          TSPkt$ = TSPkt$ & ESC$ & TFEND$              ` add ESC$ & TFEND$ to TSPkt$
        ElseIf Temp$ = ESC$ Then                       ` else if byte is an ESC$
          TSPkt$ = TSPkt$ & ESC$ & TESC$               ` add ESC$ & TESC$ to TSPkt$
        Else
          TSPkt$ = TSPkt$ & Temp$                      ` else just add Temp$ byte to TSPkt$
        End If
      Next I
      TXPkt$ = TXPkt$ & TSPkt$                         ` add new message to TX FIFO
      TPkt$ = “”                                       ` clear new message string
      TSPkt$ = “”                                      ` clear SLIP encoded string
    End If
    If Int(4 * Rnd) > 0 Then                           ` skip 25% to allow other traffic
      TNFlag = 0                                       ` clear next TX packet flag
      L = Len(TXPkt$)                                  ` get number of bytes in TXPkt$
      If L <= 240 Then                                 ` if less than 240 bytes
        ProgressBar1.Value = L                         ` show number on TX progress bar
      Else
        ProgressBar1.Value = 240                       ` else cap TX progress bar at 240
      End If
      If L > 0 Then                                    ` if TXPkt$ holds bytes
        If L > 24 Then                                 ` and there are more than 24 bytes
          SPkt$ = Left$(TXPkt$, 24)                    ` put the first 24 bytes in SPkt$
          TXPkt$ = Right$(TXPkt$, (L - 24))            ` and hold the rest in TXPkt$
        Else
          SPkt$ = TXPkt$                               ` else put all TXPkt$ bytes in SPkt$
          TXPkt$ = “”                                  ` and clear TXPkt$
```

```
        End If
        Call NxtPkt                                          ' bump packet ID number
        SPkt$ = FEND$ & PktHdr$ & Chr$(P) & SPkt$ & FEND$    ' build packet
        MSComm1.Output = SPkt$                               ' send packet
        TFlag = 1                                            ' set transfer flag
        ANFlag = 1                                           ' set ACK/NAK flag
        TCnt = 0                                             ' clear TX timeout counter
        XCnt = 0                                             ' clear TX transfer retry counter
      Else
        TXFlag = 0                                           ' clear TX flag when all bytes sent
        StatusBar1.Panels(4).Text = "Keyboard Active"        ' show keyboard active
      End If
    End If
  End If
End Sub

Public Sub Xfer()
  TCnt = TCnt + 1                                            ' increment TX timeout counter
  If TCnt > 4 Then                                           ' if trying for more than 1 second
    If TFlag = 1 Then                                        ' and transfer flag still set
      TCnt = 0                                               ' reset TCnt
      XCnt = XCnt + 1                                        ' increment transfer retry counter
      If XCnt < 17 Then                                      ' if XCnt not greater than 16
        MSComm1.Output = SPkt$                               ' resend packet
        TCnt = 0                                             ' reset TX timeout counter
      Else
        Call ReSetTX                                         ' else reset TX after eight tries
        Call LenTrap                                         ' manage textbox memory
        Text1.SelStart = Len(Text1.Text)                     ' put cursor to end of text
        Text1.SelText = "  <xfer fault>" & vbCrLf            ' show transfer fault message
      End If
    End If
  End If
  If TCnt > 64 Then                                          ' if more than 16 seconds
    If ANFlag = 1 Then                                       ' and if ACK/NAK flag still set
      Call ReSetTX                                           ' reset TX
      Call LenTrap                                           ' manage textbox memory
      Text1.SelStart = Len(Text1.Text)                       ' put cursor to end of text
      Text1.SelText = "  <ACK/NAK fault>" _
      & vbCrLf                                               ' show ACK/NAK fault message
    End If
  End If
End Sub

Public Sub ReSetTX()
  TFlag = 0                                                  ' reset transfer flag
  TXFlag = 0                                                 ' reset TX message flag
  TNFlag = 0                                                 ' reset next TX packet flag
  ANFlag = 0                                                 ' reset ACK/NAK flag
  NAFlag = 0                                                 ' reset next AutoSend flag
  TCnt = 0                                                   ' reset TCnt
  XCnt = 0                                                   ' reset XCnt
  TXPkt$ = ""                                                ' clear TX message string
  SPkt$ = ""                                                 ' clear send packet string
  ProgressBar1.Value = 0                                     ' clear progress bar
  StatusBar1.Panels(4).Text = "Keyboard Active"              ' show keyboard active
End Sub

Public Sub ASPkt()
  If NAFlag = 0 Then                                         ' if next AutoSend flag reset
    Call GetPkt                                              ' get next message packet(s)
    Temp$ = TPkt$                                            ' use Temp$ for local display
    Call LenTrap                                             ' manage textbox memory
    Text1.SelStart = Len(Text1.Text)                         ' put cursor at end of text
    Text1.SelText = Temp$                                    ' add text to textbox
    TXFlag = 1                                               ' set TX message flag
    TNFlag = 1                                               ' set next TX packet flag
    StatusBar1.Panels(4).Text = "Keyboard Locked"            ' show keyboard locked
    Call SndPkt                                              ' send via SndPkt
    NAFlag = 1                                               ' set next AutoSend flag
  End If
End Sub

Public Sub GetPkt()
  TPkt$ = ASStr$                                             ' message string for AutoSend
End Sub

Public Sub NxtPkt()
  P = P + 1                                                  ' increment packet number
```

```
  If P = 8 Then                                         ' if packet number greater than 7
    P = 0                                               ' reset to 0
  End If
End Sub

Public Sub LenTrap()
  If Len(Text1.Text) > 16000 Then                       ' avoid textbox memory overflow
    Text1.Text = ""                                     ' clear TX textbox
    Text1.SelStart = Len(Text1.Text)                    ' put cursor at end of text
  End If
  If Len(Text2.Text) > 16000 Then                       ' avoid textbox memory overflow
    Text2.Text = ""                                     ' clear RX textbox
    Text2.SelStart = Len(Text2.Text)                    ' put cursor at end of text
  End If
End Sub

Private Sub mnuExit_Click()
  MSComm1.PortOpen = False                              ' close com port
  End                                                   ' done!
End Sub

Private Sub Form_Unload(Cancel As Integer)
  MSComm1.PortOpen = False                              ' close com port
  End                                                   ' done!
End Sub

Private Sub mnuClear_Click()
  Text1.Text = ""                                       ' clear TX textbox
  Text1.SelStart = Len(Text1.Text)                      ' put cursor at end of text
  Text2.Text = ""                                       ' clear RX textbox
  Text2.SelStart = Len(Text2.Text)                      ' put cursor at end of text
End Sub

Private Sub mnuDups_Click()
  If DupFltr = 0 Then                                   ' if show RX dups active
    DupFltr = 1                                         ' toggle to inactive
    mnuDups.Checked = False                             ' and uncheck Show RX Dups
  Else
    DupFltr = 0                                         ' else toggle active
    mnuDups.Checked = True                              ' and check Show RX Dups
  End If
End Sub

Private Sub mnuShw_Click()
  If ShwACK = 1 Then                                    ' if show ACK/NAK active
    ShwACK = 0                                          ' toggle to inactive
    mnuShw.Checked = False                              ' and uncheck Show ACK/NAK
  Else
    ShwACK = 1                                          ' else toggle active
    mnuShw.Checked = True                               ' and check Show ACK/NAK
  End If
End Sub

Private Sub mnuAutoSnd_Click()
  ASFlag = ASFlag Xor 1                                 ' toggle AutoSend flag
  If ASFlag = 0 Then                                    ' if flag reset
    Call ReSetTX                                        ' reset TX
    Text1.ForeColor = QBColor(15)                       ' make letters white
    mnuAutoSnd.Checked = False                          ' uncheck AutoSend
  End If
  If ASFlag = 1 Then                                    ' if flag active
    PCnt = 0                                            ' clear TX tries counter
    NAFlag = 0                                          ' clear next AutoSend flag
    Text1.ForeColor = QBColor(10)                       ' make letters green
    mnuAutoSnd.Checked = True                           ' check AutoSend
  End If
End Sub

Private Sub mnuFN1_Click()
  FNode = 1                                             ' from Node = 1
  Call BldHdr                                           ' build new packet header
  Call RstFrmChk                                        ' reset all From check marks
  mnuFN1.Checked = True                                 ' check Node 1
End Sub

