LINX® TECHNOLOGIES WIRELESS MADE SIMPLE®

APPLICATION NOTE AN-00160
Considerations For Sending Data Over a Wireless Link

INTRODUCTION

Linx modules are designed to create a robust wireless link for the transfer of data. Since they are wireless devices, they are subject to external influences that are not present in wired communications. These influences are capable of interrupting and corrupting the data that is being sent by the transmitter, causing the output at the receiver to be incorrect. To correct for this possible interference, it is recommended that a designer implement some sort of noise tolerant protocol. The goal of the protocol is to synchronize communications between the transmitting and receiving ends, identify valid data packets, verify that the data packets are correct, and possibly even correct bad data in a packet.

One common misconception is that a digital address can be sent to distinguish one transmitter from another, even when multiple transmitters are on at the same time. RF is an analog domain so the digital content does not matter. One way to think about it is like a crowded room. If only one person is talking, everyone will be able to hear that person. If everyone talks at once, then it becomes difficult to hear any single person. A protocol can help reduce noise in a multiple transmitter system and help the receiver pick out the valid data in a noisy environment.

Except for the encoder/decoder pair, Linx modules do not place any constraints on the type or format of the data being sent. This freedom allows for more versatility in the types of applications that can use the modules, but it also places the protocol design burden on the customer. This application note is intended to help the engineers design a suitable protocol for their application.

To better understand the requirements for such a protocol, we will first examine all of the potential sources of data corruption that create the need for the protocol. To do this we will develop a general model for the communications channel that the data must go through from the transmitter to the receiver. Using this model, we can account for all of the external and internal influences on the data stream.

COMMUNICATION BASICS

A communications channel is the path that data travels from the transmitter to the receiver and is made up of all of the components required to generate the data stream, encode it, transmit it (including the propagation path), receive it, decode it, and interpret it. Figure 1 shows a generic model of a communications channel.

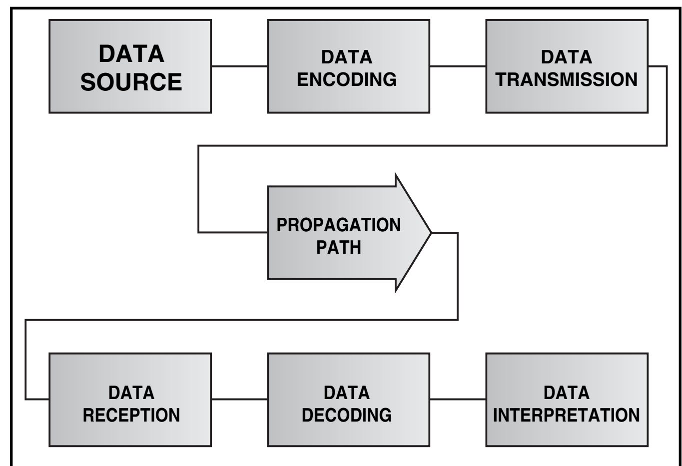

Figure 1: Generic Communications Channel

Data Source

The data source can be anything. It could be an A/D converter reading a temperature, a file on a computer hard disk, or a key press on a keypad. Data corruption at this stage in the communication channel is unlikely and can usually be traced to a bug in the hardware or software of the device.

Data Encoding

The data coming from the data source is generally raw and unprotected. Encoding the data provides a structure, security, and a form of error correction that can ensure data integrity. Data corruption at this stage in the communications channel is unlikely and can usually be traced to a bug in the hardware or software of the device.

Data Transmission

All Linx transmitters are fully tested at the factory, however, external factors such as power supply noise, and improper modulation voltage can corrupt the data stream. A transmitter that is operated as recommended by its data sheet should not contribute to data corruption.

COMMUNICATION BASICS (CONT)

Propagation Path

The propagation path is the path that the radio waves take through free space from the transmitter to the receiver. It is in this stage that data corruption is most likely to occur. Corruption is most commonly a result of either in-band interference or desensitization from unwanted RF sources present in the propagation path. Interference can manifest itself in many ways. Low-level interference will produce noise and hashing on the output and reduce the link's overall range.

Another type of interference can be caused by higher-powered devices such as frequency hopping spread-spectrum devices. Since these devices move rapidly from frequency to frequency they will usually cause short, intense losses of information. Such errors are referred to as bursting errors and will generally be dealt with through protocol.

