

Forward Error Correction in Sensor Networks

Jaemin Jeong and Cheng Tien Ee
Department of Electrical Engineering and Computer Science
University of California, Berkeley

May 16, 2003

Abstract

In any network, there are two basic methods to recover erroneous packets. One way is to use Automatic Repeat Request (ARQ), and another is Forward Error Correction (FEC). Since, in sensor networks, power is scarce and is primarily consumed by wireless transmission and reception, we would prefer to use FEC rather than ARQ. In this paper we determine empirically the error characteristics of wireless transmission signals, and from that, we implement and evaluate different types of encoding schemes that are shown to be successful in reducing the error rates.

Key words: sensor networks, forward error correction, ChipCon Radio

1 Introduction

Sensor nodes are physically small, and when fitted with various sensors such as digital thermometers as well as a means of wireless communications, can form a sensing network. These properties allow the sensor networks to be easily deployed in all types of environments, and we expect such networks to become ubiquitous in the near future. However, having a small form factor and being distributed without a connection to any power grid means that the sensors have limited power. Table 1 shows the power consumption due to various types of instructions, as determined in [3].

Table 1: Power Consumption

Instruction type	Energy per cycle (nJ)	Energy per instr (nJ)
idle	1.70	1.70
arithmetic/logic	3.41	3.41
Device	Energy per CPU cycle	Energy per quantum
LED	1.89	1.89 nJ/cycle
RFM send	2.56	2050 nJ/bit
RFM receive	2.44	1950 nJ/bit

From Table 1 it is clear that most of the power is consumed during the transmission and reception of data. By using FEC, we hope to reduce the need to retransmit data packets, thereby reducing the power consumed in the process.

Various encoding schemes, such as Reed-Solomon and LT Codes, are available and can be readily implemented. However, the choice of the encoding scheme depends on the application and the error characteristics of the wireless channel. For sensor networks, most data transmitted are that of readings

taken periodically. For instance, the sensors might be gathering data every 5 minutes on the temperature distribution of the geographical region over which they are scattered, sending just one packet per node. This is in contrast to an Internet network, where the transfer of data can be rapid and frequent as in the case of streaming video. Also, the encoding scheme should be of relatively low complexity, since a typical sensor currently has low processing power (8MHz) and a small memory (8Kbytes flash, 512 bytes DRAM). Since Reed-Solomon and LT codes are relatively complex and require high processing power and storage, they may not be ideal for our application. Furthermore, packets are sent between long intervals, thus encoding schemes that need to work with several packets at a time can result in an increase in latency to an unacceptable level. Thus, depending on the error characteristics, we would prefer using a simpler encoding scheme.

1.1 Background

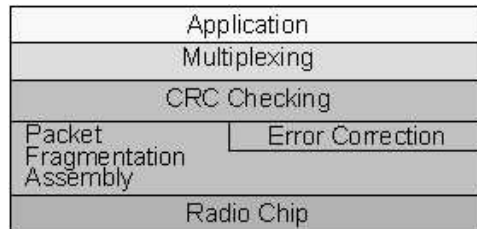


Figure 1: Network stack in wireless sensors

A wireless sensor node communicates with other sensor nodes using the network stack as shown in Figure 1. At the application level, data can be sent in packet form. Since the radio chip at the hardware layer can transmit and receive data byte-by-byte, these packets need to be fragmented before they can be sent and reassembled after being received. The media access control (MAC) layer interfaces the application layer with the radio chip. When a packet is sent by an application, the packet is fragmented into bytes. A special sequence of bytes called the *preamble* is sent before the data bytes so that the receiver can synchronize and detect the beginning of a packet. After receiving a byte, the radio chip bit-encodes the data and transmits them. At the receiver side, the radio chip signals the arrival of bytes after detecting and decoding the data bits. Then, the MAC layer reassembles the packet beginning after the preamble. After that, the MAC layer signals the arrival of a packet to the upper layer.

Data bytes can be optionally encoded after being fragmented with error correction code (ECC) to recover data bits in case of a small number of bit errors. One or more data bytes are mapped onto another sequence of data bytes, which are then passed to the radio chip. At the receiver side, the received data bytes are decoded to give the original data bytes.

