

RF COMMUNICATIONS PRIMER

Written by Aaron Ramsey (aaron@bottle-rocket.org)

1.0 INTRODUCTION

RF is a good way to pass information between robot and robot or robot and PC. In the last few years, many cheap RF boards have become available on the market. These low end RF boards are typically AM modulated with a carrier frequency in the 300-400MHz range and do not contain an encoder/decoder.

I have experience with two RF board manufacturers, Abacom (<http://abacom-tech.com/>) and Ming (http://www.ming-micro.com/rfi_product.htm). I've used the AM-RTD-315 from Abacom. It is a transceiver, which means that one single board is able to receive and transmit, although not at the same time. I've also used the RE-99/TX-99 pair from Ming. In this case, two boards (one transmitter and one receiver) are needed to form a single unit. Of the two, the Abacom boards are much more powerful and reliable. The cost is roughly the same for either, with the Abacoms being slightly more expensive. I will focus on the Abacom boards in this article, but the topics are relevant to any AM modulated low-end RF transmitter/receivers.

Abacom Spec:

Operating Frequency: 315MHz
TX Power: ~1mW (0dBm) with 50 Ohm Load
RX Sensitivity: better than 10 microvolts (-87dBm)
LF Bandwidth: 10kHz square wave
Power Consumption:
TX: less than or equal to 10mA
RX: less than or equal to 3mA

Ming Spec:

RE-99/TX-99
Operating Frequency: 300MHz
Bandwidth: ~2.5kHz square wave
Power Consumption: 1.6mA receiver and transmitter

With a quarter wave antenna, I am able to get over 200 metres with the Abacoms. With a quarter wave antenna, I was only able to get around 50 metres with the Mings. The Abacoms were much more reliable indoors where wiring, etc... can interfere with the RF transmissions.

2.0 SENDING/RECEIVING DATA

As mentioned above, these low end RF transmitter/receivers use AM to transmit data, much like an AM radio station. AM refers to amplitude modulation, which is the method by which data is passed over the RF link. Digital amplitude modulation works by turning on and off the carrier frequency. If the carrier frequency is present, then a '1' is being sent/received. If the carrier frequency is absent, then a '0' is being sent/received. An example of this can be seen in figure 1 below.

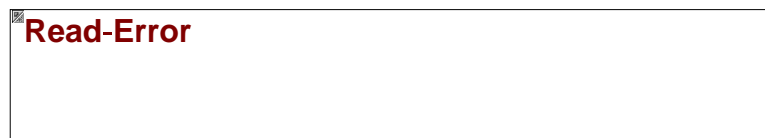


FIGURE 1 – Digital AM Modulation

Connection to a microcontroller is very straight-forward. The input/output of the RF boards can be connected directly to the microcontroller pins. A 10k pulldown on the transmit line and a 10k pullup on the receive line is useful to help fight spurious signals. This can be seen in Figure 2 below.



Figure 2 - RF Board/Microcontroller Connection

In a perfect world, this would be all that it would take to make an RF connection. Sadly, there are a few gotchas along the way. There are two problems that I ran into:

- 1) The serial data being transmitted must be 'balanced' and
- 2) Low end RF equipment doesn't like bursty communications.

2.1 MANCHESTER ENCODING - The art of balancing a serial stream

The first problem with lower end AM transceivers is that they require a balanced signal. In other words, over a short period of time the number of '1's transmitted must be equal to the number of '0's transmitted. The reason for this is that too many 1's or 0's in a row can cause the RF receiver to become 'loaded' at that particular value and it will have a hard time receiving the opposite value that it was 'loaded' at. Simply sending binary or ASCII signals over the RF link obviously won't meet this requirement, and some method must be employed to encode the serial stream. There are several options, ranging from look up tables to ensure that everything is balanced to sophisticated CPU intensive encoding schemes. Somewhere in the middle of that range lies a technique named Manchester Encoding (ME).

In Manchester Encoding, each bit is encoded as a change in level, rather than a single bit. A '1' is encoded as a low followed by a high ('01'), and a '0' is encoded as a high followed by a low ('10'). In this manner, each bit is automatically balanced in the transmission. The advantage to ME is that it is very easy to implement and fast to perform. The disadvantage with this method is that it halves the data rate. In the case of the Abacom, this drops us from a 10kbps data rate to effectively 5 kbps. More sophisticated algorithms are able to balance the stream without halving the data rate, but they require much more computational power. As I generally use relatively small packet lengths to transmit data back and forth from the robot to the robot, this slowing of the datarate is not a large problem. For example, a fixed packet length of 40 bytes would encode to 80 bytes of data. At 10kbps, a 40 byte transmission takes approximately 32ms to transmit. An 80 byte transmission would take double that at 64ms, still reasonable.