High-level interference is caused by products sharing the same frequency or from near-band high-power devices. Fortunately, this type of interference is less common than those mentioned previously, but in severe cases it can prevent all useful function of the affected device. It is in these cases that the frequency agility offered by the HP is especially useful.

Although technically it is not interference, multipath is also a factor to be understood. Multipath is a term used to refer to the signal cancellation effects that occur when RF waves arrive at the receiver in different phase relationships. This is particularly a factor in interior environments where objects provide many different reflection paths. Multipath results in lowered signal levels at the receiver and thus shorter useful distances for the link.

All of these effects can cause continuous or periodic data corruption. It must be recognized that many bands are widely used, and the potential for conflict with other unwanted sources of RF is very real. In a wired application, the propagation path is a wire instead of free space and tends to introduce few, if any, errors into the data stream. This fundamental difference between a wired and a wireless link requires the need for some type of data transfer protocol. Later in this application note we will discuss how protocol can help deal with these issues.

Data Reception

All Linx receivers are fully tested at the factory to ensure that they function to all of the specifications set forth in the receiver data guide. There are several conditions, though, that can cause data corruption in the receiver stage. If another transmitter turns on with sufficient power, it can interfere with the desired signal, causing the data output to become noisy. Furthermore, in-band interference and de-sensitization can cause the receiver to corrupt the data stream. And, lastly, the receiver can corrupt the data stream if the data source violates the minimum or maximum baud rate specification of the receiver.

The method of signal modulation also has an impact on the receiver's ability to capture a signal in the presence of interference. FM/FSK systems are generally superior than AM/OOK systems in this respect.

Data Decoding

The received data is decrypted and checked for errors. If implemented, error correction is performed to correct any detected mistakes in the data. Data corruption at this stage in the communications channel is unlikely and can usually be traced to a hardware or software bug.

Data Interpretation

Data interpretation is usually accomplished in software and involves doing something useful with the information that was received. Data corruption at this stage channel is unlikely and can usually be traced to a hardware or software bug.

WHAT IS A PROTOCOL?

Every time two people communicate, there is always a potential for misunderstanding. In some cases, the risk is acceptable. Other times, such as during negotiations, misunderstanding can cause severe repercussions (such as war). When the accuracy of communications is vital, rules for communication are established prior to negotiations to ensure that both sides are "speaking the same language". These rules are referred to as protocols. Since the communications between the transmitter and receiver can be corrupted, a protocol is absolutely required to ensure that the receiver can "understand" the data from the transmitter and also determine if the data that was received is correct or if there are any errors.

PROTOCOL DESIGN CONSIDERATIONS

Packetization

A protocol will generally cut the main data into smaller pieces that are easier to handle. It will then wrap the data with special information needed by the protocol at the reception end to put the data back together. This process is called packetization. At the reception end, the protocol strips the extra data from the packet and reassembles the original data. This process is called depacketization.

Minimum Overhead

A wireless data transfer protocol should be efficient. A protocol must add information to the main data, such as packet identification codes, error checking, encryption, etc. The amount of information added should be the minimum amount required to achieve all of the goals of the wireless data transfer.

Reliable

A protocol is said to be reliable if it can separate good data from errant data. Reliability is usually achieved by embedding some form of error detection in the data stream. Parity, checksums, and Cyclic Redundancy Checks (CRC) are all forms of error detection codes.

Robust

A protocol is said to be robust if it can correct errant data. Robustness is usually achieved by embedding forward error correction codes in the data stream. There are several methods of forward error correction, though the more complete the system, the more overhead required.

Secure

A protocol is said to be secure if it can prevent unauthorized access to the data. Since the data is transferred over a wireless link, anyone with a receiver can see the transmission. Encryption is used to make the data useless to anyone without the correct access code.

Unique

A protocol is said to be unique if it can distinguish the correct transmission in the vicinity of several transmitters. This is usually accomplished by placing address bits in each packet, enabling the receiver to tell which packets came from the correct transmitter.

Optimum Radio Performance

A wireless protocol should operate in a way that takes maximum advantage of the transmitter and receiver while staying within the legal requirements for operation.

Selecting the Features You Really Need

The features listed above are the most important features of a good protocol. There is going to be a compromise between the error detection, error correction, encryption and overhead. The stronger you encode your data, the more overhead is going to be required from your processor to do the required calculations. This will add time to the transmissions as the processors encode the data then decode it on the receiving end. The designer should choose just the features that the project really needs and no more. This will strike the balance between protocol strength and product performance.