Private Sub mnuFN2_Click()
  FNode = 2                                             ' from Node = 2
  Call BldHdr                                           ' build new packet header
  Call RstFrmChk                                        ' reset all From check marks
  mnuFN2.Checked = True                                 ' check Node 2
End Sub
```

```
Private Sub mnuFN3_Click()
  FNode = 3                                                ' from Node = 3
  Call BldHdr                                              ' build new packet header
  Call RstFrmChk                                           ' reset all From check marks
  mnuFN3.Checked = True                                    ' check Node 3
End Sub

Private Sub mnuFN4_Click()
  FNode = 4                                                ' from Node = 4
  Call BldHdr                                              ' build new packet header
  Call RstFrmChk                                           ' reset all From check marks
  mnuFN4.Checked = True                                    ' check Node 4
End Sub

Public Sub RstFrmChk()
  mnuFN1.Checked = False                                   ' uncheck From Node 1
  mnuFN2.Checked = False                                   ' uncheck From Node 2
  mnuFN3.Checked = False                                   ' uncheck From Node 3
  mnuFN4.Checked = False                                   ' uncheck From Node 4
End Sub

Private Sub mnuTN1_Click()
  TNode = 1                                                ' To Node = 1
  Call BldHdr                                              ' build new packet header
  Call RstToChk                                            ' reset all To check marks
  mnuTN1.Checked = True                                    ' check Node 1
End Sub

Private Sub mnuTN2_Click()
  TNode = 2                                                ' To Node = 2
  Call BldHdr                                              ' build new packet header
  Call RstToChk                                            ' reset all To check marks
  mnuTN2.Checked = True                                    ' check Node 2
End Sub

Private Sub mnuTN3_Click()
  TNode = 3                                                ' To Node = 3
  Call BldHdr                                              ' build new packet header
  Call RstToChk                                            ' reset all To check marks
  mnuTN3.Checked = True                                    ' check Node 3
End Sub

Private Sub mnuTN4_Click()
  TNode = 4                                                ' To Node = 4
  Call BldHdr                                              ' build new packet header
  Call RstToChk                                            ' reset all To check marks
  mnuTN4.Checked = True                                    ' check Node 4
End Sub

Public Sub RstToChk()
  mnuTN1.Checked = False                                   ' uncheck To Node 1
  mnuTN2.Checked = False                                   ' uncheck To Node 2
  mnuTN3.Checked = False                                   ' uncheck To Node 3
  mnuTN4.Checked = False                                   ' uncheck To Node 4
End Sub

Public Sub BldHdr()
  TF = (16 * TNode) + FNode                                ' TF is numeric To/From node address
  PktHdr$ = Chr$(TF)                                       ' Chr$(TF) is To/From packet header
End Sub
```

5.3 DK110K.ASM

```
;  DK110K.ASM 2002.08.01 @ 20:00 CDT
;  See RFM Virtual Wire(r) Development Kit Warranty & License for terms of use
;  Experimental software - NO representation is
;  made that this software is suitable for any purpose
;  Copyright(c) 2000 - 2002, RF Monolithics, Inc.
;  AT89C2051 assembler source code file (TASM 3.01 assembler)
;  Low signal-to-noise protocol for RFM ASH transceiver
;  Integrate & dump PLL (I&D) - 62.40 us tick

          .NOLIST
          #INCLUDE "8051.H"        ;  tasm 8051 include file
          .LIST
```

```
;  constants:

ITMOD     .EQU      022H            ;  set timers 0 and 1 to mode 2
ITICK     .EQU      141             ;  set timer T0 for 62.40 us tick
ISMOD     .EQU      080H            ;  SMOD = 1 in PCON

IBAUD     .EQU      0FAH            ;  19.2 kbps @ 22.1184 MHz, SMOD = 1
ISCON     .EQU      050H            ;  UART mode 1

RMPT      .EQU      159             ;  PLL ramp top value (modulo 0 to 159)
RMPW      .EQU      159             ;  PLL ramp reset (wrap) value
RMPS      .EQU      80              ;  PLL ramp switch value
RMPI      .EQU      20              ;  PLL ramp increment value
RMPA      .EQU      29              ;  PLL 5% advance increment value (20 + 9)
RMPR      .EQU      11              ;  PLL 5% retard increment value (20 - 9)

TXMB      .EQU      044H            ;  TX message buffer start address
RXMB      .EQU      062H            ;  RX message buffer start address
FEND      .EQU      0C0H            ;  FEND framing character (192)
SOPL      .EQU      08AH            ;  SOP low correlator pattern
SOPH      .EQU      0B3H            ;  SOP high correlator pattern
TXRO      .EQU      020H            ;  TX retry timer count

FCSS      .EQU      0FFH            ;  FCS seed
FCSH      .EQU      084H            ;  FCS high XOR mask
FCSL      .EQU      08H             ;  FCS low XOR mask
FCVH      .EQU      0F0H            ;  FCS valid high byte pattern
FCVL      .EQU      0B8H            ;  FCS valid low byte pattern

;  stack:  08H - 021H (26 bytes)

;  bit labels:

WBFLG     .EQU      010H            ;  warm boot flag (future use)
PLLON     .EQU      011H            ;  RX PLL control flag
RXISM     .EQU      012H            ;  RX inverted input sample
RXSMP     .EQU      013H            ;  RX input sample
LRXSM     .EQU      014H            ;  last RX input sample
RXBIT     .EQU      015H            ;  RX input bit
RXBFLG    .EQU      016H            ;  RX input bit flag
SOPFLG    .EQU      017H            ;  SOP detect flag
RXSFLG    .EQU      018H            ;  RX symbol flag
RM        .EQU      019H            ;  RX FCS message bit
OKFLG     .EQU      01AH            ;  RX FCS OK flag

SIFLG     .EQU      01BH            ;  serial in active flag
TSFLG     .EQU      01CH            ;  output TX sample flag
TXSMP     .EQU      01DH            ;  TX output sample
TXBIT     .EQU      01EH            ;  TX message bit
TM        .EQU      01FH            ;  TX FCS message bit
TXFLG     .EQU      020H            ;  TX active flag
TMFLG     .EQU      021H            ;  TX message flag
TOFLG     .EQU      022H            ;  get message time out flag
AMFLG     .EQU      023H            ;  AutoSend message flag
ASFLG     .EQU      024H            ;  AutoSend active flag

SFLG0     .EQU      025H            ;  spare flag 0
SFLG1     .EQU      026H            ;  spare flag 1
SFLG2     .EQU      027H            ;  spare flag 2
SFLG3     .EQU      028H            ;  spare flag 3
SFLG4     .EQU      029H            ;  spare flag 4
SFLG5     .EQU      02AH            ;  spare flag 5
SFLG6     .EQU      02BH            ;  spare flag 6
SFLG7     .EQU      02CH            ;  spare flag 7
SFLG8     .EQU      02DH            ;  spare flag 8
SFLG9     .EQU      02EH            ;  spare flag 9
SFLGA     .EQU      02FH            ;  spare flag A

;  register usage:

;  R0                                  RX data pointer
;  R1                                  TX data pointer
;  R2                                  PLL ramp buffer
;  R3                                  RX FCS buffer A
;  R4                                  not used
;  R5                                  TX FCS buffer A
;  R6                                  TX FCS buffer B
;  R7                                  RX FCS buffer B
```

```
;  byte labels:

BOOT      .EQU      022H          ;  1st byte of flags

RXID      .EQU      026H          ;  RX integrate & dump buffer
RXBL      .EQU      027H          ;  RX low buffer, SOP correlator etc.
RXBH      .EQU      028H          ;  RX high buffer, SOP correlator etc.
RXBB      .EQU      029H          ;  RX symbol decode byte buffer
RMDC      .EQU      02AH          ;  RX symbol decode loop counter
RMBIC     .EQU      02BH          ;  RX symbol decode index pointer
RMBYC     .EQU      02CH          ;  RX message byte counter
RMFCS     .EQU      02DH          ;  RX FCS byte buffer
RMSBC     .EQU      02EH          ;  RX symbol bit counter
RMLPC     .EQU      02FH          ;  RX message loop counter
RMFCC     .EQU      030H          ;  RX message FCS counter, etc.

