Linear Block Code Error Correction

Devin Trejo
devin.trejo@temple.edu

April 6, 2016

1 Summary

We introduce the (6,3) Hamming Linear Block Coding a error detecting and correcting algorithm. For this lab we will parse a string into 3-bit sized messages and encoded into 6-bit codewords. We then transmit our codewords as characters to a Windows client from a PIC32 MCU. To simulate a noisy communication medium our client introduces random errors into our message and returns it back to the server. We show that the Hamming LBC is capable of correcting up to 1-bit in error for each received codeword.

2 Introduction

2.1 Hamming Linear Block Encoding/Decoding Theory

The Hamming linear block code is a forward error detection and correction method where the original message is encoded to include more symbols to add redundancy to the original message. The process allows the receiver of the message to check and attempt to correct errors introduced by a noisy communications channel. In this lab we will use a (6,3) Hamming code generator matrix (G) which we will transform a 3-bit messages ($\vec{m}$) into 6-bit codewords ($\vec{x}$).

$$\vec{x} = \vec{p}G \text{ where,} \tag{1}$$

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

On the receiving side we first take our received codeword ($\vec{r}$) and detect any errors in the received message. We do so by taking our codeword and checking it against a parity matrix (H). Note that our linear block code produces a systematic codeword; meaning our message is separated into k message bits and (n-k) parity bits. [1]

$$\vec{z} = \vec{r}H \text{ where,} \tag{2}$$

$$H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

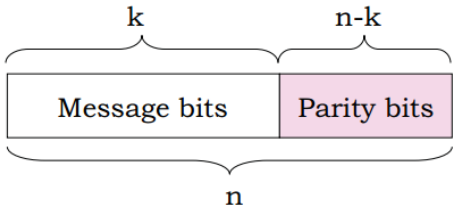

Figure 1: Example of Systematic Encoding Formating [1]

The generator matrix (G) is constructed from an identity matrix of size $k \times k$ concatenated with a matrix A of size $k \times (n-k)$ to the left. If we take matrix A we can create our parity matrix (H) by concatenating it with a identity matrix of size $(n-k) \times (n-k)$ on the bottom.

$$G_{kxn} = \begin{bmatrix} I_{k\times k} & | & A_{k\times(n-k)} \end{bmatrix} \quad H_{nxk} = \begin{bmatrix} A_{k\times(n-k)} \\ I_{(n-k)\times(n-k)} \end{bmatrix}$$

The error detector referenced by equation 2 is said to detect an error when the syndrome ($\vec{z}$) in non-zero. The error correction works by correcting the bit in the received codeword in reference to a minimized **Hamming Distance** between a syndrome in the set of possible valid syndromes ($\vec{z'}$) and the calculated syndrome ($\vec{z}$). We can calculate a set of possible valid syndromes ($\vec{z'}$) by multiplying the parity check matrix (H) by an identity (I); where the identity simulates codewords where one bit is flipped. The result tells us an error in the MSB (bit-6) maps to a syndrome of $\vec{z} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$ and so on.

$$H = I * H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

After we have our codeword corrected ($\vec{r'}$), we can retrieve the original message ($\vec{p_r}$) by using a (3,6) decoding matrix (R). The decoding matrix reverses the steps taken by the generator matrix (G).

$$\vec{p_r} = \vec{r'}R \text{ where,} \tag{3}$$

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Finally we wish to look at the performance of this Hamming Linear Block code encoder/decoder. The correction capacity for any given codeword is given by equation 4 where t denotes the number of errors that are detectable and correctable. In some instances the receiver may not be able to correct a received codeword but it is still detectable. The error detection (e) is given by equation 5.

$$t = \lfloor \frac{d_{min} - 1}{2} \rfloor \tag{4}$$

$$e = d_{min} - 1 \tag{5}$$

2.2 Hamming Linear Block Encoding/Decoding Implementation

For this lab we will transmit the message *"EE is my avocation"* and set up a communication channel between a PIC32 Microchip MCU and Windows desktop over Ethernet. The Windows machine will run a Visual Basic application which will receive a message from the PIC32 server and introduce random errors to simulate errors that may be introduced when transmitting over a noisy communications channel. VB client *Client2 v16B* will introduce at most 1-bit in error within a given message. VB Client *Client3 v16* will introduce up to 10-bits in error. After we will setup our PIC32 MCU to receive a message from the VB client in order to perform error detection and correction using the Hamming linear block scheme.