PACKETIZATION

It is a good idea to structure the data being sent into small packets so that errors can be managed without affecting large amounts of data. Typically a packet would begin with a start sequence so that the receiver can identify the beginning of valid data. This sequence would then be followed by the data and any error detection or correction information. A stop sequence can end the packet so that the receiver knows when to stop accepting information. After receiving the stop sequence, the receiver must see a valid start sequence before accepting any more data. Otherwise the receiver can count the number of bit it receives or time out after not receiving valid data for a certain time.

START CODES AND NOISE

The first thing any protocol must be able to do is identify the difference between noise and valid data. Noise appears as random bytes of information, with no obvious pattern. An ideal noise source has the ability to generate any combination of bytes with the same probability as any other combination of bytes. This property of noise makes it very difficult to find a combination of bytes to signify the start of a valid packet. Fortunately, in the real world, noise is rarely ideal.

Wake-Up Sequence

The transmitter has no way of knowing the state of the receiver, so the protocol must place the receiver into a known state prior to sending data. Failure to do this could cause the receiving processor to miss the first few bits of data potentially corrupting the entire packet. The protocol should use a start sequence that begins with two transitions to pull the receiver out of squelch and into a known state. The order of the transitions, either '010' or '101', is up to the designer.

Noise Filter

Next there needs to be a way for the receiving processor to qualify the start of the packet as valid data and not noise. A good way to implement a noise filter is with bit sampling. The transmitter would send a high marking period and a low marking period of specific lengths determined by the baud rate and processing power at the receiver. The receiving processor will repeatedly sample the bit, as fast as possible, looking for level changes. If no changes are seen within the specified period, then the receiver can accept the data as valid. Using more than two pulses would give a higher probability that the transmission is valid, but would need more overhead from the processor and require a longer transmission length.

The order of the initial wake-up sequence described above will determine the order of the sampled bits. For example, if the wake-up sequence is '010', then the first sampled bit should be a 1. The order of the pulses doesn't matter, just that at least two bits are measured since the probability of random noise producing one bit of the correct length may be fairly good.

The length of the sampled bits will generally depend on the baud rate chosen for the application. If possible, make the sampled bit longer than a bit at the chosen baud rate to prevent the receiver from detecting the start sequence in the middle of the data. Also, some margin should be built into the front and back of the pulse. Some transmitters and receivers can cause pulse stretching or shortening as a result of start-up times and modulation methods, so the radios used should be tested to determine the appropriate margin.

This method must be performed at the bit level, which may not be possible for some applications. It works well for removing random noise, but may pass structured data from other, undesired transmitters, so some method for detecting the correct data should also be used.

Digital Logic Filter

To remove the chance that an undesired data transmission will be verified, a digital logic filter should be implemented. This is a series of bits that are not likely to appear in the transmitted data but must appear in the correct position in order for the packet to be valid. This has the advantage of being able to be implemented at the byte level, so, for example, any valid text transmission must be preceded by one of the ASCII characters. This method is not good for removing random noise.

Combining the two methods described above will help to create a robust start sequence that will help to qualify a valid packet. The designer can combine them in whatever manner is appropriate for the application, though the following order will generally work the best:

[Wake-Up][Noise Filter][Logic Filter][Data]

The receiver can only hold a specific level for a certain amount of time, resulting in the minimum baud rate and maximum time between transition specifications. If the length of time between transmissions exceeds this time, then the data integrity cannot be guaranteed and the start sequence should be resent.

ERROR DETECTION

Error detection is achieved by performing some type of analysis on the data prior to transmission and adding the results of this analysis to the data packet. Then, the same analysis is performed at the reception end and compared to the results embedded in the packet. If the two are different, then the packet is errant. There are three common types of error detection: parity check, checksum, and Cyclic Redundancy Check (CRC).

Parity Check

The simplest form of error detection is the parity check. A parity check is accomplished by adding all of the '1's in a string of bits. For Even Parity, if the number is even, then the parity bit is set, otherwise it is not. For Odd Parity, if the number is odd, then the parity bit is set.

Example 1:

In this example, we will calculate the parity of a byte

that is to be transmitted using Even Parity. During transmission, one of the bits gets reversed. The receiver catches this through the parity check at the reception end.