TMFCC     .EQU      031H          ;  TX timer & loop counter
TXSMC     .EQU      032H          ;  TX output sample counter
TMBIC     .EQU      033H          ;  TX message bit counter
TMBYT     .EQU      034H          ;  TX message byte buffer
TMBYC     .EQU      035H          ;  TX message byte counter
TXSL      .EQU      036H          ;  TX message symbol low buffer
TXSH      .EQU      037H          ;  TX message symbol high buffer
TMFCS     .EQU      038H          ;  TX FCS byte buffer
TXTL      .EQU      039H          ;  TX timer low byte
TXTH      .EQU      03AH          ;  TX timer high byte


BUF0      .EQU      03BH          ;  spare buffer 0
BUF1      .EQU      03CH          ;  spare buffer 1
BUF2      .EQU      03DH          ;  spare buffer 2
BUF3      .EQU      03EH          ;  spare buffer 3
BUF4      .EQU      03FH          ;  spare buffer 4
BUF5      .EQU      040H          ;  spare buffer 5
BUF6      .EQU      041H          ;  spare buffer 6
BUF7      .EQU      042H          ;  spare buffer 7
BUF8      .EQU      043H          ;  spare buffer 8

;  I/O pins:

MAX       .EQU      P1.6          ;  Maxim 218 power (on = 1)

RXPIN     .EQU      P3.2          ;  RX input pin (inverted data)
TXPIN     .EQU      P3.3          ;  TX output pin (on = 1)
PTT       .EQU      P1.7          ;  transmit enable (TX = 0)

PCRCV     .EQU      P3.7          ;  PC (host) input LED (on = 0)
RFRCV     .EQU      P3.5          ;  RX FCS OK LED (on = 0)
RXI       .EQU      P3.4          ;  RX activity LED (on = 0)

ID0       .EQU      P1.2          ;  jumper input bit 0 (dot end)
ID1       .EQU      P1.3          ;  jumper input bit 1
ID2       .EQU      P1.4          ;  jumper input bit 2
ID3       .EQU      P1.5          ;  jumper input bit 3

;  start of code:

          .ORG      00H           ;  hardware reset
          SETB      WBFLG         ;  set warm boot flag
reset:    AJMP      start         ;  jump to start

          .ORG      0BH           ;  timer 0 interrupt vector
t_isr:    ACALL     tick          ;  sampling tick subroutine
          RETI                    ;  interrupt done

          .ORG      023H          ;  serial interrupt vector
s_isr:    ACALL     srio          ;  serial I/O subroutine
          CLR       TI            ;  clear TI (byte sent) flag
          CLR       RI            ;  clear RI (byte received) flag
          RETI                    ;  interrupt done

          .ORG      040H          ;  above interrupt code space
start:    ACALL     setup         ;  initialization code

main:     JNB       AMFLG,mn0     ;  skip if AutoSend idle
          CLR       PCRCV         ;  else turn PCRCV LED on
          ACALL     do_as         ;  do AutoSend
          SETB      PCRCV         ;  turn PCRCV LED off
mn0:      ACALL     rxsop         ;  do RX SOP detect
          JNB       SOPFLG,main   ;  if not SOP loop to main
          ACALL     do_rx         ;  else do RX message
```

```
mn_d:     AJMP      main             ;  and loop to main

do_rx:    CLR       ES               ;  deactivate serial interrupts
          ACALL     rxmsg            ;  decode RX message
          CLR       PLLON            ;  idle RX PLL
          ACALL     rxfcs            ;  test RX message FCS
          JNB       OKFLG,rx0        ;  reset if FCS error
          ACALL     rxsnd            ;  else send RX message to host
rx0:      ACALL     rxrst            ;  reset for next RX message
          SETB      PLLON            ;  enable RX PLL
          CLR       TI               ;  clear TI flag
          CLR       RI               ;  clear RI flag
          SETB      ES               ;  activate serial interrupts
rx_d:     RET                        ;  RX done

tick:     PUSH      PSW              ;  push status
          PUSH      ACC              ;  push accumulator
          MOV       C,RXPIN          ;  read RX input pin
          MOV       RXISM,C          ;  store as inverted RX sample
          JNB       TSFLG,tic0       ;  skip if TX sample out idle
          MOV       A,TXSMC          ;  else get sample count
          JZ        tic0             ;  skip if 0
          MOV       C,TXBIT          ;  else load TX bit
          MOV       TXPIN,C          ;  into TX output pin
          DEC       TXSMC            ;  decrement sample count
tic0:     JNB       PLLON,tic1       ;  skip if PLL idle
          ACALL     pll              ;  else run RX PLL
tic1:     JNB       TOFLG,tic2       ;  skip if get message timeout idle
          INC       TMFCC            ;  else bump timeout counter
          MOV       A,TMFCC          ;  get counter
          CJNE      A,#50,tic2       ;  skip if counter <> 50 (5.2 ms)
          CLR       TOFLG            ;  else reset time out flag
          MOV       TMFCC,#0         ;  reset counter
tic2:     JNB       ASFLG,tick_d     ;  done if AutoSend idle
          INC       TXTL             ;  else bump TX timer low
          MOV       A,TXTL           ;  load TX timer low
          JNZ       tick_d           ;  done if no rollover
          INC       TXTH             ;  else bump TX timer high
          MOV       A,TXTH           ;  load timer
          CLR       C                ;  clear borrow
          SUBB      A,#TXRO          ;  subtract TX retry count
          JNZ       tick_d           ;  if <> 0 done for now
          SETB      AMFLG            ;  else set AM message flag
          CLR       A                ;  clear A
          MOV       TXTL,A           ;  clear TX timer low
          MOV       TXTH,A           ;  clear TX timer high
tick_d:   POP       ACC              ;  pop accumulator
          POP       PSW              ;  pop status
          RET                        ;  tick done

pll:      MOV       C,RXSMP          ;  load RX sample
          MOV       LRXSM,C          ;  into last RX sample
          MOV       C,RXISM          ;  get inverted RX sample
          CPL       C                ;  invert
          MOV       RXSMP,C          ;  and store
          JNC       pll0             ;  if <> 1 jump to pll0
          INC       RXID             ;  else increment I&D
pll0:     JNB       LRXSM,pll1       ;  if last sample 1
          CPL       C                ;  invert current sample
pll1:     JNC       pll4             ;  if no edge jump to pll4
          MOV       A,R2             ;  else get PLL value
          CLR       C                ;  clear borrow
          SUBB      A,#RMPS          ;  subtract ramp switch value
          JC        pll3             ;  if < 0 then retard PLL
pll2:     MOV       A,R2             ;  else get PLL value
          ADD       A,#RMPA          ;  add (RMPI + 5%)
          MOV       R2,A             ;  store PLL value
          AJMP      pll5             ;  and jump to pll5
pll3:     MOV       A,R2             ;  get PLL value
          ADD       A,#RMPR          ;  add (RMPI - 5%)
          MOV       R2,A             ;  store PLL value
          AJMP      pll5             ;  and jump to pll5
pll4:     MOV       A,R2             ;  get PLL value
          ADD       A,#RMPI          ;  add ramp increment
          MOV       R2,A             ;  store new PLL value
pll5:     CLR       C                ;  clear borrow
          MOV       A,R2             ;  get PLL ramp value
          SUBB      A,#RMPT          ;  subtract ramp top
          JC        pllD             ;  if < 0 don't wrap
pll6:     MOV       A,R2             ;  else get PLL value
          CLR       C                ;  clear borrow
```

```
        SUBB    A,#RMPW         ;  subtract reset value
        MOV     R2,A            ;  and store result
        CLR     C               ;  clear borrow
        MOV     A,RXID          ;  get I&D buffer
        SUBB    A,#5            ;  subtract 5
        JNC     pll7            ;  if I&D count => 5 jump to pll7
        CLR     RXBIT           ;  else RX bit = 0 for I&D count < 5
        SETB    RXBFLG          ;  set new RX bit flag
        MOV     RXID,#0         ;  clear the I&D buffer
        AJMP    pll8            ;  and jump to pll8
pll7:   SETB    RXBIT           ;  RX bit = 1 for I&D count => 5
        SETB    RXBFLG          ;  set new RX bit flag
        MOV     RXID,#0         ;  clear the I&D buffer
pll8:   JB      SOPFLG,pllA     ;  skip after SOP detect
        MOV     A,RXBH          ;  else get RXBH
        CLR     C               ;  clear carry
        RRC     A               ;  rotate right
        JNB     RXBIT,pll9      ;  if bit = 0 jump to pll9
        SETB    ACC.7           ;  else set 7th bit
pll9:   MOV     RXBH,A          ;  store RXBH
        MOV     A,RXBL          ;  get RXBL
        RRC     A               ;  shift and pull in carry
        MOV     RXBL,A          ;  store RXBL
        AJMP    pll_d           ;  done for now
pllA:   MOV     A,RXBL          ;  get RXBL
        CLR     C               ;  clear carry
        RRC     A               ;  shift right
        JNB     RXBIT,pllB      ;  if bit = 0 jump to pllB
        SETB    ACC.5           ;  else set 5th bit
pllB:   MOV     RXBL,A          ;  store RXBL
        INC     RMSBC           ;  bump bit counter
        MOV     A,RMSBC         ;  get counter
        CJNE    A,#6,pllC       ;  if <> 6 jump to pllC
        MOV     RXBB,RXBL       ;  else get symbol
        MOV     RMSBC,#0        ;  reset counter
        SETB    RXSFLG          ;  set symbol flag
pllC:   AJMP    pll_d           ;  done
pllD:   CLR     RXBFLG          ;  clear RXBFLG
pll_d:  RET                     ;  PLL done