Bit modulation converts a bit input into an analog bit sequence [4]. The simplest bit modulation scheme is NRZ (Non-Return to Zero) encoding. NRZ encoding simply generates a waveform with voltage level A for an input bit 0 and voltage level B for the input bit 1. The radio transceiver keeps track of the mean analog signal level to distinguish between the incoming bits. By comparing the received signals with the average level, the transceiver converts the analog waveform to either the binary value 0 or 1. This can pose a problem when there is a long sequence of 0s or 1s, which skews the mean analog signal level, thus reducing the ability of the transceiver to correctly interpret the transmitted digital signal. In NRZ encoding, *stuffing*, where we add the complement of the input bit after itself, needs to be done to avoid this problem. Manchester encoding removes this issue by generating the same number of 0s and 1s for any bit sequence. This encoding scheme encodes a

0 to give a rising edge (A → B) and a 1 to give a falling edge (B → A). An example is shown in Figure 2.

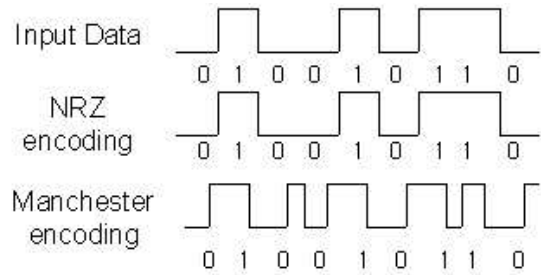


Figure 2: Comparison of NRZ encoding and Manchester encoding

ChipCon radio is the radio transceiver that is currently used in the latest generation of wireless sensor nodes. Compared to RFM radio that was used in earlier platforms, the ChipCon radio has several advanced features. Firstly, wireless sensor nodes with ChipCon radios have greater range than those with the RFM radio. In an outdoor test, ChipCon sensor nodes had a range of about 1000 feet compared to about 400 feet for the RFM sensor nodes [5]. Secondly, the data bits are modulated using FSK (Frequency Shift Keying) making the data transfers more resilient to noise [6]. Lastly, the ChipCon radio supports Manchester encoding at the hardware level, allowing the data bits to be transmitted without the need for explicit, software-level DC balancing.

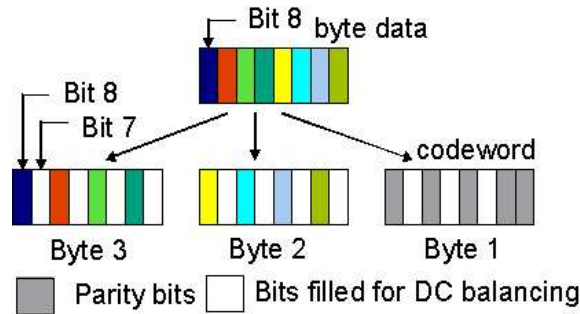


Figure 3: Bit encoding of RFM radio

The error correction code for the RFM radio is unsuitable for use in the ChipCon radio because it unnecessarily stuffs additional bits for DC balancing. Figure 3 shows how a byte is encoded to give a three-byte codeword. For example, if the bit 8 of data byte is 1, then the codeword has 1 in the bit 8 of 3rd byte and 0 in the bit 7. If the bit 8 of the data byte is 0, then the codeword has 0 and 1 for the bit 8 and 7 in the 3rd byte.

1.2 Preliminary Measurements

To determine the error characteristics of the ChipCon radio, we performed several preliminary measurements. A sender node sends the same unencoded packet 10000 times, with the received packets logged into the PC at the receiver end. The experiments were conducted outside Soda hall.

From Figure 4 we observe that most bit errors are single or double bit errors. Burst errors are present but rare. Thus, it is likely that an encoding scheme that corrects single and double bit errors