Transmitter:

10101010

There is an even number of '1's, so the parity bit is set and the byte is transmitted as follows:

101010101

Receiver:

001010101

The last bit is not used in the parity calculation since it is the parity bit from the transmitter. Calculating parity on the received byte yields an odd number of '1's and the parity is '0'. Since the receiver's parity calculation does not match the transmitter's calculation, the byte is determined to be errant and is thrown out.

The parity check is the easiest to implement, but is also the most unreliable. It can only catch an odd number of errors in the bit stream. If the number of errors is even, then the parity calculation will incorrectly indicate that the byte is good. Thus, there is 50% chance of the parity check catching an error, which is less than optimal.

Checksum

A checksum is calculated based on a series of bytes by adding the values of the bytes together and truncating the result to the desired bit length. For example:

4	data byte 1
109	data byte 2
65	data byte 3
204	data byte 4
126	8-bit checksum

A checksum will catch many more errors than the parity check. However, by simply transposing data bytes 2 and 3, the data packet becomes errant, but the checksum would provide the same result. The checksum only gives weight to the value of the bytes, not their order. Thus, errors of ordering cannot be caught with the checksum.

Cyclic Redundancy Check

The most popular form of error checking is the cyclic redundancy check (CRC). A CRC is more reliable than the checksum because every bit can individually contribute to the checksum. This makes it much less likely that multiple errors will cancel each other out.

The idea behind the CRC is fairly straightforward while the math is more complicated. Essentially the data is considered a large binary number. This number is divided by another fixed binary number (called the generator) and the remainder is used as the checksum. This checksum is appended to the end of the data and transmitted. While the individual bits do not affect the quotient very much, they do have a large affect on the remainder. This is why division is much more robust than addition.

The receiver can then do one of two things:

1. Remove the checksum from the received data, recalculate the checksum, and then compare the received and calculated versions
2. Calculate the checksum for the entire received message and see if it comes out to 0.

Both of these methods work, but the second option is a little cleaner and faster.

Both, the data stream and the fixed number are generally described in terms of polynomials and are written in the form of aXy, where a is either '1' or '0' and y is the bit position. For example:

10011

would be represented as:

1X4 + 0X3 + 0X2 + 1X1 + 1X0

Since anything times 0 is 0, you can remove all of the exponentials with a coefficient of 0 and simplify the equation.

1X4 + 1X1 + 1X0

This seems inefficient and confusing if the generator polynomial is small, but when dealing with 32-bit polynomials it is actually easier to understand.

The generator polynomial is chosen so that it has the greatest chance of detecting the kinds of errors that are most likely to be found in the real world. This includes one or two bit errors through bursting errors. Some popular generator polynomials are:

16 bits: (16,12,5,0) [X25 standard]

(16,15,2,0) ["CRC-16"]

32 bits:
(32,26,23,22,16,12,11,10,8,7,5,4,2,1,0) [Ethernet]

These polynomials all have a leading '1'. This means that the 16-bit polynomial is actually 17 bits long with the leading '1' (bit 17) and then the other bits listed above set. This leading '1' is called the implicit top bit while the remaining bits are called active bits.

There are two ways of performing this operation. The most basic is to feed the data into a shift register one bit at a time. This is shown in the figure below for an 8-bit generator polynomial.

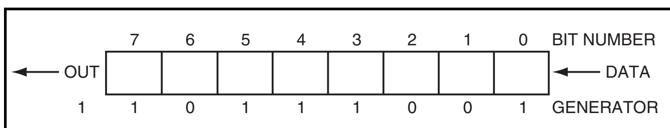

Figure 2: CRC Shift Register

Since the generator is eight bits long, eight '0's are appended to the end of the data. This data is then fed into the shift register 1 bit at a time. Every time a '1' comes out the other end, the generator polynomial is XORd with what is in the shift register. What is left in the shift register at the end of the data is the checksum.

This is a fairly slow way of computing the checksum, though it requires very little overhead. The faster way of computing it is using a look-up table. It turns out that a full byte can be computed at a time. More than that, most of the necessary calculations can be pre-computed and stored in a table. What happens here is that the registers shift out a byte and this byte is used to reference a location in a table of 256 values. The number at this location is then XORd with the values in the registers. This is repeated until all of the data bytes are shifted out. While this method is faster, it requires the table to be generated and stored in memory.

This is a glossed over description of a complicated scheme, but covers the basic idea.

