Data Encoding for Wireless Transmission

Sean McQuay

2 May 2014

The intended overall goal of this project is to develop and implement a reliable and cost effective wireless transmitter/receiver pair that is available to the public. Wireless transmission costs have dropped significantly over the years but the intentions of this project are to further reduce those costs by utilizing a publicly available wireless transmitter/receiver pair and developing a software layer which will work to aggressively remove bit errors and implement an addressing system so that multiple transmitters can be used with a single receiver. Through the use of inexpensive AVR microcontrollers, this software layer will be implemented and the whole transmitter/receiver repackaged on a new board allowing users a serial interface.

Table of Contents

Introduction:

This project aims to implement the Wenshing TWS-BS-3 transmitter and RWS-371 receiver pair in wireless communication. The transmitter supports data speeds of up to 8000bps but the receiver can only sustain a 4800bps connection according to the datasheets (however it was noted during initial testing that speeds above 1200bps created various timing errors in the duration of bit width). This pair operates on a carrier frequency of 433.92MHz with a 1MHz bandwidth. The pair uses amplitude shift keying (ASK) as its mode of modulation. As this is an extremely low-cost transmitter/receiver pair, there is no onboard hardware to resolve bit errors. It is therefore the project's goal to implement this bit error correction in a software layer. AVR microcontrollers will be used to "buffer" the input and output of the pair. By adding this pair of microcontrollers and repackaging the transmitter/receiver pair with the new onboard hardware, the cost of this transmitter/receiver pair could still remain under $10USD. With an ideal maximum distance of 150m and a transmission speed of 1200bps, this pair will serve as an extremely useful item for amateur hobbyists. A secondary goal of the project will be to implement an addressing scheme for the transmitters. An addressing scheme will be implemented along with all necessary software modifications to minimize simultaneous transmission interference and ensure data is read from the correct transmitter.

Detailed Design:

As this project involves data encoding techniques, there are two portions associated with the project and they can therefore be discussed separately even though there are only minor variants between the transmitter and receiver. The encoding technique used and hardware required for the transmitter will first be discussed and following that, the decoding algorithm and hardware required on the receiver will be discussed.

At the heart of the transmitter is the TWS-BS-3 transmitter module. This transmitter operates with amplitude shift keying (ASK) and has no other guards to prevent data corruption. It simply accepts asynchronous serial data (0-5V) and modulates it with a 433.92Mhz signal [4]. The transmitter accepts voltage ranges from 3-12V and broadcasts with a maximum power of 32mW. The transmitter is currently powered by a regulated 5V source. The data is read and encoded by an Atmel ATTiny2313. The AVR will accept data stream inputs in the form of asynchronous serial data and will then encode the data and append all necessary packets to the data to be transmitted and output it to the transmitter module. As a proof of concept, data is currently generated internally and the series of ASCII characters 1-10 are used as the transmitted data. The board also includes a jumper that allows the user to change between transmitting raw data and encoded data to demonstrate the benefits of encoded data.