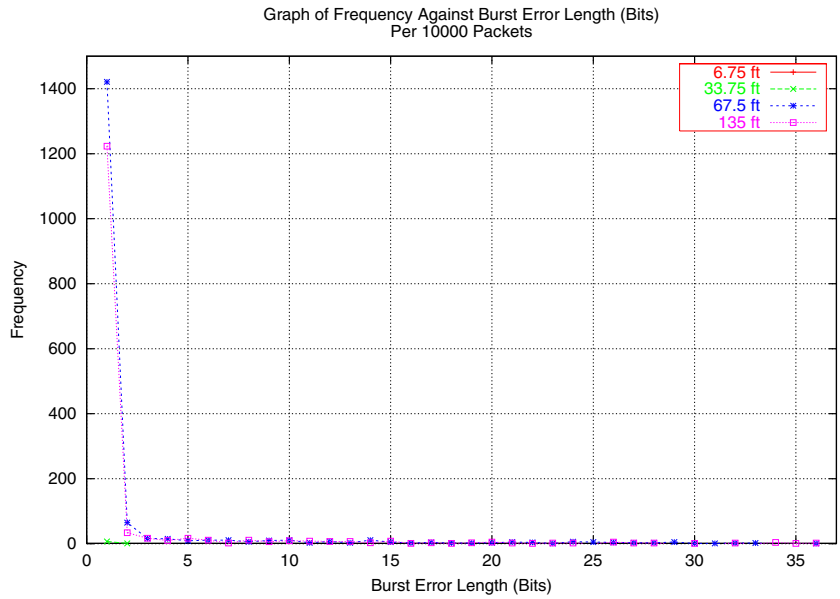


Figure 4: Burst bit error lengths without using ECC

can reduce a significant portion of the errors.

We also took measurements to determine the proportion of packets lost due to preamble misdetection. Figure 5 shows the graph obtained. We note that the loss is very small, about 0.4% at a distance of 135 feet. Thus, we confirmed that preamble misdetection does not cause significant packet loss, which also implies that most of the packet losses will be due to errors in the other parts of the packet.

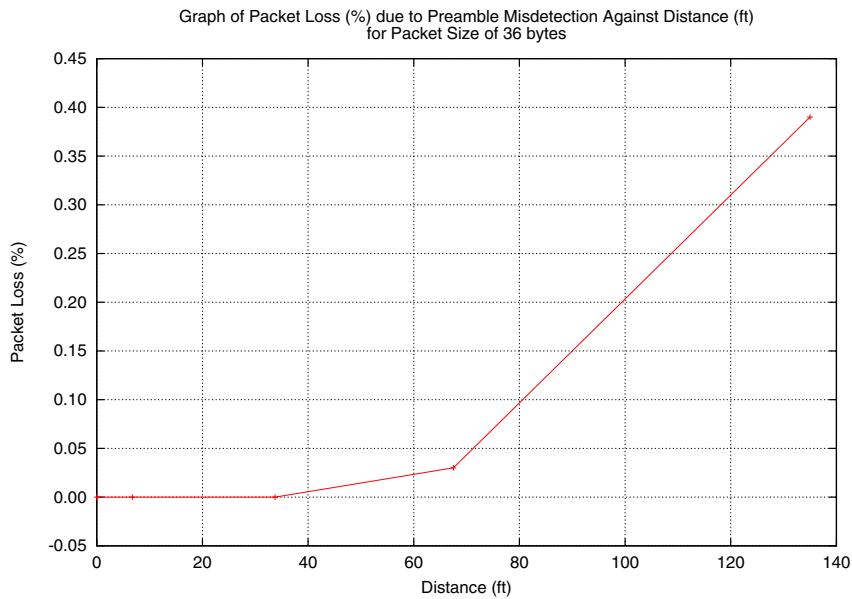


Figure 5: Preamble Misdetection

2 Theory

In this section, we elaborate on the basics of coding theory that is used in our implementation. In particular, we concentrate on *linear block codes*, which are simple and efficient in practice.

2.1 Linear Block Codes

Figure 6 shows the basic schematics of the encoding, transmission and decoding process.