rxsop:  JNB     RXBFLG,sop_d    ;  done if no RX bit flag
        CLR     RXBFLG          ;  else clear RX bit flag
        MOV     A,RXBL          ;  get low RX buffer
        CJNE    A,#SOPL,sop_d   ;  done if <> SOPL
        MOV     A,RXBH          ;  else get high RX buffer
        CJNE    A,#SOPH,sop_d   ;  done if <> SOPH
        CLR     A               ;  else clear A
        MOV     RXBL,A          ;  clear RX low buffer
        MOV     RXBH,A          ;  clear RX high buffer
        MOV     RMSBC,A         ;  clear RX symbol bit counter
        CLR     RXSFLG          ;  clear RX symbol flag
        SETB    SOPFLG          ;  set SOP detected flag
        CLR     RXI             ;  RXI LED on
sop_d:  RET                     ;  SOP detect done

rxmsg:  JNB     RXSFLG,rxmsg    ;  wait for RX symbol flag
        CLR     RXSFLG          ;  clear RX symbol flag
rxm1:   MOV     DPTR,#smbl      ;  point to RX symbol decode table
        MOV     RMDC,#16        ;  16 symbol decode table entries
        MOV     RMBIC,#0        ;  index into symbol table
rxm2:   MOV     A,RMBIC         ;  load index into A
        MOVC    A,@A+DPTR       ;  get table entry
        XRL     A,RXBB          ;  XOR to compare with RXBB
        JZ      rxm3            ;  exit loop with decoded nibble
        INC     RMBIC           ;  else bump index
        DJNZ    RMDC,rxm2       ;  and try to decode again
rxm3:   MOV     A,RMBIC         ;  get decoded nibble
        SWAP    A               ;  swap to high nibble
        MOV     RXBH,A          ;  into RXBH (low nibble is high)
rxm4:   JNB     RXSFLG,rxm4     ;  wait for symbol flag
        CLR     RXSFLG          ;  clear flag
rxm5:   MOV     DPTR,#smbl      ;  point to symbol decode table
        MOV     RMDC,#16        ;  16 symbol decode table entries
        MOV     RMBIC,#0        ;  reset symbol table index
rxm6:   MOV     A,RMBIC         ;  load index into A
        MOVC    A,@A+DPTR       ;  get table entry
        XRL     A,RXBB          ;  XOR to compare with RXBB
        JZ      rxm7            ;  exit loop with decoded nibble
        INC     RMBIC           ;  else bump index
        DJNZ    RMDC,rxm6       ;  and try to decode again
rxm7:   MOV     A,RMBIC         ;  get decoded nibble
```

```
        ORL     A,RXBH          ;  add RXBH low
        SWAP    A               ;  nibbles now in right order
        MOV     RXBH,A          ;  store in RXBH
        MOV     @R0,RXBH        ;  and store in RX message buffer
        CJNE    R0,#RXMB,rxm8   ;  skip if not 1st message byte
        MOV     A,RXBH          ;  else get 1st byte
        ANL     A,#63           ;  mask upper 2 bits
        MOV     RMBYC,A         ;  load message byte counter
        MOV     RMFCC,A         ;  and RX message loop counter
        CLR     C               ;  clear borrow
        SUBB    A,#28           ;  compare # bytes to 28
        JC      rxm8            ;  skip if < 28
        MOV     RMBYC,#4        ;  else force byte counter to 4
        MOV     RMFCC,#4        ;  and force loop counter to 4
rxm8:   INC     R0              ;  bump pointer
        DJNZ    RMFCC,rxmsg     ;  if <> 0 get another byte
        MOV     R0,#RXMB        ;  reset RX message pointer
        SETB    RXI             ;  turn LED off
rxm_d:  RET                     ;  RX message done

rxfcs:  MOV     RMFCC,RMBYC     ;  move byte count to loop counter
rxf0:   MOV     RMFCS,@R0       ;  get next message byte
        INC     R0              ;  bump pointer
        ACALL   b_rfcs          ;  build FCS
        DJNZ    RMFCC,rxf0      ;  loop for next byte
        ACALL   a_rfcs          ;  test FCS
rxf_d:  RET                     ;  RX FCS done
rxsnd:  CLR     PCRCV           ;  turn PC LED on
        DEC     RMBYC           ;  don't send
        DEC     RMBYC           ;  the 2 FCS bytes
        MOV     R0,#RXMB        ;  reset RX message pointer
        MOV     @R0,#FEND       ;  replace # bytes with 1st FEND
        CLR     TI              ;  clear TI flag
rxs1:   MOV     SBUF,@R0        ;  send byte
rxs2:   JNB     TI,rxs2         ;  wait until byte sent
        CLR     TI              ;  clear TI flag
        INC     R0              ;  bump pointer
        DJNZ    RMBYC,rxs1      ;  loop to echo message
        MOV     SBUF,#FEND      ;  add 2nd FEND
rxs3:   JNB     TI,rxs3         ;  wait until byte sent
        CLR     TI              ;  clear TI flag
        SETB    RFRCV           ;  turn FCS LED off
        SETB    PCRCV           ;  turn PC LED off
rxs_d:  RET                     ;  send RX message done

rxrst:  CLR     A               ;  clear A
        MOV     RXBH,A          ;  clear buffer
        MOV     RXBL,A          ;  clear buffer
        MOV     RXBB,A          ;  clear buffer
        MOV     RMBYC,A         ;  clear rx byte count
        MOV     RMFCC,A         ;  clear loop counter
        MOV     R0,#RXMB        ;  point R0 to message start
        CLR     OKFLG           ;  clear packet OK flag
        CLR     SOPFLG          ;  enable SOP test
        SETB    RXI             ;  RXI LED off
rxr_d:  RET                     ;  RX reset done

b_rfcs: MOV     RMLPC,#8        ;  load loop count of 8
brf0:   CLR     C               ;  clear carry bit
        MOV     A,RMFCS         ;  load RX message byte
        RRC     A               ;  shift lsb into carry
        MOV     RMFCS,A         ;  store shifted message byte
        MOV     RM,C            ;  load RM with lsb
        CLR     C               ;  clear carry bit
        MOV     A,R3            ;  load high FCS byte
        RRC     A               ;  shift right
        MOV     R3,A            ;  store shifted high FCS
        MOV     A,R7            ;  load low FCS byte
        RRC     A               ;  shift and pull in bit for FCS high
        MOV     R7,A            ;  store shifted low FCS
        JNB     RM,brf1         ;  if lsb of low FCS = 0, jump to brf1
        CPL     C               ;  else complement carry bit
brf1:   JNC     brf2            ;  if RM XOR (low FCS lsb) = 0 jump to brf2
        MOV     A,R3            ;  else load high FCS
        XRL     A,#FCSH         ;  and XOR with high FCS poly
        MOV     R3,A            ;  store high FCS
        MOV     A,R7            ;  load low FCS
        XRL     A,#FCSL         ;  XOR with low FCS poly
        MOV     R7,A            ;  store low FCS
brf2:   DJNZ    RMLPC,brf0      ;  loop through bits in message byte
brfcs_d: RET                    ;  done this pass
```