To use a (6,3) Hamming code requires our individual messages to be 3-bits in size. Therefore, we format our transmission string into messages 3-bits in size. We then will encode our message to produce 6-bit codewords. Since our message is using characters within the standard ASCII table, the MSB in all characters will always be zero. Therefore, ignoring the MSB can now take our 7-bit ASCII character and break it up 3 individual bits. A combination of 3 characters will end up creating a perfect set to complete 7 packets we can send.

$$\begin{array}{rcccccc} 'E' & = & [0100 & 0101] & \rightarrow & [100 & 0101] \\ 'E' & = & [0100 & 0101] & \rightarrow & [100 & 0101] \\ '\text{␣}' & = & [0100 & 0000] & \rightarrow & [100 & 0000] \end{array}$$

100	010	110	001	011	000	000
$\vec{p_0}$	$\vec{p_1}$	$\vec{p_2}$	$\vec{p_3}$	$\vec{p_4}$	$\vec{p_5}$	$\vec{p_6}$

Table 1: 3-Bit Message Composition for First 3 Characters

Our overall message is 18 characters in length. Distributing our 18 characters into 7 packets of length 3-bits produces 42 packets. Those 42 packets will next be encoded using our (6,3) Hamming linear block encoder to produce packets of 6-bits. Finally, we will pad each 6-bit packet with two leading zeros and send it as a 8-bit ASCII character. The final transmitted packet sequence for the first 3 characters are shown in table 2 by applying equation 1.

0010 0110	0001 0111	0011 0001	0000 1101	0001 1010	0010 0110	0000 0000
$\vec{x_0}$	$\vec{x_1}$	$\vec{x_2}$	$\vec{x_3}$	$\vec{x_4}$	$\vec{x_5}$	$\vec{x_6}$

Table 2: 6-Bit Codewords Composition for First 3 Characters

3 Discussion

To begin we need several functions that will allows us to create our As will be seen in order to analyze our experiments, we send the PIC32 MCU server a a return message back. The returned message may have errors randomly distributed throughout as to simulate a noisy communications channel. The server will receive the message and detect and attempt to correct any errors. With the code provided (see Appendix) a uncorrectable error will light up the PIC32's red LED (LED0). A correctable error will flash the PIC32's yellow LED (LED1) as it goes about correcting any errors.

We can use WireShark to analyze the packet transmitted from our server PIC32 MCU to our VB client. We expect the first seven packets to be as demonstrated in table 2. The 42 packets that make up our overall transmission can be seen in highlighted figure 2.