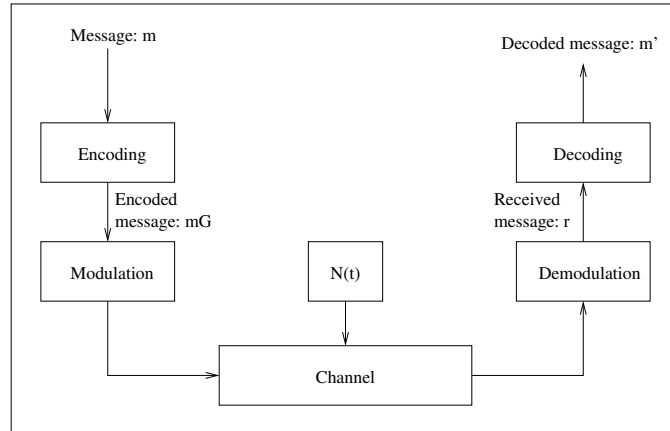


Figure 6: Encoding, transmission and decoding of message

We define an encoding function f to be the functional mapping such that $v = f(u)$ where u is the message and v is the encoded message. Then f is linear such that if

$$\vec{u}_1 \rightarrow \vec{v}_1$$

and

$$\vec{u}_2 \rightarrow \vec{v}_2$$

then

$$\vec{u} = a\vec{u}_1 + b\vec{u}_2 \Rightarrow \vec{v} = a\vec{v}_1 + b\vec{v}_2$$

The general process of encoding, transmission and decoding of the message is shown in Figure 6.

The encoding of message \vec{m} can be achieved by the multiplication with the generator matrix \mathbf{G} . For data of width k -bits, \mathbf{G} is of form $[\mathbf{I}_k : \mathbf{C}]$, where \mathbf{I}_k is the $k \times k$ identity matrix and \mathbf{C} is the $k \times r$ binary matrix.

On the receiver end, the *syndrome* s is calculated from the received signal r . The *parity matrix* \mathbf{H} is constructed from the generator matrix \mathbf{G} and is of the form

$$\mathbf{H} = [\mathbf{C}^T : \mathbf{I}_r]$$

where r is the number of parity bits. Denoting the *error vector* by e , we have

$$s = r\mathbf{H}^T = (m + e)\mathbf{H}^T = e\mathbf{H}^T$$

2.2 Odd-weight column code

Odd-weight column code can correct single bit errors and detect double bit errors (SECDED) [2]. As the name implies, each column of the parity matrix \mathbf{H} has an odd number of 1s.

To construct an odd-weight column code for an input of k data bits, the parity matrix \mathbf{H} should include a sufficient number of parity bits r so that the number of columns of \mathbf{H} is at least $k + r$. For example, $r = 5$ parity bits are needed to recover $k = 8$ bits of data, and $r = 6$ parity bits are needed to recover $k = 24$ bits of data.

The columns of \mathbf{H} are constructed as follows:

- The last r columns of \mathbf{H} form the identity matrix \mathbf{I}_r .
- The first k columns of \mathbf{H} are chosen from any other odd weight column vectors than the ones used in \mathbf{I}_r .

For example, \mathbf{G} , \mathbf{H} for $k = 8$, $r = 5$ are:

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

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

2.3 Double Error Correction code

In this section we describe a (16,8) systematic, quasi-cyclic code that can correct 2-bit errors, as well as detect 3-bit errors.

A quasi-cyclic code is defined to be one whereby a cyclic shift of n_0 digits always produces another codeword [1]. Thus, encoding of the message can be done serially using linear feedback shift registers, or in parallel if low latency is required. Our implementation checks for and corrects single and double bit errors. If any errors are detected, we simply signal to the higher layer that an error has occurred.

Since the syndrome value is represented by 8-bits, this implies that there are a total of 256 possible syndromes. Of these, the zero syndrome is used when there are no errors in the received message, 16 are used by single-correctable errors, and 120 are used by double-correctable errors.

The generator matrix \mathbf{G} is given by

$$\mathbf{G} = [\mathbf{I}_8 : \mathbf{C}] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Note that the minimum Hamming distance between any pair of rows is 5. The parity matrix \mathbf{H} is given by

$$\mathbf{H} = [\mathbf{C}^T : \mathbf{I}_8] = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

We now give an example of how data is encoded, and how the received message can be corrected. Let the message being sent to be 0100 0010. The encoded message is therefore

$$[01000010] \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} = [0100001010011100]$$

Assume that the second and third bits are inverted due to noise in the wireless channel. Then, received message is 0010 0010 1001 1100. Multiplying the received message by the tranpose of the parity matrix \mathbf{H} , we get

$$[0010001010011100] \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} = [10101111]$$

We note that the syndrome obtained is the exclusive-or of the 2nd and 3rd rows of \mathbf{H}^T , thus at the receiver end, we know that the 2nd and 3rd bits are inverted, and we can correct the errors.