```
a_rfcs:   MOV       A,R3            ;  load FCS high
          XRL       A,#FCVH         ;  compare with 0F0H
          JNZ       arf0            ;  if <> 0 jump to arf0
          MOV       A,R7            ;  load FCS low
          XRL       A,#FCVL         ;  else compare with 0B8H
          JNZ       arf0            ;  if <> 0 jump to arf0
          CLR       RFRCV           ;  else turn FCS LED on
          SETB      OKFLG           ;  set FCS OK flag
arf0:     MOV       R3,#FCSS        ;  reseed FCS high
          MOV       R7,#FCSS        ;  reseed FCS low
arfcs_d:  RET                       ;  RX FCS done

srio:     PUSH      PSW             ;  save
          PUSH      ACC             ;  environment
          JNB       TI,sr_0         ;  skip if TI flag clear
          CLR       TI              ;  else clear TI flag
sr_0:     JNB       RI,sr_1         ;  skip if RI flag clear
          CLR       RI              ;  else clear RI flag
          JNB       SIFLG,sr_1      ;  skip if serial in flag reset
          CLR       PCRCV           ;  else turn PC LED on
          ACALL     do_tx           ;  get & TX host message
          SETB      PCRCV           ;  turn PC LED off
sr_1:     POP       ACC             ;  restore
          POP       PSW             ;  environment
          RET                       ;  serial in done

do_as:    CLR       PLLON           ;  idle RX PLL
          ACALL     hello2          ;  get AutoSend message
          ACALL     txfcs           ;  build and add FCS
          ACALL     txpre           ;  send TX preamble
          ACALL     txmsg           ;  send TX message
          ACALL     txrst           ;  reset TX
          SETB      PLLON           ;  enable RX PLL
          RET                       ;  TX message done

do_tx:    ACALL     txget           ;  get TX message from host
          JNB       TXFLG,do0       ;  skip if TXFLG not set
          CLR       PLLON           ;  else idle RX PLL
          ACALL     txfcs           ;  build and add FCS
          ACALL     txpre           ;  send TX preamble
          ACALL     txmsg           ;  send TX message
do0:      ACALL     txrst           ;  reset TX
          SETB      PLLON           ;  enable RX PLL
          RET                       ;  TX message done

txget:    MOV       A,SBUF          ;  get byte
          MOV       TMBYT,A         ;  copy to TMBYT
          XRL       A,#FEND         ;  compare to FEND
          JZ        txg0            ;  if FEND jump to txg0
          AJMP      txg_d           ;  else done
txg0:     MOV       @R1,TMBYT       ;  store 1st FEND
          INC       TMBYC           ;  bump TX byte counter
txg1:     MOV       TMFCC,#0        ;  reset timeout counter
          SETB      TOFLG           ;  set timeout flag
          CLR       RI              ;  reset flag
txg2:     JNB       TOFLG,txg3      ;  if TOFLG reset jump to txg3
          JNB       RI,txg2         ;  else loop until next byte
          CLR       RI              ;  reset RI flag
          CLR       TOFLG           ;  reset TOFLG
          AJMP      txg4            ;  and jump to txg4
txg3:     MOV       TMBYC,#2        ;  look like null message
          AJMP      txg6            ;  and jump to txg6
txg4:     MOV       A,SBUF          ;  get byte
          MOV       TMBYT,A         ;  copy to TMBYT
          INC       TMBYC           ;  bump byte counter
          INC       R1              ;  bump pointer R1
          MOV       @R1,TMBYT       ;  store byte
          MOV       A,TMBYC         ;  load counter
          CLR       C               ;  clear carry
          SUBB      A,#26           ;  test for 26 bytes
          JZ        txg5            ;  if 26 handle overflow at txg5
          MOV       A,TMBYT         ;  else load byte
          CJNE      A,#FEND,txg1    ;  if <> FEND loop to txg1
          AJMP      txg6            ;  else jump to txg6 on 2nd FEND
txg5:     MOV       @R1,#FEND       ;  force 2nd FEND
txg6:     MOV       R1,#TXMB        ;  reset TX message pointer
          MOV       A,TMBYC         ;  get byte count
          CJNE      A,#2,txg7       ;  if <> 2 jump to txg7
          MOV       TMBYC,#0        ;  else reset byte counter
          AJMP      txg_d           ;  jump to txg_d
```

```
txg7:     CLR       SIFLG           ;  idle serial in
          SETB      TXFLG           ;  set TX flag
txg_d:    RET                       ;  get TX message done

txfcs:    INC       TMBYC           ;  # bytes including FCS
          MOV       @R1,TMBYC       ;  replace 1st FEND with # bytes
          MOV       TMFCC,TMBYC     ;  move byte count to loop counter
          DEC       TMFCC           ;  loop count is 2 less
          DEC       TMFCC           ;  than # bytes including FCS
txf0:     MOV       TMFCS,@R1       ;  get next message byte
          INC       R1              ;  bump pointer
          ACALL     b_tfcs          ;  build FCS
          DJNZ      TMFCC,txf0      ;  loop for next byte
          ACALL     a_tfcs          ;  add FCS
          MOV       R1,#TXMB        ;  reset TX message pointer
          SETB      TMFLG           ;  set TX message flag
txf_d:    RET                       ;  TX FCS done

txpre:    CLR       PTT             ;  turn PTT on
          MOV       B,#200          ;  load PTT delay count
txp0:     DJNZ      B,txp0          ;  loop to delay
txp1:     MOV       DPTR,#tstrt     ;  point to TX start table
          MOV       B,#0            ;  clear B
          MOV       A,B             ;  B holds table offset
          MOVC      A,@A+DPTR       ;  load table entry
          MOV       TMBYT,A         ;  into TMBYT
          MOV       TMBIC,#4        ;  load bit count
          MOV       TXSMC,#0        ;  clear sample count
          SETB      TSFLG           ;  turn TX sample out on
txp2:     MOV       A,TXSMC         ;  get sample count
          JNZ       txp2            ;  loop until sample count 0
          MOV       A,TMBIC         ;  get bit count
          JNZ       txp3            ;  if <> 0 jump to txp3
          MOV       A,B             ;  else get current offset (0 to 11)
          CLR       C               ;  clear carry
          SUBB      A,#11           ;  subtract ending offset
          JZ        txp_d           ;  if 0  done
          INC       B               ;  else bump byte count
          MOV       A,B             ;  get count/offset
          MOVC      A,@A+DPTR       ;  load table entry
          MOV       TMBYT,A         ;  into TMBYT
          MOV       TMBIC,#4        ;  reload bit count
txp3:     MOV       A,TMBYT         ;  get TX message byte
          CLR       C               ;  clear carry
          RRC       A               ;  shift right into carry
          MOV       TXBIT,C         ;  load next bit
          MOV       TMBYT,A         ;  store shifted message byte
          DEC       TMBIC           ;  decrement bit count
          MOV       TXSMC,#8        ;  reload sample count
          AJMP      txp2            ;  loop again
txp_d:    RET                       ;  TX preamble done

txmsg:    MOV       B,#1            ;  count 1st byte sent
          MOV       R1,#TXMB        ;  reset TX message pointer
          MOV       A,@R1           ;  get 1st TX message byte
          MOV       TMBYT,A         ;  into TMBYT
          MOV       DPTR,#smbl      ;  point to symbol table
          ANL       A,#0FH          ;  clean offset
          MOVC      A,@A+DPTR       ;  get 6-bit symbol
          MOV       TXSL,A          ;  move to TXSL
          MOV       A,TMBYT         ;  get TMBYT
          SWAP      A               ;  swap nibbles
          ANL       A,#0FH          ;  clean offset
          MOVC      A,@A+DPTR       ;  get 6-bit symbol
          MOV       TXSH,A          ;  move to TXSH
          MOV       TMBIC,#12       ;  set bit count to 12
          MOV       TXSMC,#0        ;  clear sample count
txm0:     MOV       A,TXSMC         ;  get sample count
          JNZ       txm0            ;  loop until sample count 0
          MOV       A,TMBIC         ;  get bit count
          CLR       C               ;  clear carry
          SUBB      A,#7            ;  subtract 7
          JNC       txm1            ;  if => 7 jump to txm1
          MOV       A,TMBIC         ;  else get bit count
          JNZ       txm2            ;  if > 0 jump to txm2
          MOV       A,B             ;  else get current byte number
          CLR       C               ;  clear carry
          SUBB      A,TMBYC         ;  subtract TX message byte count
          JZ        txm3            ;  if 0 done
          INC       R1              ;  else bump byte pointer
          INC       B               ;  and bump byte counter
```