FORWARD ERROR CORRECTION

The goal of error correction is to embed redundant data in the packet at the transmitter end so that the receiver can correct the data if the error detection mechanism indicates that the data are errant.

A Simple Error Correction Algorithm

A very simple method of forward error correction has been developed at Linx that is suitable for many wireless data links. The data are duplicated two times (for a total of 3 copies) in the packet at the transmission end. At the reception end, the first copy of the data in the packet is checked for errors. If there is an error, the two redundant copies of the data in the packet are used to generate one correct version of the data.

The correction is achieved by comparing the bits of each of the three copies of the data. If two or more bits are set, the corrected version has that bit set:

0 0 0 0 1 0 1 1 copy 1 (errant byte)

1 0 1 0 1 0 1 0 copy 2

1 0 1 1 1 0 1 0 copy 3 (errant copy)

----

1 0 1 0 1 0 1 0 corrected byte

Once all of the bytes are corrected, they should be resubmitted to the error checking procedure to verify that the corrected bytes are valid. If they are not, then the data are uncorrectable. Otherwise the data can be used.

Hamming Code

Hamming codes use a modification of the parity check to correct a single errant bit or detect 2 errant bits in a byte and they are used with great success in computer memory applications. In wireless data links, though, the noise tends to be bursting and will corrupt several concurrent bits. Hamming cannot correct these types of errors so are probably not the best choice for use in the 900MHz band where high-power spread spectrum devices can cause frequent interference. It should be good enough for short-range links, especially remote control applications that register a key press. Combined with repeated transmission of the packet, Hamming can make a fairly reliable link.

Implementing Hamming is fairly straightforward. We will look at the case of eight data bits in a byte for this example.

First, all of the bit positions that are powers of two are marked as the parity bit positions. All of the other bit positions are for the data bits. For example, suppose we have the data byte

10100101

Spaces for the parity bits are added in the positions

that are a power of 2.

Position	12	11	10	9	8	7	6	5	4	3	2	1
Bit	1	0	1	0	?	0	1	0	?	1	?	?

The parity is calculated by using modulus arithmetic (XOR logic) on the decimal number for the bit location of all of the data bits that are '1'. So in this example it would be:

3 XOR 6 XOR 10 XOR 12 = 3 = 0011

This gives us the final code word.

Position	12	11	10	9	8	7	6	5	4	3	2	1
Bit	1	0	1	0	0	0	1	0	0	1	1	1

The receiver would then XOR the bit positions of all of the '1's:

1 XOR 2 XOR 3 XOR 6 XOR 10 XOR 12 = 0

Since the result is 0 there are no errors. Now suppose position 6 was switched during transmission, making the word:

Position	12	11	10	9	8	7	6	5	4	3	2	1
Bit	1	0	1	0	0	0	0	0	0	1	1	1

The receiver calculates the parity as:

1 XOR 2 XOR 3 XOR 10 XOR 12 = 6

So position 6 is in error. This bit is flipped and the byte is corrected. The receiver then removes the parity bits and the data is recovered.

If more than 2 data bits are wrong and the byte cannot be corrected and must be discarded. More data and parity bits can be used, but the (12,8) code shown above is a good compromise between code integrity and overhead.

Manchester Encoding

Manchester encoding is a way of combining the clock and the data of a synchronous bit stream into one serial data stream. In this method, the bits are transmitted as a level change in the middle of the data bit. A data '1' is transmitted as a 0 to 1 transition, and a data '0' is transmitted as a '1' to '0' transition. This is essentially an exclusive NOR between the clock and the data as seen in Figure 3.

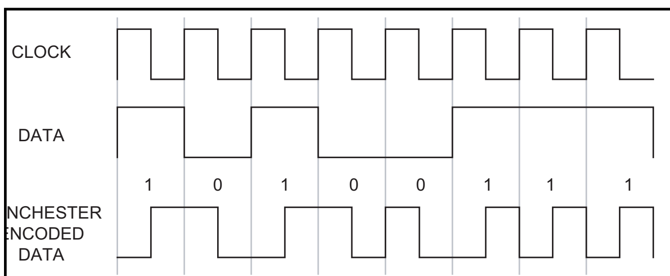

Figure 3: Manchester Encoding

This is advantageous because regardless of what happens at the bit edges, there will always be a level change at the center of each bit, so a series of '1's or '0's will not result in the receiver seeing just a DC level. It also helps the receiver synchronize with the transmitter by having a level change at predictable intervals. The down side is that the frequent level changes require more bandwidth than the original signal. So for a given data rate, the clock can operate at the maximum rate, but the data will be half that rate. An advantage pertaining to Linx AM RF modules is that a 50% duty cycle will draw less current than a higher duty cycle, and that the output power, which is averaged over time, may be increased and still comply with FCC limits.