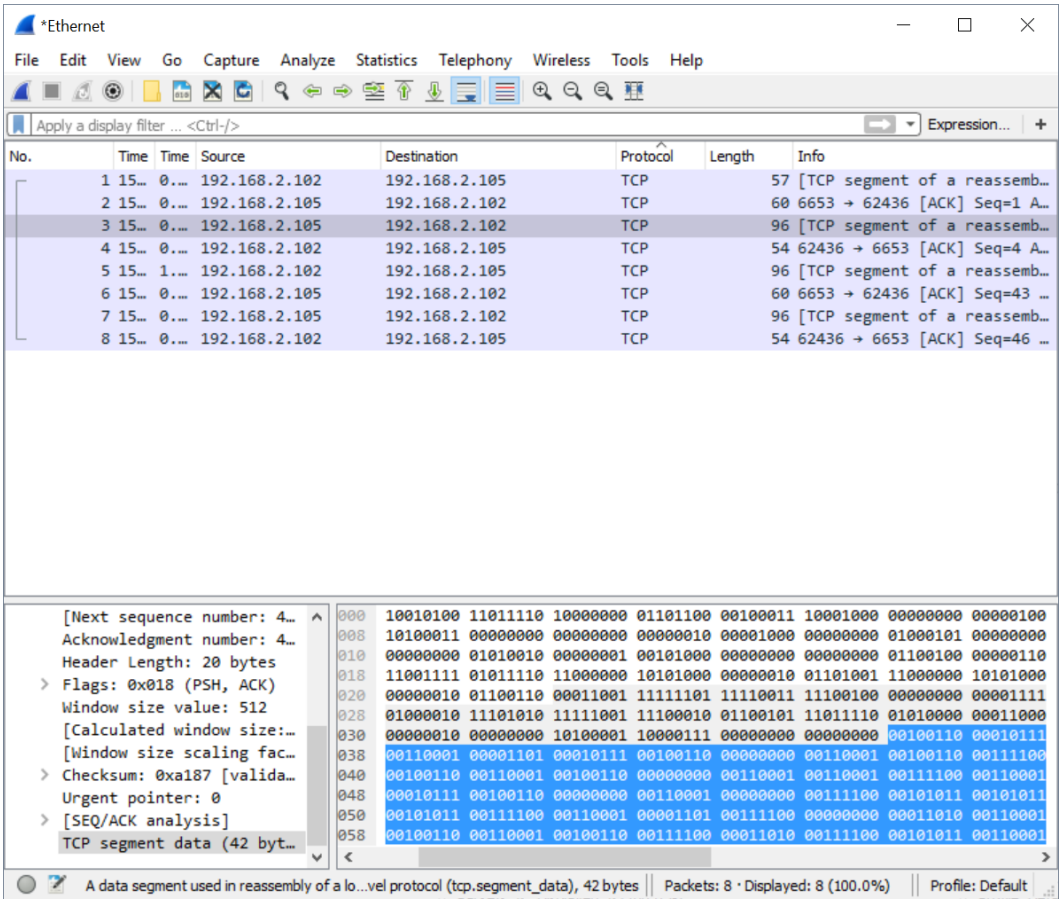

Figure 2: WireShark Capture of 42 Packets Tx PIC32 MCU → Windows Desktop

3.1 Control Case: No Errors Introduced

Next we want to ensure decoding of a message with no errors introduced works as designed. The expected transmitted message from the Windows Desktop with no errors will be the same as that seen when the message was initially transmitted to the Windows Desktop client referenced by figure 2. As can be seen in the transmitted message from the Windows Desktop client, no errors were introduced.

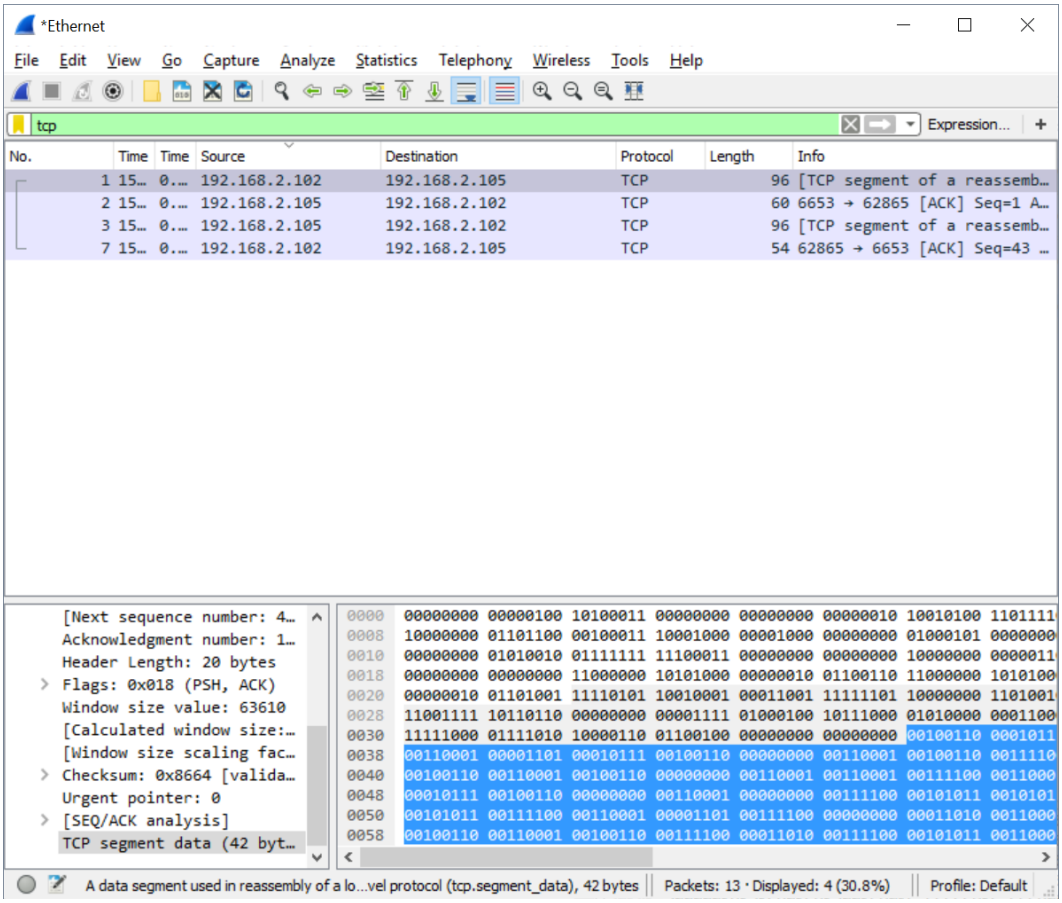

Figure 3: WireShark Capture of 42 Packets Tx Windows Desktop → PIC32 MCU with No Error

The message echoed by the PIC32 server should be the same as that transmitted to it since there were no errors introduced. Comparing figure 3 with figure 4 we can see no discrepancies.