```
          MOV       A,@R1           ;  get next byte
          MOV       TMBYT,A         ;  into TMBYT
          MOV       DPTR,#smbl      ;  point to symbol table
          ANL       A,#0FH          ;  offset
          MOVC      A,@A+DPTR       ;  get 6-bit symbol
          MOV       TXSL,A          ;  move to TXSL
          MOV       A,TMBYT         ;  get TMBYT
          SWAP      A               ;  swap nibbles
          MOV       DPTR,#smbl      ;  point to symbol table
          ANL       A,#0FH          ;  clean offset
          MOVC      A,@A+DPTR       ;  get 6-bit symbol
          MOV       TXSH,A          ;  move to TXSH
          MOV       TMBIC,#12       ;  set bit count to 12
txm1:     MOV       A,TXSL          ;  get low TX symbol
          CLR       C               ;  clear carry
          RRC       A               ;  shift right into carry
          MOV       TXBIT,C         ;  load next bit
          MOV       TXSL,A          ;  store shifted message byte
          DEC       TMBIC           ;  decrement bit count
          MOV       TXSMC,#8        ;  reload sample count
          AJMP      txm0            ;  loop again
txm2:     MOV       A,TXSH          ;  get high TX symbol
          CLR       C               ;  clear carry
          RRC       A               ;  shift right into carry
          MOV       TXBIT,C         ;  load next bit
          MOV       TXSH,A          ;  store shifted message byte
          DEC       TMBIC           ;  decrement bit count
          MOV       TXSMC,#8        ;  reload sample count
          AJMP      txm0            ;  loop again
txm3:     CLR       TSFLG           ;  clear TX sample out flag
          CLR       TXPIN           ;  clear TX out pin
          SETB      PTT             ;  turn PTT off
txm_d:    RET                       ;  TX message done

txrst:    CLR       TMFLG           ;  clear TX message flag
          CLR       AMFLG           ;  clear AutoSend message flag
          CLR       A               ;  reset for next TX
          MOV       TMBYT,A         ;  clear TX message byte
          MOV       TMFCC,A         ;  clear TX FCS count
          MOV       TXSMC,A         ;  clear TX out count
          MOV       TXSL,A          ;  clear TX symbol low
          MOV       TXSH,A          ;  clear TX symbol high
          MOV       R1,#TXMB        ;  point R1 to message start
          JB        ASFLG,txr_d     ;  skip if in AutoSend
          MOV       TMBYC,A         ;  clear TX message byte count
          CLR       TXFLG           ;  clear TX flag
          SETB      SIFLG           ;  set serial in flag active
txr_d:    RET                       ;  TX reset done

b_tfcs:   MOV       B,#8            ;  load loop count of 8
btf0:     CLR       C               ;  clear carry bit
          MOV       A,TMFCS         ;  load TX message byte
          RRC       A               ;  shift lsb into carry
          MOV       TMFCS,A         ;  store shifted message byte
          MOV       TM,C            ;  load TM with lsb
          CLR       C               ;  clear carry bit
          MOV       A,R5            ;  load high FCS byte
          RRC       A               ;  shift right
          MOV       R5,A            ;  store shifted high FCS
          MOV       A,R6            ;  load low FCS byte
          RRC       A               ;  shift and pull in bit for FCS high
          MOV       R6,A            ;  store shifted low FCS
          JNB       TM,btf1         ;  if lsb of low FCS = 0, jump to btf1
          CPL       C               ;  else complement carry bit
btf1:     JNC       btf2            ;  if TM XOR (low FCS lsb) = 0 jump to btf2
          MOV       A,R5            ;  else load high FCS
          XRL       A,#FCSH         ;  and XOR with high FCS poly
          MOV       R5,A            ;  store high FCS
          MOV       A,R6            ;  load low FCS
          XRL       A,#FCSL         ;  XOR with low FCS poly
          MOV       R6,A            ;  store low FCS
btf2:     DJNZ      B,btf0          ;  loop through bits in message byte
btfcs_d:  RET                       ;  done this pass

a_tfcs:   MOV       A,R6            ;  load FCS (high/low switch)
          CPL       A               ;  1's complement
          MOV       @R1,A           ;  store at end of TX message
          INC       R1              ;  increment TX message byte pointer
          MOV       A,R5            ;  load FCS (high/low switch)
          CPL       A               ;  1's complement
          MOV       @R1,A           ;  store at end of TX message
```

```
         MOV      R5,#FCSS        ;  reseed FCS high
         MOV      R6,#FCSS        ;  reseed FCS low
atfcs_d: RET                      ;  add TX FCS done

setup:   CLR      EA              ;  disable interrupts
         SETB     PTT             ;  turn PTT off
         CLR      TXPIN           ;  turn TX modulation off

tick_su: MOV      TMOD,#ITMOD     ;  set timers T0 and T1 to mode 2
         CLR      TR0             ;  stop timer T0
         CLR      TF0             ;  clear T0 overflow
         MOV      TH0,#ITICK      ;  load count for 62.40 us tick
         MOV      TL0,#ITICK      ;  load count for 62.40 us tick
         SETB     TR0             ;  start timer T0
         SETB     ET0             ;  unmask T0 interrupt
uart_su: SETB     MAX             ;  power up Maxim RS232 converter
         CLR      TR1             ;  stop timer T1
         CLR      TF1             ;  clear T1 overflow
         MOV      TH1,#IBAUD      ;  load baud rate count
         MOV      TL1,#IBAUD      ;  load baud rate count
         MOV      PCON,#ISMOD     ;  SMOD = 1 for baud rate @ 22.1184 MHz
         SETB     TR1             ;  start baud rate timer T1
         MOV      SCON,#ISCON     ;  enable UART mode 1
         MOV      A,SBUF          ;  clear out UART RX buffer
         CLR      A               ;  clear A
         CLR      RI              ;  clear get flag
         CLR      TI              ;  clear TI flag
         ACALL    hello           ;  send start up message
         ACALL    initr           ;  initialize TX & RX
         SETB     SIFLG           ;  set serial in flag active
         MOV      C,ID0           ;  read ID0
         JC       ser_on          ;  skip if no ID0 jumper
         ACALL    hello2          ;  else do AutoSend
ser_on:  SETB     ES              ;  enable serial ISR
isr_on:  SETB     EA              ;  enable interrupts
         SETB     PLLON           ;  activate RX PLL
setup_d: RET                      ;  setup done

initr:   ANL      BOOT,#1         ;  warm boot (don't reset WBFLG)
         MOV      R0,#35          ;  starting here
         MOV      B,#93           ;  for 93 bytes
         CLR      A               ;  clear A
clr_r:   MOV      @R0,A           ;  clear RAM
         INC      R0              ;  bump RAM pointer
         DJNZ     B,clr_r         ;  loop again
         MOV      R0,#RXMB        ;  load RX buffer pointer
         MOV      R1,#TXMB        ;  load TX buffer pointer
         MOV      R2,A            ;  clear R2
         MOV      R3,#FCSS        ;  seed R3
         MOV      R5,#FCSS        ;  seed R5
         MOV      R6,#FCSS        ;  seed R6
         MOV      R7,#FCSS        ;  seed R7
         CLR      SOPFLG          ;  clear SOPFLG
         SETB     PT0             ;  tick is 1st priority
ini_d:   RET                      ;  done

hello:   MOV      DPTR,#table     ;  point to table
         MOV      B,#12           ;  load loop count in B
         MOV      R7,#0           ;  R7 has 1st table entry
snd_h:   MOV      A,R7            ;  move table offset into A
         MOVC     A,@A+DPTR       ;  load table byte
         CLR      TI              ;  reset TI flag
         MOV      SBUF,A          ;  send byte
nxt_tx:  JNB      TI,nxt_tx       ;  wait until sent
         INC      R7              ;  bump index
         DJNZ     B,snd_h         ;  loop to send message
hello_d: RET                      ;  done

hello2:  MOV      DPTR,#tbl_2     ;  point to table 2
         MOV      R1,#TXMB        ;  reset TX buffer pointer
         MOV      B,#8            ;  loop count for 8 bytes
         MOV      TMBYC,#0        ;  offset for 1st table entry
snd_h2:  MOV      A,TMBYC         ;  move table offset into A
         MOVC     A,@A+DPTR       ;  load table byte
         MOV      @R1,A           ;  into TX buffer
         INC      TMBYC           ;  increment TMBYC
         INC      R1              ;  increment R1
         DJNZ     B,snd_h2        ;  loop to load message
         MOV      R1,#TXMB        ;  reset TX pointer
         CLR      SIFLG           ;  reset serial input
         SETB     TXFLG           ;  set TX flag
```