3 Implementation

3.1 ECC module

Our error correction code implementation interacts with the network stack via the interface `RadioEncoding`. An interface is a file used in TinyOS nesC language to specify a list of methods and event handlers which a module uses from other modules or which a module exposes.

```
interface RadioEncoding
{
    command result_t encode_flush();
    command result_t encode(char data);
    command result_t decode(char data);
    event result_t decodeDone(char data, char error);
    event result_t encodeDone(char data);
}
```

`encode` is called by MAC layer (`ChannelMonC` module) for each byte of data in the packet. The input data that is passed as a parameter is stored in the internal data structure of ECC module. After a sufficient number of input data bytes have been received, the input data bytes are encoded by an internal function `radio_encode_thread` and codeword is passed to the `ChannelMonC` through `encodeDone` event.

Likewise, `ChannelMonC` calls `decode` for each byte of its received packets. After a sufficient number of input data bytes have been received, the received data bytes are decoded by `radio_decode_thread` and the original data bytes are sent to `ChannelMonC` through `decodeDone`.

The implementation of `radio_encode_thread` and `radio_decode_thread` depends on the type of error correction code. `radio_encode_thread` calculates parity bits by comparing each bit of input data bytes. This corresponds to calculating $m\mathbf{G}$, where m is the message and \mathbf{G} is the generator matrix. In `radio_decode_thread`, the syndrome s is calculated by looking at each bit of received data bits. This is equivalent to $r\mathbf{H}^T$ where r is received data and \mathbf{H}^T is the transpose of the parity matrix. A non-zero syndrome implies an error and the the position of bit errors can be found by comparing the syndrome with column vectors of \mathbf{H} matrix. We made this lookup fast by using an array that maps any possible syndrome value to the error bit position.

We implemented three different error correction codes: `SecDedEncodingOneByte`, `SecDedEncodingThreeByte` and `DecTedEncoding`. Module `SecDedEncodingOneByte` uses an odd weight column code that takes 8-bit data and generates 13-bit codeword (SECDED (13,8)). `SecDedEncodingThreeByte` uses an odd weight column code that takes 24-bit data and generates 30-bit codeword (SECDED (30,24)). Finally, `DecTedEncoding` uses a quasi-cyclic code that takes 8-bit data as input and generates 16-bit codeword (DECTED (16,8)).

3.2 Experimental Tools

We developed application programs that are used in the measurement process.

- **Wireless sensor node application:** This application sends packets at maximum rate by sending the next packet whenever the previous packet finishes transmission. This program sends the same packet and the content of the packet is predetermined so that the receiver node can tell whether the packet is corrupted without using the checksum.

- **Data logging program:** This java application receives a fixed size packet from the receiver node and writes the packet into a file.
- **Data analysis program:** The analysis program compares the logged data with the predetermined contents, and computes the following statistics:
 - Percentage of packets that have corrupted bits.
 - Frequency of burst error length.
 - Number of bits corrupted per packet.
 - Consecutive packet losses.

4 Experimental Results

4.1 Experiment Setup

To see the effectiveness of our ECC implementation, we conducted experiments for the following schemes: **No ECC**, **SECDED original** that was used for RFM radio, **SECDED (13,8)**, **SECDED (30,24)** and **DECTED (16,8)**.

We performed the tests using the same pair of sender and receiver nodes for different schemes. This removes possible effects due to the manufacturing deviation amongst the sensor nodes. The transmitter node sent the same packet 5000 times for each iteration of the experiments. The data received at the receiver node were logged in a PC for further processing.

First, we performed the tests outdoors to determine the effects of distance on packet errors. The sender and receiver nodes were placed along the South Hall Drive in the central Berkeley campus at a distance of 600 feet apart with direct line of sight. Next, we conducted the tests indoors, where the receiver was placed in 373 Cory Hall and the sender node was placed along the 3rd floor corridor at four different places in turn. The locations are shown in Figure 7.