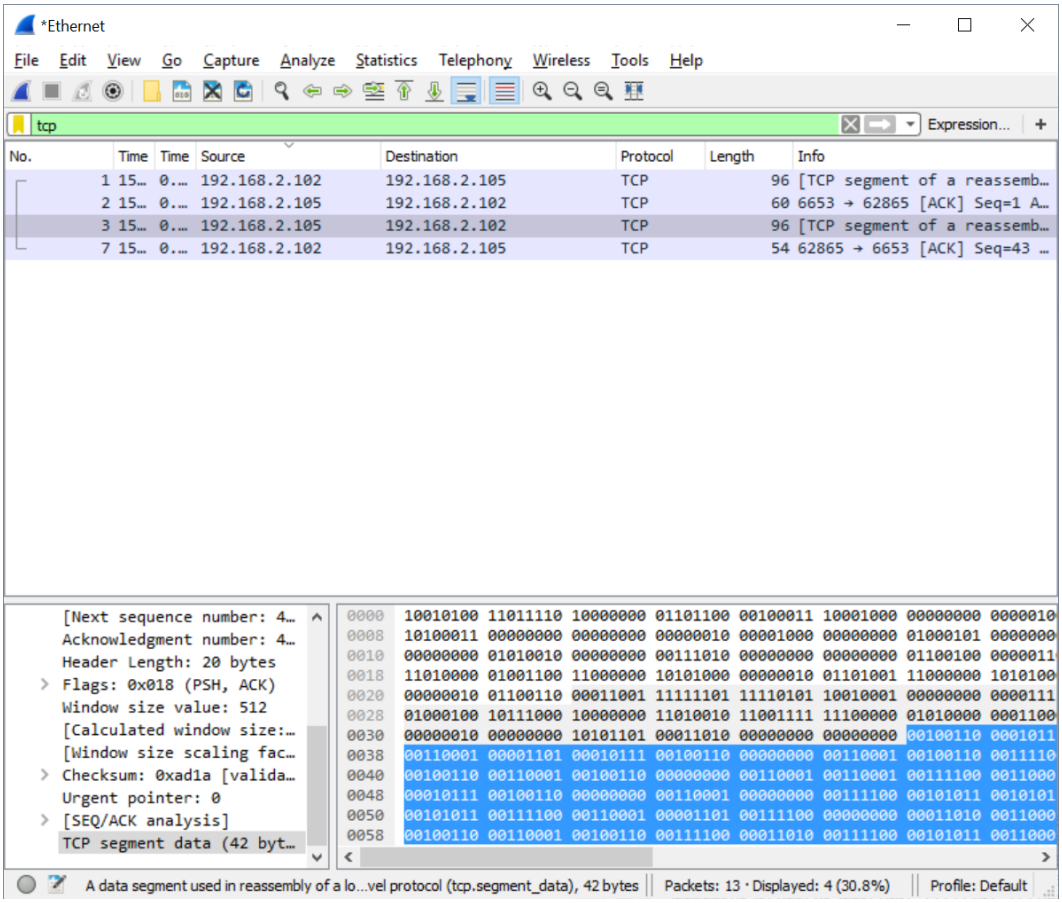

Figure 4: WireShark Capture of 42 Packets Tx PIC32 MCU → Windows Desktop with No Error

3.2 1-Bit Error

Next we show that for one bit in error our Hamming error detector and corrector will be able to detect any introduced errors. The error in this 1-bit error test was introduced into the 48th row of the transmission. A 8-bit sequence of 0x00 was converted to 0x01.