```
          SETB      ASFLG          ;  set AutoSend flag
helo2_d   RET

;  tables:

tstrt:    .BYTE     10             ;  preamble/SOP table
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     10             ;  table data
          .BYTE     8              ;  table data
          .BYTE     3              ;  table data
          .BYTE     11             ;  table data

smbl:     .BYTE     13             ;  4-to-6 bit table
          .BYTE     14             ;  table data
          .BYTE     19             ;  table data
          .BYTE     21             ;  table data
          .BYTE     22             ;  table data
          .BYTE     25             ;  table data
          .BYTE     26             ;  table data
          .BYTE     28             ;  table data
          .BYTE     35             ;  table data
          .BYTE     37             ;  table data
          .BYTE     38             ;  table data
          .BYTE     41             ;  table data
          .BYTE     42             ;  table data
          .BYTE     44             ;  table data
          .BYTE     50             ;  table data
          .BYTE     52             ;  table data
          .BYTE     00             ;  overflow

table:    .BYTE     192            ;  start up message
          .BYTE     ‘ ‘            ;  table data
          .BYTE     ‘D’            ;  table data
          .BYTE     ‘K’            ;  table data
          .BYTE     ‘1’            ;  table data
          .BYTE     ‘1’            ;  table data
          .BYTE     ‘0’            ;  table data
          .BYTE     ‘K’            ;  table data
          .BYTE     ‘:’            ;  table data
          .BYTE     ‘ ‘            ;  table data
          .BYTE     ‘ ‘            ;  table data
          .BYTE     192            ;  table data

tbl_2:    .BYTE     192            ;  table data
          .BYTE     ‘H’            ;  table data
          .BYTE     ‘e’            ;  table data
          .BYTE     ‘l’            ;  table data
          .BYTE     ‘l’            ;  table data
          .BYTE     ‘o’            ;  table data
          .BYTE     ‘ ‘            ;  table data
          .BYTE     192            ;  table data

          .END                     ;  end of source code
```

5.4 V110T05B.FRM

```
VERSION 5.00
Object = “{648A5603-2C6E-101B-82B6-000000000014}#1.1#0"; ”MSCOMM32.OCX"
Object = “{F9043C88-F6F2-101A-A3C9-08002B2F49FB}#1.2#0"; ”COMDLG32.OCX"
Begin VB.Form Form1
   Caption         =   “V110T05B Terminal Program for DK110K Protocol”
   ClientHeight    =   4335
   ClientLeft      =   165
   ClientTop       =   735
   ClientWidth     =   6375
   BeginProperty Font
      Name            =   “MS Sans Serif”
      Size            =   9.75
      Charset         =   0
      Weight          =   400
      Underline       =   0   ‘False
      Italic          =   0   ‘False
```

```
      Strikethrough   =   0   'False
   EndProperty
   LinkTopic       =   "Form1"
   MaxButton       =   0   'False
   ScaleHeight     =   4335
   ScaleWidth      =   6375
   StartUpPosition =   3  'Windows Default

   Begin MSComDlg.CommonDialog CommonDialog1
      Left            =   240
      Top             =   3600
      _ExtentX        =   688
      _ExtentY        =   688
      _Version        =   393216
   End
   Begin VB.TextBox Text2
      BeginProperty Font
         Name            =   "System"
         Size            =   9.75
         Charset         =   0
         Weight          =   700
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      Height          =   2052
      Left            =   120
      Locked          =   -1  'True
      MultiLine       =   -1  'True
      ScrollBars      =   2  'Vertical
      TabIndex        =   1
      Top             =   0
      Width           =   6135
   End
   Begin VB.Timer Timer1
      Left            =   720
      Top             =   3600
   End
   Begin MSCommLib.MSComm MSComm1
      Left            =   1200
      Top             =   3600
      _ExtentX        =   794
      _ExtentY        =   794
      _Version        =   393216
      DTREnable       =   -1  'True
   End
   Begin VB.TextBox Text1
      BeginProperty Font
         Name            =   "System"
         Size            =   9.75
         Charset         =   0
         Weight          =   700
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      Height          =   2052
      Left            =   120
      MultiLine       =   -1  'True
      ScrollBars      =   2  'Vertical
      TabIndex        =   0
      Top             =   2160
      Width           =   6135
   End
   Begin VB.Menu mnuFile
      Caption         =   "&File"
      Begin VB.Menu mnuClear
         Caption         =   "&Clear"
      End
      Begin VB.Menu mnuAutoSnd
         Caption         =   "&AutoSend"
      End
      Begin VB.Menu mnuExit
         Caption         =   "E&xit"
      End
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
```

```
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False

' V110T05B.FRM, 2002.08.07 @ 08:00 CDT
' See RFM Virtual Wire(r) Development Kit Warranty & License for terms of use
' Experimental software - NO representation is
' made that this software is suitable for any purpose
' Copyright(c) 2000 - 2002, RF Monolithics, Inc.
' For experimental use with the RFM DR1200-DK and DR1201-DK
' and DR1300-DK ASH Transceiver Virtual Wire(R) Development Kits
' For protocol software version DK110K.ASM
' Check www.rfm.com for latest software updates
' Compiled in Microsoft Visual Basic 6.0

' global variables
  Dim ASMsg$                                              ' AutoSend string
  Dim ComData$                                            ' string from com input
  Dim ComTime!                                            ' InCom timer
  Dim InDel!                                              ' InCom timer delay value
  Dim FEND$                                               ' packet framing character
  Dim J As Integer                                        ' FEND$ string position
  Dim Q As Integer                                        ' RPkt$ length
  Dim RPkt$                                               ' RX message FIFO string
  Dim R2Pkt$                                              ' RX message display string
  Dim KeyIn$                                              ' keystroke input buffer
  Dim Pkt$                                                ' TX message string
  Dim Temp$                                               ' temp string buffer
  Dim N As Integer                                        ' TX message byte counter
  Dim TXFlag As Integer                                   ' TX flag
  Dim TXCnt As Integer                                    ' TX try counter
  Dim TXTO As Integer                                     ' TX timeout counter
  Dim ASFlag As Integer                                   ' AutoSend flag

Private Sub Form_Load()

'initialize variables:
  ASMsg$ = "12345678901234567890" & vbCrLf
  ComData$ = ""
  ComTime! = 0
  FEND$ = Chr$(192)
  J = 1
  Q = 0
  RPkt$ = ""
  R2Pkt$ = ""
  KeyIn$ = ""
  Pkt$ = ""
  Temp$ = ""
  N = 0
  TXFlag = 0
  TXCnt = 0
  TXTO = 0
  ASFlag = 0

  Form1.Left = (Screen.Width - Form1.Width) / 2           ' center form left-right
  Form1.Top = (Screen.Height - Form1.Height) / 2          ' center form top-bottom
  Text1.BackColor = QBColor(0)                            ' black background
  Text1.ForeColor = QBColor(15)                           ' white letters
  Text1.FontSize = 10                                     ' 10 point font
  Text2.BackColor = QBColor(0)                            ' black background
  Text2.ForeColor = QBColor(15)                           ' white letters
  Text2.FontSize = 10                                     ' 10 point font

  MSComm1.CommPort = 1                                    ' initialize com port 1
  MSComm1.Settings = "19200,N,8,1"                        ' at 19.2 kbps
  MSComm1.RThreshold = 0                                  ' poll only, no interrupts
  MSComm1.InputLen = 0                                    ' read all characters
  MSComm1.PortOpen = True                                 ' open com port
  InDel! = 0.1                                            ' initialize delay at 100 ms

  Randomize                                               ' initialize random number generator

  Show                                                    ' show form
  Text1.Text = "**TX Message Window**" & vbCrLf           ' display TX start up message
  Text1.SelStart = Len(Text1.Text)                        ' put cursor at end of text
  Text2.Text = "**RX Message Window**" & vbCrLf           ' display RX start up message
  Text2.SelStart = Len(Text2.Text)                        ' put cursor at end of text

  Timer1.Interval = 300                                   ' 300 ms timer interval
  Timer1.Enabled = True                                   ' start timer

End Sub
```

```
Private Sub Timer1_Timer()
  If TXFlag = 1 Then                                  ' if TX flag set
    Call DoTX                                         ' send/resend/NAK
  End If
  If MSComm1.InBufferCount > 0 Then                   ' if bytes in input buffer
    Call RxPkt                                        ' call RxPkt
  End If

  If ASFlag = 1 Then                                  ' if AutoSend flag set
    Call ASPkt                                        ' call Autosend
  End If
End Sub

Public Sub RxPkt()
  Call InCom                                          ' InCom will get it
  Call ShowPkt                                        ' ShowPkt will show it
End Sub