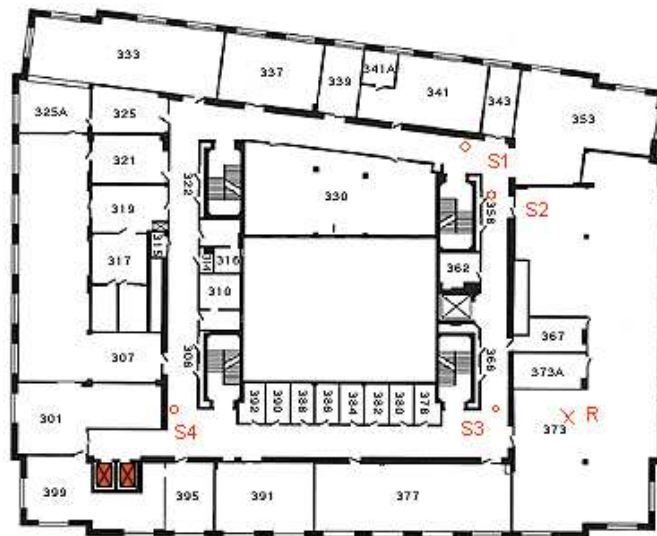


Figure 7: Locations of indoor tests

4.2 Outdoor Tests

Table 2: Packet Loss Rate: 5000 packets (outdoors)

Distance	No FEC	SEC original	SEC (13,8)	SEC (30,24)	DEC
600 ft	0.26%	0.22%	0.02%	0.00%	0.10%

Table 3: Receiving Time: 5000 packets (outdoors)

	No FEC	SEC original	SEC (13,8)	SEC (30,24)	DEC
Time(s)	534.5	836.2	682.3	581.9	682.6

Our ECC implementation was effective for correcting packet errors outdoors. As we can see from Table 2, the percentage of packets that have corrupted bits is less than 0.10%. With no ECC implemented, we had a slightly larger error rate of 0.26%.

Table 3 shows the time taken for the receiver to receive 5000 packets. As expected, the encoding scheme with the smaller overhead has a smaller running time. In this sense, SEC (30,24) is a good choice with small loss rate and transmission time.

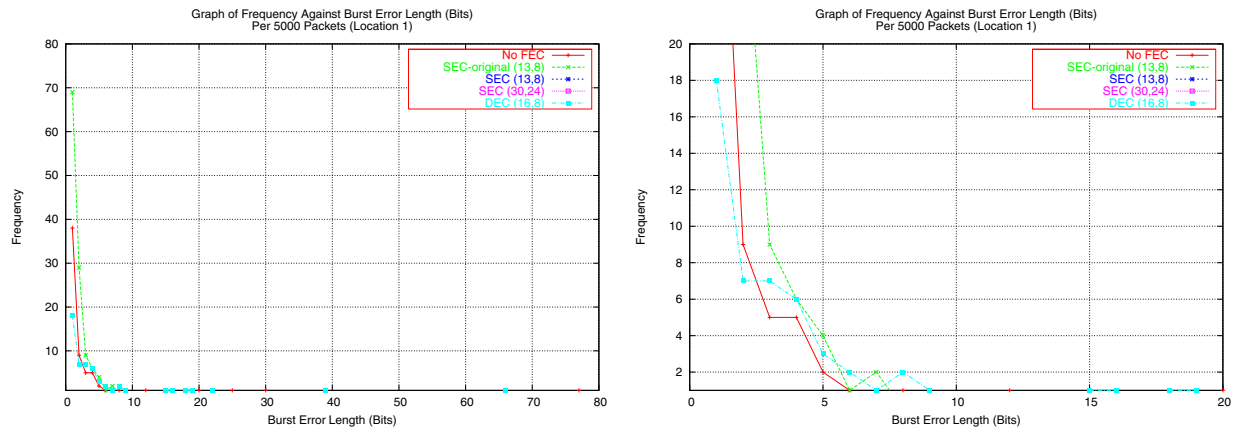


Figure 8: Burst Bit Errors (outdoors, 600 feet)

From Figure 8, we can see that most bit errors are single or double bit errors.

4.3 Indoor Tests

In indoor tests, ECC was not as effective as in outdoor tests. This is due to the increased multiple bit and burst errors as shown in Figure 9.

The percentage of packets that had corrupted bits is rather high: 10% with and without ECC, and more than 40% for SECDED for RFM (Table 4).