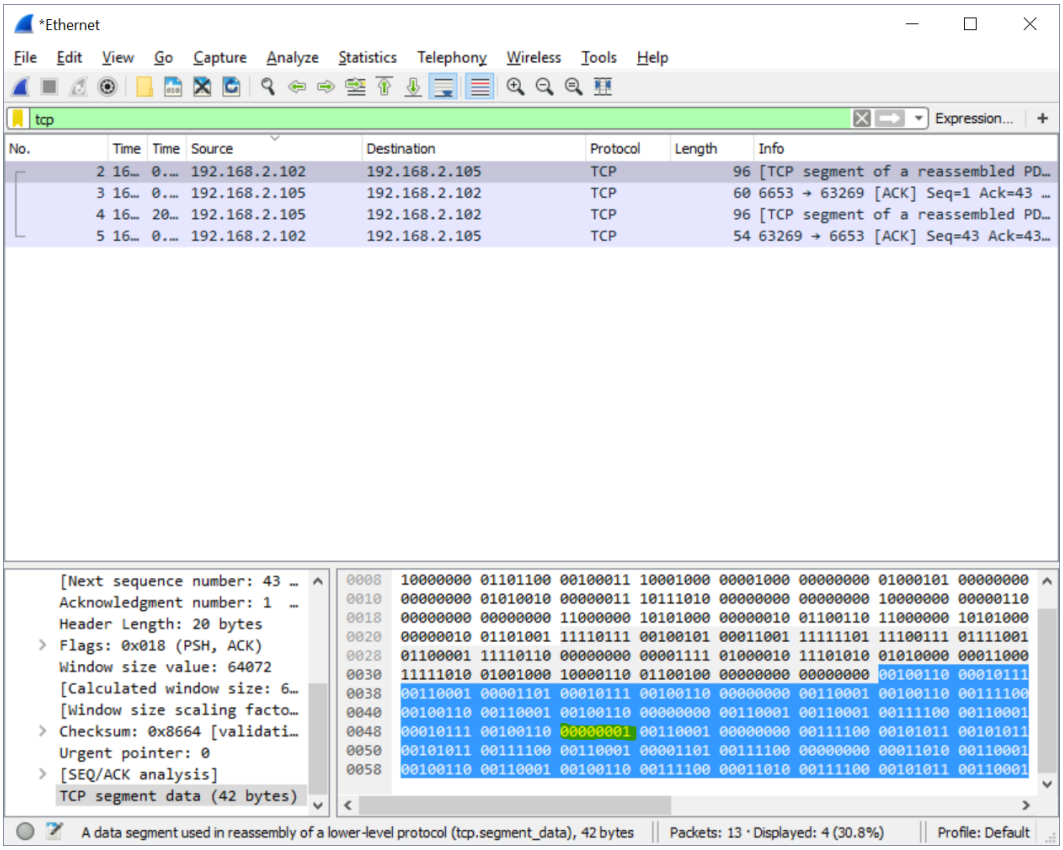

Figure 5: WireShark Capture of 42 Packets Tx Windows Desktop → PIC32 MCU with 1-Bit Error

The error highlighted in figure 5 can be seen corrected in the echoed message from the PIC32. In reality this introduced error effected one of the parity bits so our original message was still intact, but the test shows that LBC can even correct errors in the parity portion of a transmission.

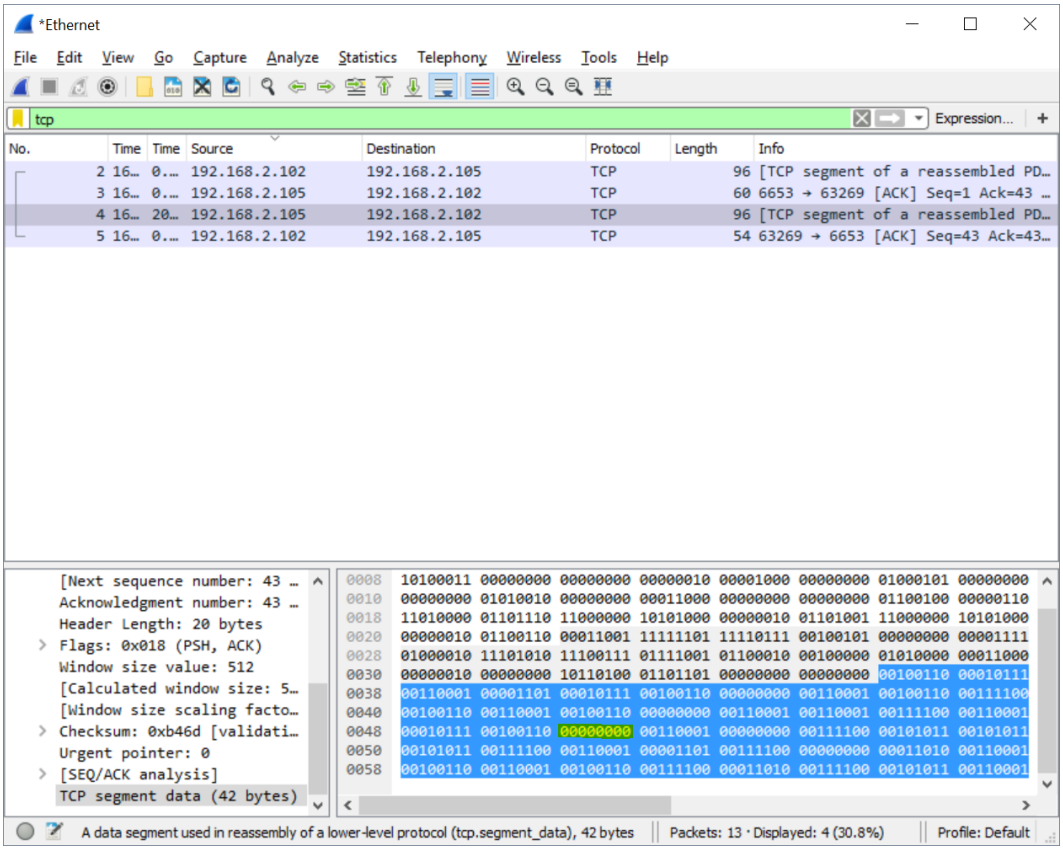

Figure 6: WireShark Capture of 42 Packets Tx PIC32 MCU → Windows Desktop with 1-Bit Error

3.3 Multiple Errors in Transmission

Our last test will show that in extreme cases where more than one bit in error in introduced to our transmission, our Hamming decoder has the capability to correct the error. In the next example we introduce 8 bit in error to the packet. There are two bit errors that occur in the two MSB of the packet. Recall that the two MSB are padded zeros, so these bit errors are ignored.