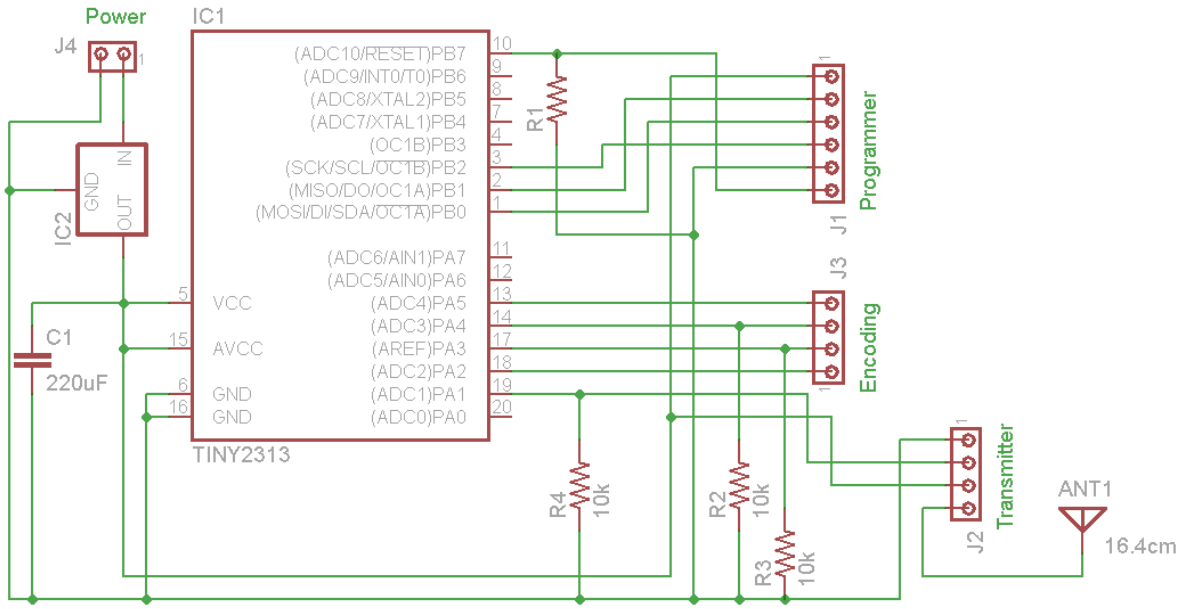

Figure 1: Transmitter Schematic

As it can be seen from *figure 1*, the circuit requires a minimum number of additional components to implement the data encoding. The circuit provides headers for power and in-circuit programmers. It can also be seen that an encoding jumper is present.

Data encoding is achieved through a forward error correction algorithm based on code words and calculating the hamming distance between the received packet and the code word [9]. As the quality of the transmitter/receiver pair is quite low, code words with a hamming distance of eight were selected. This allows the receiver to potentially correct three bit errors in every byte of data received. To achieve a hamming distance of eight, the code words 0x00 and 0xFF were selected. The AVR begins by generating the ASCII character that it will transmit. It then

passes this character to an encoding function along with the address of the current transmitter. The encoding function then parses the address byte and data byte bit by bit and updates an array with the corresponding code words. A digital zero is encoded as 0x00 and a digital one is encoded as 0xFF. Once all encoded data to be transmitted has been written to the array the function then writes two bytes of 0xAA to the transmitter. These two bytes serve as a "synchronization" period. The receiver can easily be affected by noise and adjusts its amplitude to this noise [2]. These two bytes of alternating ones and zeroes serve to return the receiver to a state ready to accept coherent data. The function then transmits the current transmitter address. This address byte is unencoded and can be susceptible to bit corruption but is required to synchronize the receiver and notify it that it should begin to record the received data. The transmitter then is fed 16 additional bytes of data. Eight bytes represent the address and eight represent the data byte.