Reed-Solomon Encoding

Reed-Solomon codes have the capability of identifying and correcting strings of errant bits. They are used in everything from satellite transmission to CD players to compensate for interference and physical deformities (i.e. a scratch on a CD). Turbo codes are a way of encoding the data twice to allow for more error correction power. These codes are very math intensive and are generally overkill for the applications using Linx modules, so they will not be discussed in detail here.

Encryption

Encryption is used to scramble a message in such a way that anyone without the correct key will not be able to decipher the message and understand the content. This is done based on very complex mathematical equations that can only be solved in one way using a number called a key.

The key is known at the transmitter and receiver, but is not transmitted and is kept secret since without the key, the transmission cannot be decrypted. Due to the variety, sophistication, and mathematical complexity of encryption methods, it is often best to utilize encoder and decoder chips, such as those offered by Linx for on-off applications. For data applications, a method suitable to the processing resources should be chosen.

CREATING A SERIAL LINK TO THE MODULES

There are several ways of creating a serial link to Linx RF modules. The modules themselves do not require any programming or do anything to the data, but act like a virtual wire and will simply send along whatever is presented to them. This means that the data source will do all of the protocol generation, set the baud rate, etc. Typically the data source will be one of two things, either a PC or a microcontroller.

Interfacing to a PC will happen one of two ways; through a RS232 serial port or through USB. The RS232 specification uses large positive and negative voltage swings to send data over long distances. These voltage levels can damage the RF modules, so a level converter, such as a MAX232, must be used to reduce the voltages to levels appropriate for digital logic. These converters are common in industry and are made by manufacturers such as Maxim Semiconductor, National Semiconductor, and Sipex.

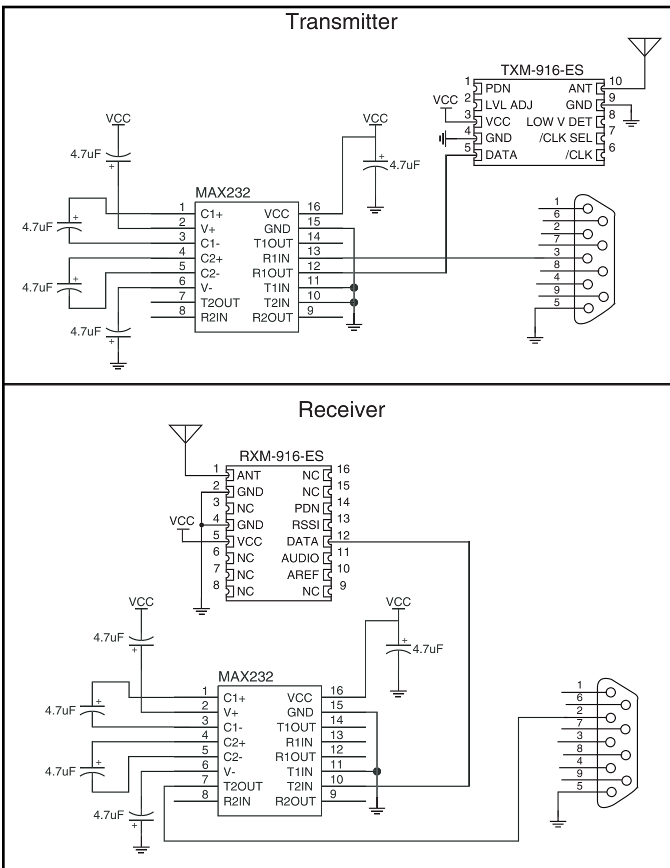

Figure 4: ES Series RF Modules and MAX232

Since most new computers do not have serial ports, especially laptops, Linx has developed a USB module. This module can interface to a PC via a USB cable and the RF module via a PCB trace.