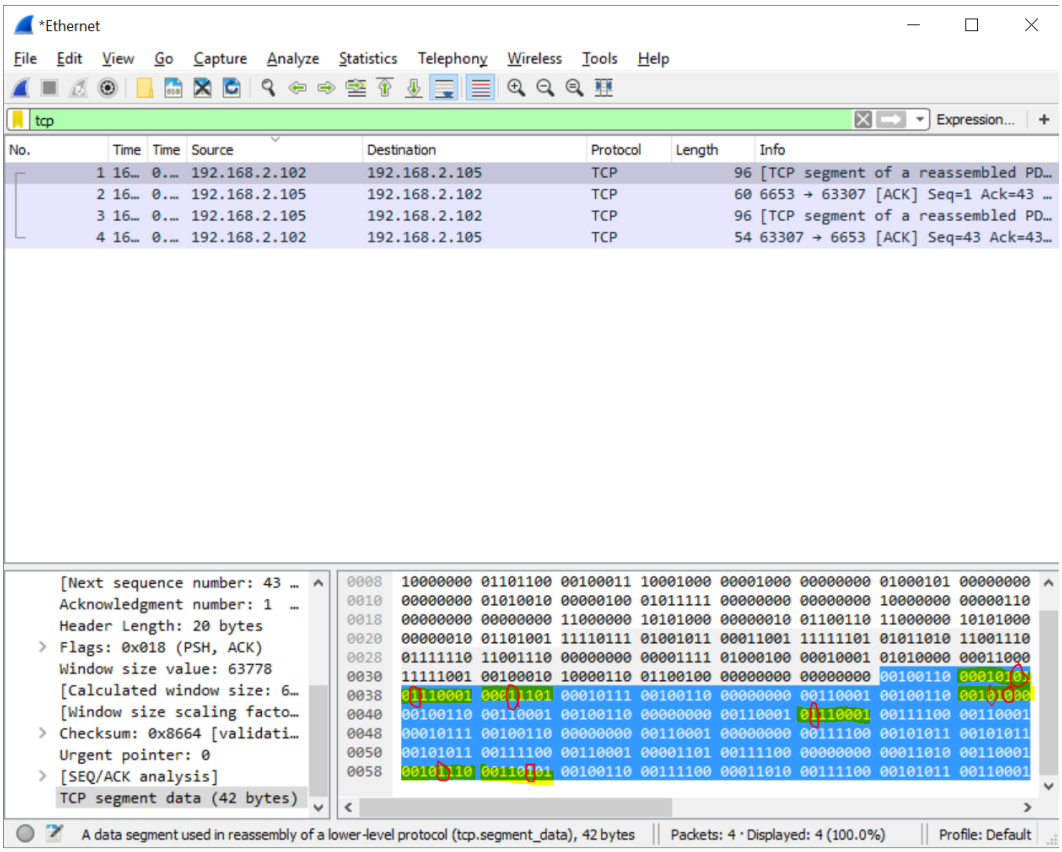

Figure 7: WireShark Capture of 42 Packets Tx Windows Desktop → PIC32 MCU with 8-Bit Errors

The corrected message sent echoed by the server corrects the issues it could. In the second byte, we can see how the second bit from the left was corrected back to a binary 1. Our third highlighted error was also corrected so it incorrect bit was flipped. Our forth byte sequence which contained an error was changed to all zeros. These is behavior coded for when an error is detected but could not be corrected. Since there were more than one bit and error in the message the receiver could not correct the message. In this scenario typically we would have the receiver re-transmit the message.