The user may also select to transmit the "raw" data via the external jumper. In this mode, the transmitter once again sends two initialization bytes of 0xAA to equalize the receiver. The transmitter then sends the notification address byte and finally then transmits the two unencoded bytes containing the address and the data respectively.

```
#define Tx1          0x81
#define Tx2          0x7E

void TxChar_Raw(unsigned char address, unsigned char data);
void TxChar_Encoded(unsigned char address, unsigned char data);

int main(void)
{
	DDRD |= 0b00100110;
	PORTD |= 0b00100101;

	//UART_init(51);            //1200bps
	UART_init(155);             //400bps
	//UART_init(311);           //200bps

	while(1)
	{
		while(PIND & 0x10)  //Encoded data jumper setting
		{
			for(int i = 0; i <= 9; i++)
			{
				TxChar_Encoded(Tx1, i + '0');
				_delay_ms(500);
			}
		}
		while(PIND & 0x08)  //Raw data jumper setting
		{
			for(int i = 0; i <= 9; i++)
			{
				TxChar_Raw(Tx1, i + '0');
				_delay_ms(500);
			}
		}
	}
}


void TxChar_Raw(unsigned char address, unsigned char data)
{
	UART_PutChar(0xAA);
	UART_PutChar(0xAA);
	UART_PutChar(address);
	UART_PutChar(address);
	UART_PutChar(data);
	return;
}
```

```
void TxChar_Encoded(unsigned char address, unsigned char data)
{
	unsigned char dataCodes[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned char tmp = 0;

	for(int i = 0; i <= 7; i++)
	{
		tmp = ((address >> i) & 0x01);

		if(tmp == 0x00)
		{
			dataCodes[i] = 0x00;
		}
		else if(tmp == 0x01)
		{
			dataCodes[i] = 0xFF;
		}
	}

	for(int i = 0; i <= 7; i++)
	{
		tmp = ((data >> i) & 0x01);

		if(tmp == 0x00)
		{
			dataCodes[i+8] = 0x00;
		}
		else if(tmp == 0x01)
		{
			dataCodes[i+8] = 0xFF;
		}
	}
	UART_PutChar(0xAA);
	UART_PutChar(0xAA);
	UART_PutChar(address);
	UART_PutChar(dataCodes[0]);
	UART_PutChar(dataCodes[1]);
	UART_PutChar(dataCodes[2]);
	UART_PutChar(dataCodes[3]);
	UART_PutChar(dataCodes[4]);
	UART_PutChar(dataCodes[5]);
	UART_PutChar(dataCodes[6]);
	UART_PutChar(dataCodes[7]);
	UART_PutChar(dataCodes[8]);
	UART_PutChar(dataCodes[9]);
	UART_PutChar(dataCodes[10]);
	UART_PutChar(dataCodes[11]);
	UART_PutChar(dataCodes[12]);
	UART_PutChar(dataCodes[13]);
	UART_PutChar(dataCodes[14]);
	UART_PutChar(dataCodes[15]);
}
```

Figure 2: Transmitter Source Code

The receiver board has similar hardware to the transmitter. The receiver board uses an RWS-371 receiver module [5]. The decoding software runs aboard an Atmel ATMega164-P. The receiver board uses the same jumper configuration as the transmitter to select between which protocols are used to receive and decode the packets sent to the receiver. It is user configurable between raw data and encoded data via the selectable jumper. The receiver board also features an I2C header used to interface with an external LCD module used to display the received data, current transmitter address, and any errors that may have occurred during transmission and decoding.