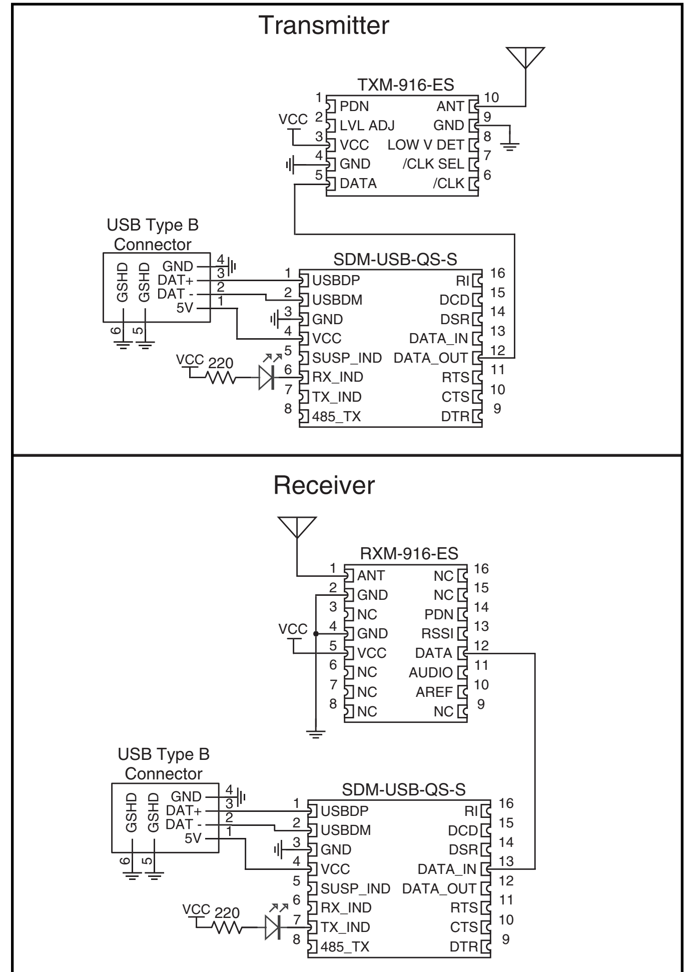

Figure 5: ES Series RF Modules and QS Series USB Module

In either case, application software running on the transmitter PC would generate the packet according to whatever protocol was used and send the data to the transmitter, and the receiving PC would receive the data and decode it according to the same protocol.

Using a microcontroller provides some more challenges, but offers a great deal of flexibility. The microcontroller would create the packet and then send the data to the RF module, then receive it and decode it. Microcontrollers with a built in UART can send data to the RF module via a Print statement (or similar depending on the programming language used) and can receive from it via a Get statement and the UART will perform all of the necessary functions. The actual operation may vary based on the type of microcontroller used.

Controllers without a UART can do what is called "Bit Banging" to an output line. This is when an output line is toggled for a specific time to send a bit at a specific baud rate. For example, to send a '1' at

2400bps, the microcontroller will pull the line high, wait 417uS, and pull the line low. This would be done for each bit in the packet. The wait time can change to vary the baud rate.

One thing that the RF modules cannot do is transmit actual voltage levels since there is no physical connection to a common reference point. A common reason to do this would be to send sensor telemetry. Sensors often represent pressure, temperature, flow, acceleration, etc. as a voltage level. Since the modules will not be able pass this voltage it must be represented by either a frequency (analog) or by a number (digital)

A voltage to frequency converter may be used to send the level information in an analog form or a microcontroller with an Analog-to-Digital Converter (ADC) can be used to convert the level to a digital number. A microcontroller is most commonly used since it can be more accurate and many products need to have some intelligence on the board anyway.

The ADC will convert the analog voltage level to a digital number that can then be packetized and sent to the RF module. Another nice feature about using the controller is that the controller can convert the ADC number into a number that represents what is being measured. Since the sensor, voltage, and ADC value are all typically linear in relation, the processor can use linear interpolation to come up with the temperature or pressure that was measured rather than just send the ADC value. The receiving controller would then be able to display the actual measurement, which would be easier to understand than the ADC value.

PUTTING IT ALL TOGETHER

The designer has a great deal of freedom when creating a protocol, so it can be tailored to meet the requirements and resources of a given application. There is no magic sequence that will work for all applications since each project will have different baud rates, processing speeds, and available overhead, so it is up to the designer to decide what will work best for the project and write it into software. While there are many other methods of encoding and decoding transmissions, those covered here are common and should give the designer a good starting point for the development of their own protocol.