Public Sub InCom()
  On Error Resume Next                                ' set up error handler
  ComTime! = Timer                                    ' get current time
  Do Until Abs(Timer - ComTime!) > InDel!             ' get bytes for InDel! interval
    Do While MSComm1.InBufferCount > 0                ' while bytes are in com buffer
      ComData$ = ComData$ & MSComm1.Input             ' put them in ComData$
    Loop
  Loop
End Sub

Public Sub ShowPkt()
  RPkt$ = RPkt$ & ComData$                            ' add ComData$ to end of RPkt$ FIFO
  ComData$ = ""                                       ' clear ComData$ for next time
  Do                                                  ' do until FEND$s gone
    Q = Len(RPkt$)                                    ' Q is RPkt$ packet length
    J = InStr(1, RPkt$, FEND$)                        ' find position of next FEND$
    If (J < 2) Then                                   ' if FEND$ is in the 1st position
      RPkt$ = Right$(RPkt$, (Q - J))                  ' just delete it
    Else                                              ' else
      R2Pkt$ = Left$(RPkt$, (J - 1))                  ' R2Pkt$ what's left of FEND$
      RPkt$ = Right$(RPkt$, (Q - J))                  ' RPkt$ what's right of FEND$
      If R2Pkt$ <> " ACK" Then                        ' if it's not an ACK
        Call LenTrap                                  ' manage textbox memory
        Text2.SelStart = Len(Text2.Text)              ' put cursor at end of text
        Text2.SelText = R2Pkt$                        ' show RX message
        Call SndACK                                   ' send ACK back
        R2Pkt$ = ""                                   ' and clear R2Pkt$ for the next time
      ElseIf R2Pkt$ = " ACK" Then                     ' if it is an ACK
        Call LenTrap                                  ' manage textbox memory
        Text1.SelStart = Len(Text1.Text)              ' put cursor at end of text
        Text1.SelText = " <OK> " & vbCrLf             ' show OK
        TXFlag = 0                                    ' reset TX flag
        TXCnt = 0                                     ' clear TX counter
        TXTO = 0                                      ' clear TX timeout counter
        Pkt$ = ""                                     ' clear TX packet string
        R2Pkt$ = ""                                   ' and clear RPkt$
      End If
    End If
  Loop Until (J = 0)                                  ' done when there are no more FEND$s
End Sub

Private Sub Text1_KeyPress(KeyAscii As Integer)
  If TXFlag = 0 Then                                  ' if not TX cycle
    KeyIn$ = Chr$(KeyAscii)                           ' get KeyIn
    If KeyIn$ = Chr$(8) Then                          ' if it's a backspace from keyboard
      If N > 0 Then                                   ' and character counter > 0
        Pkt$ = Left$(Pkt$, (N - 1))                   ' trim right end of packet
        N = N - 1                                     ' back up character counter
      End If
    ElseIf KeyIn$ = Chr$(13) Then                     ' else if it's a Cr
      Pkt$ = Pkt$ & vbCrLf                            ' add vbCrLf
      ASMsg$ = Pkt$                                   ' update AutoSend message
      Pkt$ = FEND$ & Pkt$ & FEND$                     ' add framing FENDs
      N = 0                                           ' reset N
      TXFlag = 1                                      ' set TX flag
      TXCnt = 0                                       ' clear TX try counter
      TXTO = 0                                        ' clear TX timeout counter
    Else
      Pkt$ = Pkt$ & KeyIn$                            ' else add character to TX message
      N = N + 1                                       ' increment character counter
    End If
    If (N = 23) Then                                  ' if character count 23
      ASMsg$ = Pkt$                                   ' update AutoSend message
```

```
      Pkt$ = FEND$ & Pkt$ & FEND$                       ` add packet framing characters
      N = 0                                             ` reset N
      TXFlag = 1                                        ` set TX flag
      TXCnt = 0                                         ` clear TX try counter
      TXTO = 0                                          ` clear TX timeout counter
    End If
    Call LenTrap                                        ` manage textbox memory
  Else
    KeyAscii = 0                                        ` else don't echo to the screen
  End If
End Sub

Public Sub DoTX()
  If TXTO = 0 Then                                      ` if TX timeout zero
    TXCnt = TXCnt + 1                                   ` increment TX try counter
    If TXCnt = 1 Then                                   ` if TX try count 1
      Call SndPkt                                       ` send packet
      TXTO = 4                                          ` set 0.8 second timeout
    ElseIf (TXCnt > 1) And (TXCnt < 7) Then             ` for try counts 2 through 6
      Call SndPkt                                       ` send packet
      TXTO = 4 + Int(8 * Rnd)                           ` load random TX timeout count
    ElseIf TXCnt >= 7 Then                              ` else if past 6th try
      Call LenTrap                                      ` manage textbox memory
      Text1.SelStart = Len(Text1.Text)                  ` put cursor at end of text
      Text1.SelText = " <NAK>" & vbCrLf                 ` show NAK
      TXFlag = 0                                        ` reset TX flag
      TCnt = 0                                          ` clear TX counter
      TXTO = 0                                          ` clear TX timeout counter
      Pkt$ = ""                                         ` clear TX packet string
      R2Pkt$ = ""                                       ` clear RPkt$
    End If
  Else                                                  ` else if TX timeout counter not 0
    TXTO = TXTO - 1                                     ` decrement it one count
  End If
End Sub

Public Sub SndPkt()
  If Pkt$ <> "" Then                                    ` if Pkt$ not null
    MSComm1.Output = Pkt$                               ` send packet
  End If
End Sub

Public Sub ASPkt()
  If TXFlag = 0 Then                                    ` if TXFlag not set
    Temp$ = ASMsg$                                      ` use Temp$ for local display
    Call LenTrap                                        ` manage textbox memory
    Text1.SelStart = Len(Text1.Text)                    ` put cursor at end of text
    Text1.SelText = Temp$                               ` add message to textbox
    Pkt$ = FEND$ & ASMsg$ & FEND$                       ` add packet framing to message
    TXFlag = 1                                          ` set ACK flag
    TXCnt = 0                                           ` clear TX try counter
    TXTO = 0                                            ` clear TX timeout counter
  End If
End Sub

Public Sub SndACK()
  MSComm1.Output = FEND$ & " ACK" & FEND$               ` send ACK back
End Sub

Public Sub LenTrap()
  If Len(Text1.Text) > 16000 Then                       ` manage textbox memory
    Text1.Text = ""                                     ` clear TX textbox
    Text1.SelStart = Len(Text1.Text)                    ` put cursor at end of text
  End If
  If Len(Text2.Text) > 16000 Then                       ` manage textbox memory
    Text2.Text = ""                                     ` clear RX textbox
    Text2.SelStart = Len(Text2.Text)                    ` put cursor at end of text
  End If
End Sub

Private Sub Form_Unload(Cancel As Integer)
  MSComm1.PortOpen = False                              ` close com port
  End                                                   ` done!
End Sub

Private Sub mnuAutoSnd_Click()
  ASFlag = ASFlag Xor 1                                 ` toggle AutoSend flag
  If ASFlag = 0 Then                                    ` if flag reset
    mnuAutoSnd.Checked = False                          ` uncheck AutoSend
    Text1.ForeColor = QBColor(15)                       ` white characters
```

```
  Else                                                    ` else
    mnuAutoSnd.Checked = True                             ` check AutoSend
    Text1.ForeColor = QBColor(10)                         ` green characters
  End If
End Sub

Private Sub mnuClear_Click()
  Text1.Text = ""                                         ` clear TX textbox
  Text1.SelStart = Len(Text1.Text)                        ` put cursor at end of text
  Text2.Text = ""                                         ` clear RX textbox
  Text2.SelStart = Len(Text2.Text)                        ` put cursor at end of text
End Sub

Private Sub mnuExit_Click()
  MSComm1.PortOpen = False                                ` close com port
  End                                                     ` done!
End Sub
```

6 Revisions and Disclaimers

There are several improvements in the example software in this revision. The RF data rate in both link layer protocol examples has been increased from 1200 to 2000 bps, and the packet retry back off interval in DK200A.ASM has been better randomized. The V110T30C host terminal program now supports multi-packet messages and both host terminal programs provide better Windows efficiency. Component values in Figure 4.2 have been adjusted to match the higher RF data rate.

The information in this design guide is for tutorial purposes only. Any software developed using the information provided in this guide should be thoroughly tested before use. No representation is made that the software techniques and example code documented in this guide will work in any specific application. Please refer to the Virtual Wire® Development Kit Software License and Warranty for additional information.

file: tr_swg19.vp, 2002.08.07