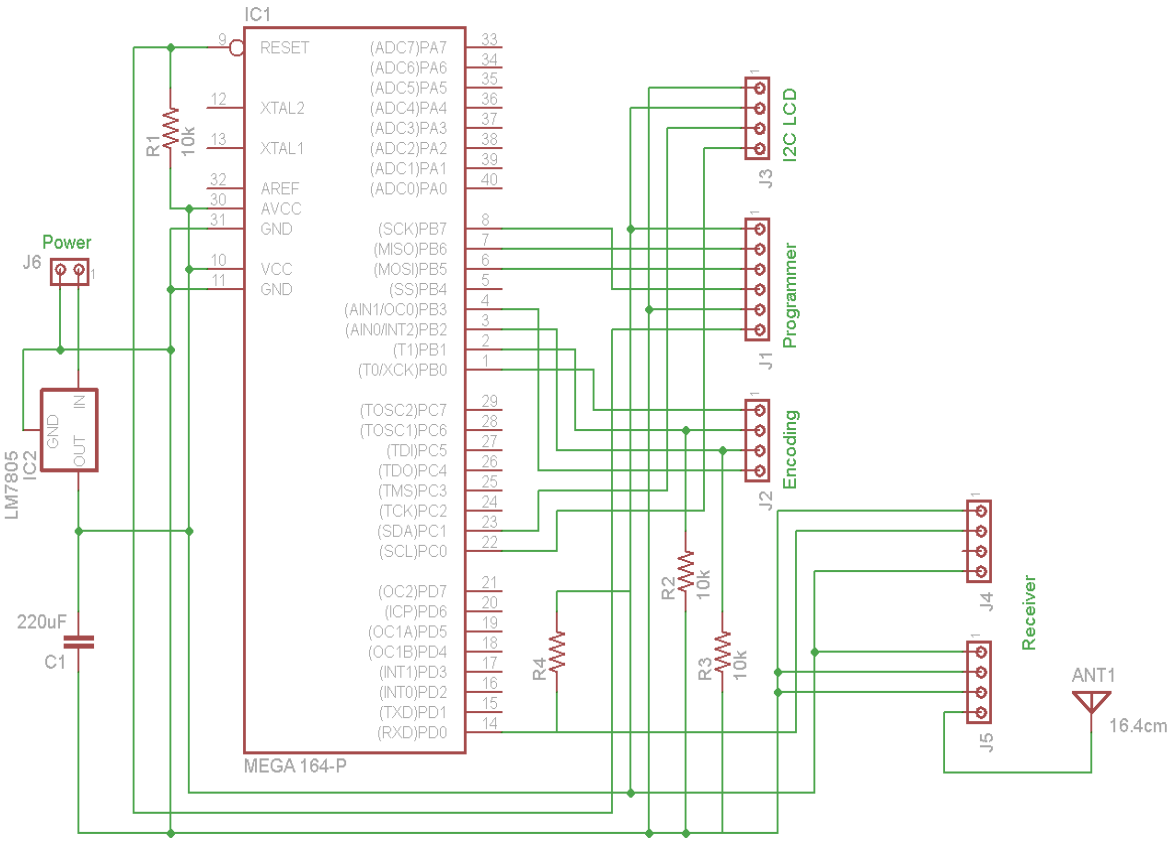

Figure 3: Receiver Schematic

Data decoding on the receiver works in a similar fashion to the reversed encoding scheme. The receiver remains in standby until it then sees a raw synchronization byte that contains the address of one of the known transmitters. Once this occurs, the receiver then pushes the next 16 bytes of data into an array. Decoding then begins starting with the initial eight bytes. The function XORs each byte with the given code words and then proceeds to record the quantity of remaining digital ones [27]. These recorded numbers simply equate to that byte's hamming distance from the given code words. As the code words used in this encoding scheme are 0x00 and 0xFF, the function is capable of recovering three bit errors per byte before the hamming distance leads to an uncertainty condition. The function then verifies which code word produced the smallest hamming distance between itself and the received byte. The address byte is then reconstituted accordingly. The same procedure is then applied to the data byte. As it is assumed that critical data will not be transmitted over this pair, the decoding function attempts to make a best guess when a hamming distance uncertainty condition occurs instead of considering the data too corrupt to reconstitute.

Once all data has been decoded the receiver then cross references the address byte against the table of known transmitter addresses. If the address byte does not match one of the known addresses, the data received will not be output. A potential future feature would allow users to pair the receiver to a new transmitter and update the lookup table with the new address.

As with the transmitter, users may also select to receive raw data from a transmitter. In this case, the receiver once again waits for a raw address byte to