Figure 10 gives a snapshot of the data packets logged in a file. As we observed in Figure 9, most errors are single bit errors that are well distributed across the different packets. All the encoding schemes were subject to burst errors even though the frequency varied depending on the time and location at which the experiments were conducted.

Table 4: Packet Loss Rate: 5000 packets (indoors)

	No FEC	SEC original	SEC (13,8)	SEC (30,24)	DEC
Location 1	14.9%	39.8%	7.0%	5.9%	9.4%
Location 2	5.0%	44.5%	9.0%	5.2%	9.2%
Location 3	4.6%	45.1%	9.8%	7.7%	12.2%
Location 4	4.4%	52.3%	10.6%	5.6%	16.8%
Average	7.2%	45.5%	9.1%	6.1%	11.9%

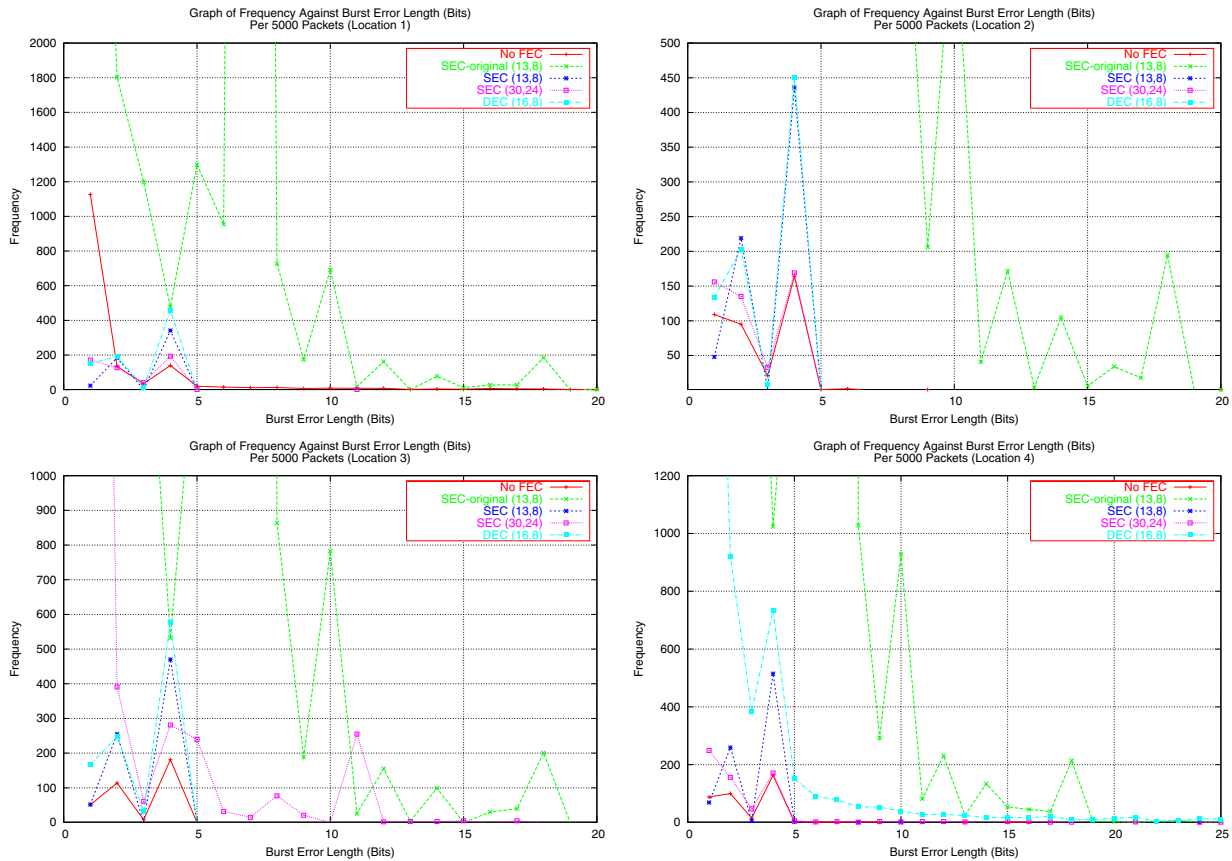


Figure 9: Burst Bit Errors (indoors)