Figure 3 - Example of Manchester Encoding

2.2 BURST COMMUNICATIONS

Most low end RF receivers are not immediately ready to receive data when an incoming RF transmission begins. My guess is that the receiver oscillator takes a brief amount of time to start working correctly, but this may be also caused by other factors. In my experiments with the Abacom transceivers, I found that it took almost 30ms before the receiver would reliably detect the data being transmitted.

This leads to two possible solutions. The first solution is to transmit 30ms of meaningless data before the actual data to be transmitted. The second solution is to not transmit bursty traffic. Both solutions work well; the method that you choose in your system depends on how much data traffic you are trying to pass through the RF link.

In my case, I wanted to maximize the amount of data that I could pass. I didn't like the fact that I had to add 30ms of 'header' to the start of every data transmission, which represented a 50% increase in transmission time (based on the 40 byte packet from above). This meant that I had to find a way to keep the RF link active at all times. After some thought, I discovered that it is not that difficult. The details are discussed below in the case study in section 3.

2.3 ERROR DETECTION/CORRECTION

There are many ways to detect bit errors in a serial transmission. Some methods are even able to correct the bit errors without requiring a retransmission. Obviously, the more sophisticated a solution becomes, the more code space and CPU time it requires. Originally I had planned on using a CRC16 (cyclic redundancy check, 16 bits) error detection scheme. This method performs a mathematical calculation on the transmitted data and adds a 16 bit value to the transmission based on the calculations. The receiver performs the same math function on the incoming data and checks the value with the 16 CRC bits transmitted over the link. This test is fairly fast and easy to perform. When bad transmission is detected, the receiver must request a re-transmission.

I found that my RF communications were extremely reliable and I did not often have any bit errors, so I did not end up implementing a CRC16 check. Instead I used the manchester encoding of the data to check for bit errors. This works because of the balanced nature of the manchester encoding. If the routine decoding the manchester data detects that the raw data is no longer balanced, an error flag is raised and the receiver then signals the transmitter to resend the data.

Obviously a noisy link could quickly become a problem. Too many retransmissions will cause the RF link to be essentially useless. I didn't implement anything to combat this, but there are

some solutions. One is to use an error detection scheme which allows the receiver to recreate the lost data. Another solution is to only allow a certain number of retries before giving up. In this case, the receiver could keep copies of each transmission attempt. After giving up, it would then attempt to recreate its best guess of the correct data.

In all cases, the data being transmitted should not be counted on too much. For instance, a self-destruct signal had better be confirmed before being executed. ;-) In other words, mission critical data should have a robust transmission scheme. Data which is not as important can be dealt with in a simpler manner, as I did.

The fact that you may not be able to trust the communications brings up another point. If you are using a PC to control a robot over a RF link, the robot should have enough brains to be able to keep itself out of trouble in case the RF link is disrupted. The robot should also contain some low level behaviours which will keep it from executing (pun intended) a bad command, such as driving off a cliff.

3.0 CASE STUDY

For my final year engineering degree project, I had decided to build a cooperative robot group which would work together to solve various tasks. The idea was that the task would be complex enough that it would take a minimum of three robots to solve it. Obviously I needed some way to communicate between the robots in order for them to coordinate their movements. I decided on RF communications, as I couldn't ensure line of sight between the robots at all time. Each robot used a PIC 16C77 for a main controller and had a Abacom AM-RTD-315 transceiver.