initialize the function. Once this byte has been received, the receiver simply pushes the next two bytes into an array. These bytes are expected to be the raw address and data bytes respectively. The receiver then once again cross references the address byte with the table of know transmitter address. If the data transmission originated from a known transmitter, the data will then be output.

```
#define Tx1          0x81
#define Tx2          0x7E

int main(void)
{
	DDRB |= 0b00001001;
	PORTB |= 0b00001001;

	unsigned char dataCodes[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned char address = 0;
	unsigned char data = 0;
	unsigned char tmp = 0;
	unsigned char count_zero, count_one = 0;

	LCD_init();
	//UART_init(51);           //1200bps
	UART_init(155);            //400bps
	//UART_init(311);          //200bps

	LCD_Write('T', RS);
	LCD_Write('r', RS);
	LCD_Write('a', RS);
	LCD_Write('n', RS);
	LCD_Write('s', RS);
	LCD_Write('m', RS);
	LCD_Write('i', RS);
	LCD_Write('t', RS);
	LCD_Write('t', RS);
	LCD_Write('e', RS);
	LCD_Write('r', RS);
	LCD_Write(':', RS);
	LCD_Shift(Cur_Shift, Right_Cur, 8);

	LCD_Write('D', RS);
	LCD_Write('a', RS);
	LCD_Write('t', RS);
	LCD_Write('a', RS);
	LCD_Write(':', RS);
	LCD_Shift(Cur_Shift, Right_Cur, 2);

	while(1)
```

```
{
        while(PINB & 0x04)   //Encoded data jumper setting
        {
                while(UART_ReadChar() != Tx1)     //Wait for Tx synchronization
                {
                }

                dataCodes[0] = UART_ReadChar();
                dataCodes[1] = UART_ReadChar();
                dataCodes[2] = UART_ReadChar();
                dataCodes[3] = UART_ReadChar();
                dataCodes[4] = UART_ReadChar();
                dataCodes[5] = UART_ReadChar();
                dataCodes[6] = UART_ReadChar();
                dataCodes[7] = UART_ReadChar();
                dataCodes[8] = UART_ReadChar();
                dataCodes[9] = UART_ReadChar();
                dataCodes[10] = UART_ReadChar();
                dataCodes[11] = UART_ReadChar();
                dataCodes[12] = UART_ReadChar();
                dataCodes[13] = UART_ReadChar();
                dataCodes[14] = UART_ReadChar();
                dataCodes[15] = UART_ReadChar();


                for(int i = 0; i <= 7; i++)
                {
                        count_zero = count_one = 0;

                        for(int j = 0; j <= 7; j++)
                        {
                                if((((dataCodes[i] ^ 0x00) >> j) & 0x01) == 1)
                                {
                                        count_zero  ++;
                                }
                                if((((dataCodes[i] ^ 0xFF) >> j) & 0x01) == 1)
                                {
                                        count_one++;
                                }
                        }
                        if(count_zero < count_one)
                        {
                                address |= (0x00 << i);
                        }
                        else if(count_one < count_zero)
                        {
                                address |= (0x01 << i);
                        }
                }


                for(int i = 0; i <= 7; i++)
                {
                        count_zero = count_one = 0;

                        for(int j = 0; j <= 7; j++)
                        {
                                if((((dataCodes[i+8] ^ 0x00) >> j) & 0x01))
```

```
                        {
                                count_zero++;
                        }
                        if((((dataCodes[i+8] ^ 0xFF) >> j) & 0x01))
                        {
                                count_one++;
                        }
                }
                if(count_zero < count_one)
                {
                        data |= (0x00 << i);
                }
                else if(count_one < count_zero)
                {
                        data |= (0x01 << i);
                }
        }

        if(address == Tx1 && data >= '0' && data <= '9')
        {
                LCD_Write(data, RS);
                LCD_Shift(Cur_Shift, Left_Cur, 14);
                LCD_Write('0', RS);
                LCD_Write('x', RS);
                LCD_Write('8', RS);
                LCD_Write('1', RS);
                LCD_Shift(Cur_Shift, Right_Cur, 9);

        }
        else if(address == Tx2 && data >= '0' && data <= '9')
        {
                LCD_Write(data, RS);
                LCD_Shift(Cur_Shift, Left_Cur, 14);
                LCD_Write('0', RS);
                LCD_Write('x', RS);
                LCD_Write('7', RS);
                LCD_Write('E', RS);
                LCD_Shift(Cur_Shift, Right_Cur, 9);
        }
        else
        {
                LCD_Write('!', RS);
                LCD_Shift(Cur_Shift, Left_Cur, 14);
                LCD_Write('!', RS);
                LCD_Write(' ', RS);
                LCD_Write(' ', RS);
                LCD_Write(' ', RS);
                LCD_Shift(Cur_Shift, Right_Cur, 9);
        }


        address = data = 0;
}


while(PINB & 0x02)   //Raw data jumper setting
{
        address = data = 0;
```

```
                    while(UART_ReadChar() != Tx1) //Wait for Tx synchronization
                    {
                    }

                    address = UART_ReadChar();
                    data = UART_ReadChar();

                    if(address == Tx1 && data >= '0' && data <= '9')
                    {
                           LCD_Write(data, RS);
                           LCD_Shift(Cur_Shift, Left_Cur, 14);
                           LCD_Write('0', RS);
                           LCD_Write('x', RS);
                           LCD_Write('8', RS);
                           LCD_Write('1', RS);
                           LCD_Shift(Cur_Shift, Right_Cur, 9);

                    }
                    else if(address == Tx2 && data >= '0' && data <= '9')
                    {
                           LCD_Write(data, RS);
                           LCD_Shift(Cur_Shift, Left_Cur, 14);
                           LCD_Write('0', RS);
                           LCD_Write('x', RS);
                           LCD_Write('7', RS);
                           LCD_Write('E', RS);
                           LCD_Shift(Cur_Shift, Right_Cur, 9);
                    }
                    else
                    {
                           LCD_Write('!', RS);
                           LCD_Shift(Cur_Shift, Left_Cur, 14);
                           LCD_Write('!', RS);
                           LCD_Write(' ', RS);
                           LCD_Write(' ', RS);
                           LCD_Write(' ', RS);
                           LCD_Shift(Cur_Shift, Right_Cur, 9);
                    }
             }
       }
}
```