5 Future Work

Our ECC implementations are effective in reducing bit errors when most errors are single or double bit, but are not so effective in the case of multiple or burst errors. We suspect that this is because for

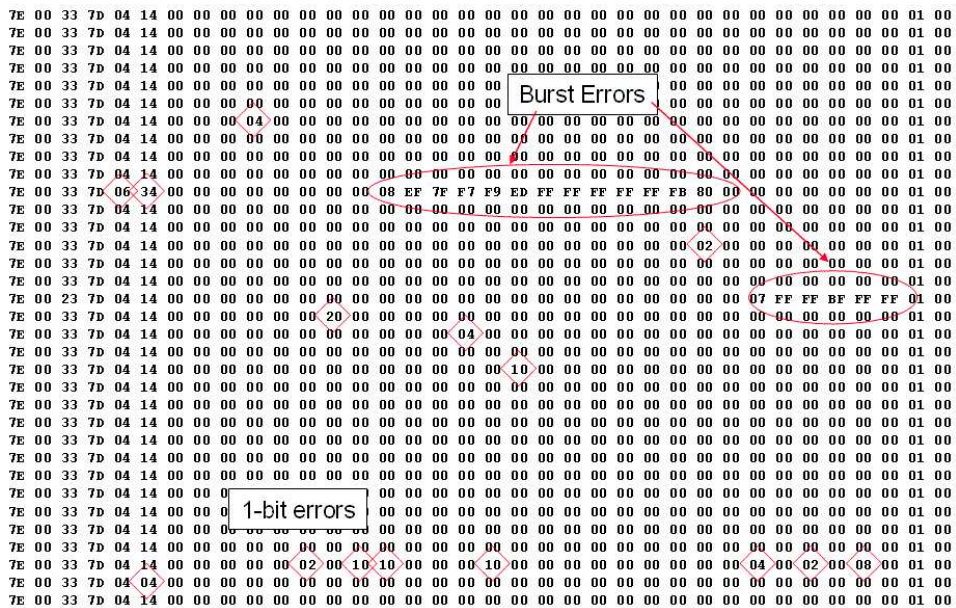


Figure 10: Burst Error Patterns: No FEC, indoors(location 1)

indoor tests, the bit synchronization mechanism of the hardware level Manchester encoding did not work properly. We plan to see whether the Manchester-violation detection provided by the ChipCon radio helps in addressing this problem. We also intend to investigate the effects of changing the transmission rate, to determine if the error rates are affected by the rate.

6 Conclusion

Forward Error Correction is a way of correcting packets by transmitting additional information bits. We implemented and tested different error correction codes on the ChipCon radio. Due to the constraints of low power consumption and small form factor, the error correction codes have been designed to be simple and can correct single or double bit errors. Our ECC implementations are effective when the bit error rate is not high and most errors are single bit. When most of the errors are bursty, our codes are not as effective in reducing packet losses. We expect that we can achieve lower error rates in such cases by using correction schemes that can correct burst errors. However, such schemes are likely to be complex and require high processing power as well as large amounts of storage memory.

Acknowledgements

This work is funded in part by the DARPA NEST Contract F33615-01-C-1815.

References

- [1] T. Aaron Gulliver and Vijay K. Bhargava, "A Systematic (16,8) Code for Correcting Double Errors and Detecting Triple-Adjacent Errors," *IEEE Transactions on Computers*, Vol.42, No.1, Jan. 1993

- [2] S. Kaneda, "A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications," *IEEE Transactions on computers*, Vol.c-33, No.8, Aug. 1984
- [3] E. Jason Riedy and Robert Szewczyk, "Power and Control in Networked Sensors"
- [4] L. L. Peterson and B. S. Davie, "Computer Networks: A System Approach", Morgan Kaufman Publishers, 2000
- [5] J. Jeong and S. Kim, "Dot3 Radio Stack," <http://webs.cs.berkeley.edu/retreat-1-03/>, NEST retreat Jan. 2003
- [6] "ChipCon CC1000 Data Sheet," http://www.chipcon.com/files/CC1000_Data_Sheet_2_1.pdf