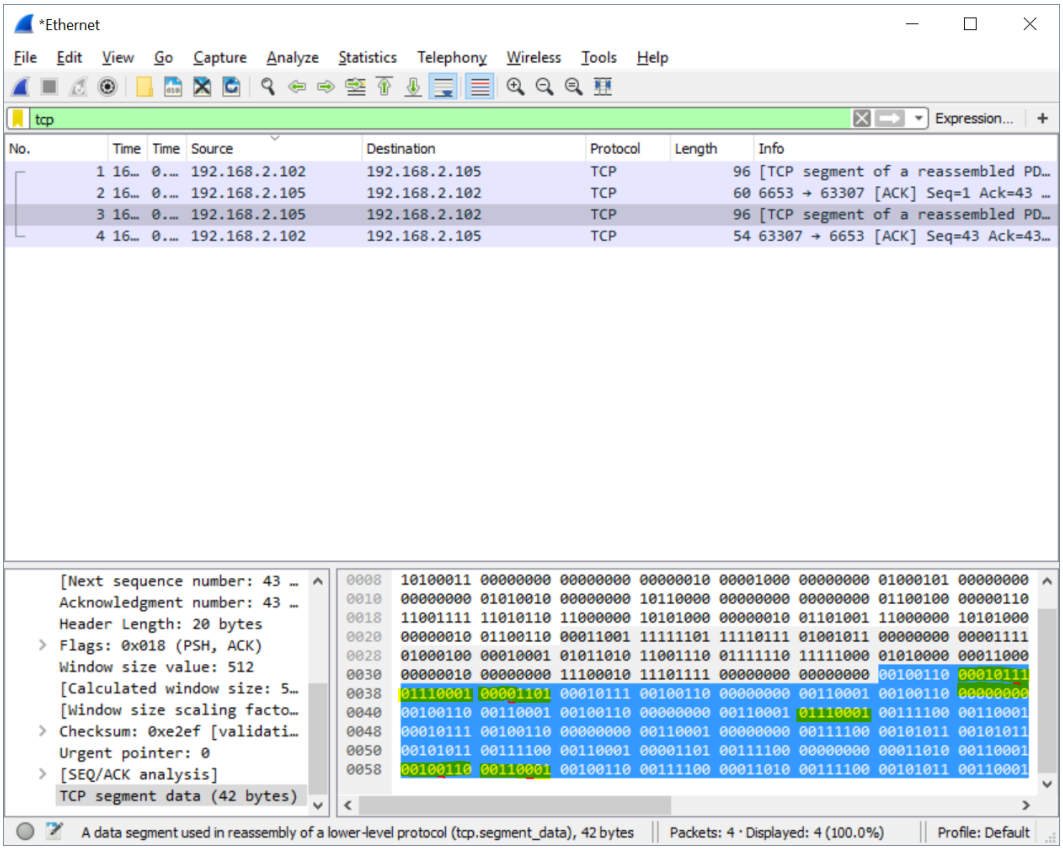

Figure 8: WireShark Capture of 42 Packets Tx PIC32 MCU → Windows Desktop with 8-Bit Errors

As is seen in the two byte sequences that contained errors in the higher two MSB locations, our implementation is not at its peak efficiency. Hypothetically our message size is $k = 3$ and our codeword size is $n = 6$, we can say out **code rate** $= \frac{k}{n} = \frac{6}{3} = \frac{1}{2}$. However, since we pad our codeword with two leading zeros it can be said that our code rate decreases to code rate $= \frac{3}{8}$.

The padding of zeros only causes our data rate to decrease. A way we could improve our Hamming linear block encoder/decoder is by fully utilizing the entire length of our 8-bit packets. It will complicate the decoding process since it would require that we extract bits from other various packets to construct our codeword.

4 Conclusion

We have introduce another form of error detection and correction that has been proven to correct up to one in error. The Hamming linear block code are useful in forwarding error correction to reduce the need for retransmission of data corrupted by the transmission medium. The technique is effective in correcting a message in its use of codewords which convert a message a sequence where the difference between each codeword is at least two bits. By maximizing the between messages we can improve the performance of this error detection and correction scheme.

References

[1] Hari Balakrishnan and George Verghese. Introduction to EECS II: Digital Communication Systems, 2010.

Appendix

Main program source code available on my GitHub:
`https://github.com/dtrejod/myece4532/blob/master/lab4/ECE4532%20PIC32%20BSD%20Server/source/main.c`

Listing 1: Hamming Encoder Function

```
void hammingEncoder(const char *myStr, char *tbfr, int tlen)
{
    int i;
    char messageResized = 0x00;

    for(i=0; i < tlen; i++)
    {
        messageResized = getPacket(myStr, i, PACKETLEN);
        tbfr[i] = getEncodeCodeword(messageResized);
    }
}
```

Listing 2: Hamming Decoder Function

```
void hammingDecoder(char *recieveBuffer, int rlen)
{
    int i;
    char codewordResized = 0x00;
    char receievedMessage;
    char z = 0x00;

    for(i=0; i < rlen; i++)
    {
        recieveBuffer[i] = hammingErrorDetectorCorrector(recieveBuffer[i]);
        receievedMessage = getDecodeCodeword(z);
    }
}
```

Listing 3: Packet Creator (Creates 3-bit Packet Sizes from Characters)

```
char getPacket(const char *myStr, int packetNum, int packetSize)
{
    // We return a char containing packetSize bits
    char bitPacket = 0x00;
```

```
    // We look to see at what bit interval our packet starts on

    int stringBitStart = packetNum*packetSize;
    int charBitStart = 6-(stringBitStart%7);
    // loop tracker
    signed int i, maskPos;

    for(i=0; i < packetSize; i++)
    {
        // We create our own modulus by making sure negative numbers round
        // down to the lower negative number.
        if (charBitStart-i < 0) maskPos = divideDown(charBitStart-i,7);
        else maskPos = (charBitStart-i)/7;
        maskPos = (charBitStart-i)-7*maskPos;
        // We use seven since our char MSB is irrelevant
        bitPacket |= ((myStr[(stringBitStart+i)/7]
                      & (0x01 << maskPos))
                             >> maskPos)
                             << (packetSize-i-1);
    }

    return bitPacket;
}
```