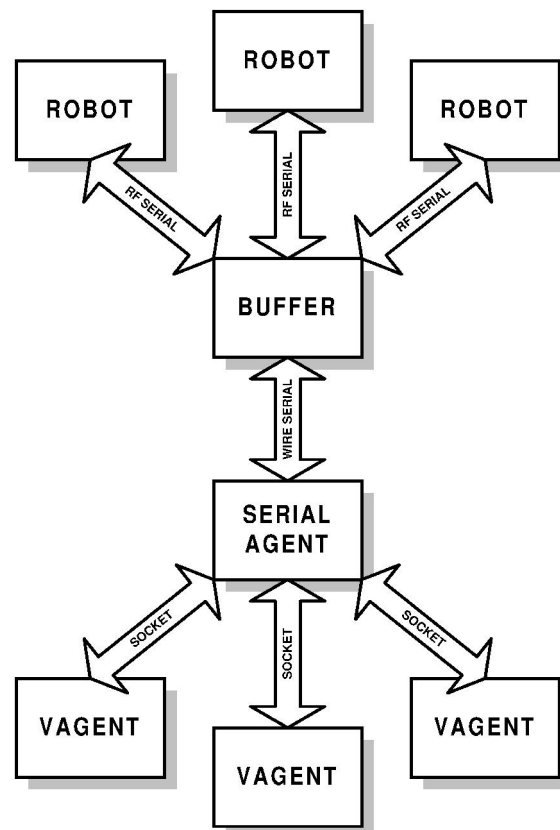


Figure 4 – Three Robots Communicating

The PIC16C77 was found to be fairly limiting when trying to planning complex tasks. It was very adept at providing low-level behaviours, but lacked the necessary ram and rom for more involved tasks. Rather than redesigning the CPU board on the robots, a decision was made to create a virtual controller on the PC for each robot. This virtual agent was labelled vagent.

3.1 BRIEF DESCRIPTION OF THE PC ROLE

I won't deal very much with the details of the PC/robot connection, as the purpose of this article is to deal with the technical details of using an RF transceiver, not actual communication schemes. As the robots were meant to be autonomous and would need to coordinate themselves without some sort of master, the PC routines also needed to follow this setup.

The vagent routines on the PC are running in complete separate threads and cannot communicate directly with each other. In order to communicate between the vagents, they must simulate a RF transmission through the Serial Agent routine. The Serial Agent also takes care of making the RF link transparent between vagent/physical robot pairs. This method was undertaken so that eventually the vagent routines could be moved to the physical robots once the controllers were upgraded.

In figure 4 above, the Buffer block is a Microchip Pic16C73B connected to the serial port of the PC. It takes care of translating the PC raw serial data into the manchester encoded RF serial stream and vica versa.

In essence, the Serial Agent requests data from each robot in turn. The request is passed to the robot by the Buffer block. Each request has a 8 bit address at the start of the packet which lets the robots know which robot needs to respond. Once a robot gets the request, it compiles the requested data/performs the requested action and replies back to the Buffer block. Until the robot replies back, serial communications are blocked to all the other robots. This could cause a problem if one of the robots fails (ie- batteries are dead, robot out of range, etc...). When a request would be sent from the Serial Agent to the failed robot, no reply would every be returned and the RF communications would ground to a halt. In order to combat this, the Buffer has a timeout counter after transmitting to the robot. If the counter times out, the robot is considered to be failing, and the Buffer reports back to the Serial Agent that the communication has failed. The Serial Agent can then move onto the next robot.

3.2 RF Communications using the PIC

The RF communications are written as a serial transmission routine that is interrupt driven in the PIC16C77. Interrupts are used for two reasons. 1) The robot software doesn't need to actively take care of transmitting/receiving; it simply works in the background 2) The incoming RF data can be used to synchronize the data. The routines were coded to work with the ABACOM transceivers, which are designed to work at 10000 bps. As we are merely receiving data on the other end with another PIC, our baud rates do not need to be standard baud rates, but instead merely need to be the same on either end. This program will be written for a 9765.625bps rate, which is close to 9600 (under 2% error), and also very close to the maximum rate of the Abacom transceivers. This baud rate comes around from the fact that we are using a 20MHz clock. That gives a 5000000 instruction rate on a PIC16C77. We will be using timer0 which is a 8 bit timer, which will overflow at a rate of $\text{MHz}/256=19531.25$ which we will then divide in two.

The main CPU, a 16C77 uses pin B0 as the receiving input and B1 as the transmitting output. Pin B0 is used as it has an interrupt that will tell us when an input signal has begun to arrive. This program uses timer0 (RTCC) for timing the RX and TX routines.

The abacom receivers take about 30ms to 'turn on' before they are able to receive the

transmission. This means that we would need to tx meaningless data for 30 ms or so before we could start transmitting the real data. This is because we are using bursts of data rather than a continuous stream of data. This is obviously not acceptable, as it drastically reduces the number of bits per second that we can achieve. In this case, with a 9765.625bps rate, the 30ms adds almost 100% overhead to our data transmission. To compensate for this, we transmit a stream of 1's and 0's whenever the channel is not being used for data. The line is kept active by whoever received the last packet of data. The PC transceiver keeps the line active initially, before communication with any of the robots. It then will send data to one of the robots at some point. After receiving the data, that robot is then given the task of keeping the line active until it returns data to the PC. The main transceiver then takes over until the next packet of data to the next robot... and on and on it goes.