Figure 4: Receiver Source Code

Experimental Results:

Testing of the forward error correction was a fairly straightforward procedure. A number sequence of 0-9 was generated on the transmitter side and transmitted to the receiver. If the receiver has received the correct address and the data is within the valid range then it is printed to the LCD, otherwise an error character is displayed. No distinction was made between a corrupt address or corrupt data and both resulted in generating an error in transmission. Therefore, if the address and data are displayed then the AVR has assumed that the transmission was recovered correctly. Initial testing began using code words with a hamming distance of just three but these proved to not be powerful enough to overcome the errors in transmission. Using code words with a hamming distance of eight resolved the majority of errors. It can be seen that the error correcting code is quite effective at resolving transmission errors. Transmission could be even more effective if the raw synchronization byte was able to be encoded as well but it is required to be transmitted raw to initialize the receiver. While transmitting raw data, it is noted that many times the address or data bytes appear to be corrupted while the encoded data is able to recover the data almost every single time. Under continuous transmission, many of these errors are removed. It seems that when the receiver is subjected to a constant stream of data that the output stabilizes significantly.

References:

[1] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04036314&tag=1

[2] http://winavr.scienceprog.com/example-avr-projects/running-tx433-and-rx433-rf-modules-with-avr-microcontrollers.html

[3] https://www.sparkfun.com/datasheets/RF/KLP_Walkthrough.pdf

[4] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Wireless/General/TWS-BS-3_433.92MHz_ASK_RF_Transmitter_Module_Data_Sheet.pdf

[5] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Wireless/General/RWS-371-6_433.92MHz_ASK_RF_Receiver_Module_Data_Sheet.pdf

[6] http://nmgroup.tsinghua.edu.cn/yang/paper/SAC09-BPR-He.pdf

[7] http://www.google.com/patents/US7876685

[8] http://nms.csail.mit.edu/papers/pv2002.pdf

[9] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5982354

[10] https://www.omnisterra.com/walker/pdfs.talks/cdrtutor.maker.pdf

[11] http://www2.research.att.com/~slee/pubs/nsdi10.pdf

[12] http://www.ntu.edu.sg/home/sfoo/publications/1998/98_iccs1_fmt.pdf

[13] http://tinyurl.com/qgudg2l

[14] http://www.ijera.com/papers/Vol3_issue2/JH3216491654.pdf

[15] http://spectrum.library.concordia.ca/976241/1/MR45509.pdf

[16] http://sipi.usc.edu/~ortega/Papers/icme-2001-jiang.pdf

[17] http://www.sciencedirect.com/science/article/pii/S1389128612003829

[18] http://nms.lcs.mit.edu/papers/fp315-jamieson.pdf

[19] http://arxiv.org/pdf/1004.2542.pdf?origin=publication_detail

[20] http://people.cs.nctu.edu.tw/~yctseng/papers.pub/mobile95-dvb-net-coding-ieee-cl.pdf

[21] http://tinyurl.com/nw63d96

[22] http://tinyurl.com/qeba2tw

[23] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1303586

[24] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6115907

[25] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6026424

[26] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5371733

[27] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5171235

[28] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6492515