Listing 4: Create codeword from 3-Bit Message

```
// Takes a first PACKETLEN bits in a char and converts it to a 8 bit char
// encoded codeword
char getEncodeCodeword (char message)
{
    // Generator Matrix
    int G[PACKETLEN][CODEWORDLEN] = {{1, 0, 0, 1, 1, 0},
                                    {0, 1, 0, 1, 1, 1},
                                    {0, 0, 1, 1, 0 ,1}};
    int i, j;
    char codeword = 0x00;
    int codewordPos = CODEWORDLEN -1;

    // Matrix math at the bit level.
    for(j=0; j < CODEWORDLEN; j++)
    {
        for(i=0; i < PACKETLEN; i++){
            // Exclusive OR is same as adding two binary bits
            codeword ^= (((message & (0x01 << PACKETLEN-1-i))
                                        >> PACKETLEN-1-i)* G[i][j])
                                        << codewordPos;
        }
```

```
        // Move codeword position
        codewordPos--;
    }
    return codeword;
}
```

Listing 5: Check Message Integrity using Parity Matrix

```
char hammingErrorDetectorCorrector(char codeword)
{
    // ParityMatrix Matrix
    int H[CODEWORDLEN][PACKETLEN] = {{1, 1, 0},
                                    {1, 1, 1},
                                    {1, 0, 1},
                                    {1, 0, 0},
                                    {0, 1, 0},
                                    {0, 0, 1}};

    int i, j;
    char syndrome = 0x00;
    int syndromePos = PACKETLEN-1;

    // Matrix math at the bit level.
    for(j=0; j < PACKETLEN; j++)
    {
        for(i=0; i < CODEWORDLEN; i++){
            // Exclusive OR is same as adding two binary bits
            syndrome ^= (((codeword & (0x01 << CODEWORDLEN-1-i))
                                        >> CODEWORDLEN-1-i)* H[i][j])
                                        << syndromePos;
        }
        // Move codeword position
        syndromePos--;
    }

    // check if syndrome is zero
    if (syndrome != 0x00)
    {
        // if non-zero we say we have an error. We check to see if we can
        // correct the codeword.
        // Set led 1 (red) high
        mPORTDSetBits(BIT_1);
        DelayMsec(100);
        // Bit 5 an error
        if (syndrome == 0b00000110)
        {
            // Reset LED saying we fixed the error
            mPORTDClearBits(BIT_1);
```

```
    DelayMsec(100);
    codeword ^= 0x01 << 5;
}

// Bit 4 an error
else if (syndrome == 0b00000111)
{
    // Reset LED saying we fixed the error
    mPORTDClearBits(BIT_1);
    DelayMsec(100);
    codeword ^= 0x01 << 4;
}

// Bit 3 an error
else if (syndrome == 0b00000101)
{
    // Reset LED saying we fixed the error
    mPORTDClearBits(BIT_1);
    DelayMsec(100);
    codeword ^= 0x01 << 3;
}

// Bit 2 an error
else if (syndrome == 0b00000100)
{
    // Reset LED saying we fixed the error
    mPORTDClearBits(BIT_1);
    DelayMsec(100);
    codeword ^= 0x01 << 2;
}

// Bit 1 an error
else if (syndrome == 0b00000010)
{
    // Reset LED saying we fixed the error
    mPORTDClearBits(BIT_1);
    DelayMsec(100);
    codeword ^= 0x01 << 1;
}

// Bit 0 an error
else if (syndrome == 0b00000001)
{
    // Reset LED saying we fixed the error
    mPORTDClearBits(BIT_1);
    DelayMsec(100);
    codeword ^= 0x01 << 0;
```

```
        }

        // Otherwise there is an uncorrectable error. We set to zero
        // for analysis purposes.
        else{
            mPORTDSetBits(BIT_0);
            codeword = 0x00;
        }

    }
    return codeword;

}
```

Listing 6: Decode Codeword into 3-Bit Message

```
// Takes a code of CODEWORDLEN bits in a char and converts it to a
// PACKETLEN bit ASCII decoded char
char getDecodeCodeword (char codeword)
{

    // Decoder Matrix
    int R[CODEWORDLEN][PACKETLEN] = {{1, 0, 0},
                                    {0, 1, 0},
                                    {0, 0, 1},
                                    {0, 0, 0},
                                    {0, 0, 0},
                                    {0, 0, 0}};
    int i, j;
    char message = 0x00;
    int messagePos = PACKETLEN -1;

    // Matrix math at the bit level.
    for(j=0; j < PACKETLEN; j++)
    {
        for(i=0; i < CODEWORDLEN; i++){
            // Exclusive OR is same as adding two binary bits
            message ^= (((codeword & (0x01 << CODEWORDLEN-1-i))
                                        >> CODEWORDLEN-1-i)* R[i][j])
                                        << messagePos;
        }
        // Move codeword position
        messagePos--;
    }
    return message;
}
```