The start bit is 4 high bits, and is easily distinguished from the high/lows that keep the line active. The receiver will be interrupted on every low to high change on the RX_PIN line (pinB0). The ISR will first check a flag to see if we are already receiving or sending data. If we are, it just returns. If not, it loads the timer with a 253. This causes the timer0 interrupt to interrupt in 258 (256+3) clocks from then. This places the timer0 in the approximate middle of the next possible bit... This synchronizes the receiver with the transmitter constantly. In this manner, we never need to purposely resynchronize the two during a transmission.

In the timer0 interrupt service routine, we take care of the transmitting and receiving functions. The transmitter is easy to understand.. It first sends the 4 bit start header, then it simply takes the manchester encoded data and shifts it out the TX_PIN. When it has transmitted one byte, it encodes the next byte, and tx's it too, etc., until it has sent all the data in the buffer. It then sends the 4 bit stop bit (4 lows). The receiver side of the ISR first watches for the start header bits. If it receives 4 highs in a row, it have received a valid start bit. It then brings in the 16 bits of data, decodes it and stores it in the receive buffer. This continues until the total number of bytes that constitutes our packet. (40 bytes) have been read in. No start bits/stop bits between bytes are needed to resynchronize our clocks.

3.3 EASY MANCHESTER ENCODING

The actual implementation of the Manchester Encoding algorithm is slightly different from described above and seen in Figure 3. In order to avoid unnecessary bit manipulation, the bits are encoded in a unique way. It is faster to encode alternate bits in each byte so that we can process four bits at a time.. Just shift, mask, complement and OR the bits together to get an encoded byte. The odd bits (7,5,3,1) are encoded into the most significant 8 bits of data to return and the even bits (6,4,2,0) are encoded into the least significant 8 bits of data to return. In this implementation, 0x0E (00001110) encodes into 0x5AA6 (0101101010100110) instead of the standard 0x55A9 (0101010110101001).

```
short receive_error=0;

uint16 man_encode(uint8 unenc)
    char odd_byte,even_byte;

    odd_byte=(unenc&0xAA) | (~unenc&0xAA)>>1;
    even_byte=(unenc&0x55) | (~unenc&0x55)<<1;
    return ((long)odd_byte<<8) | even_byte;
}

uint16 man_decode(uint8 enc) {
    uint8 odd_byte,even_byte;

    odd_byte=(int)(enc>>8);
    if((odd_byte&0xAA)^((~odd_byte&0x55)<<1)){
        receive_error=1;
    }
}
```

```

        return(0);
    } else odd_byte&=0xAA;

    even_byte=(int)enc;
    if((even_byte&0x55)^((~even_byte&0xAA)>>1)) {
        receive_error=1;
        return(0);
    } else even_byte&=0x55;

    receive_error=0;
    return(odd_byte|even_byte);
}

```

Code Listing 1 - Manchester Encoding and Decoding Routines

This method of encoding saves a couple of instructions and some time. It has the same effect as the traditional manchester encoding, so there is no downside.

3.4 FULL CODE LISTING

The full code demonstrating the RF transmitter/receiver is almost 500 lines long, so it is a little long to list in this article. It is available at this link [<http://www.ottawarobotics.org/articles/rf/abacom.c>] though. The code is written for the CCS compiler (<http://www.ccsinfo.com>) which is a very nice hobbyist C compiler for the Microchip PICs. The code should be reasonably easy to convert to other compilers or processors. Recompiling for a different PIC with the CCS compiler is as simple as changing the `#include <pic16C77.h>` line to including the new processor header file instead. The biggest trick with moving the code to a different compiler or processor architecture would be to convert the interrupt routines over. The CCS compiler makes dealing with interrupts very easy and it automatically creates an interrupt jump table to the various interrupt procedures. The code is well commented though, so it should be easy to read through and re-use.

4.0 CONCLUSION

Well, that wraps up this article. In summary, there are two tricks to working with these AM transmitter/receiver pairs. The first trick is that the serial data must be balanced. Manchester Encoding is the easiest way to overcome this, but it has a penalty of cutting the bit rate in half. The second trick is that many RF receivers require a certain amount of startup time before reliably detecting data. This can be overcome either by transmitting a short burst of meaningless data before the actual real data, or by keeping the RF receivers active all the time by transmitting a '10' pattern (or similar) when not actually transmitting data.

This is only the beginning when dealing with RF communications. I didn't cover many details such as more in-depth error detection/correction (parity checking, CRC16, etc.), RF link control negotiation (where any robot can initiate communications, rather than having a round-robin scheme), as well as many other topics. I'll leave you to discover these on your own.