SENSOR COMPUTATION AND COMMUNICATION

FOR REMOTE STRUCTURAL MONITORING

By

Olabode Ajiboye

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

August, 2009

Nashville, Tennessee

Approved:

Dr. William H. Robinson

Dr. Akos Ledeczi

ACKNOWLEDGMENTS

I thank God for the opportunities that have been presented to me thus far. His providence is very evident in all aspects of my life and for that I am very grateful. I will also like to thank my family and friends for their words of encouragement throughout the course of my time here at Vanderbilt University.

My Advisor, Dr. Robinson has been a great teacher and an even better role model for me during these past few years. His support has been very instrumental and his very calm and amicable demeanor did more in terms of encouragement than words will be able to express. I also want to thank Dr. Ledeczi for being my second reader, and providing the proper insight and objectivity necessary to make this work a success.

TABLE OF CONTENT

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

The economy of the United States is greatly tied to the transportation system, and bridges are an integral part of the network that connects people and businesses. Therefore, it is important to make the inspection, maintenance and repair of these structures a high priority. Prior to the catastrophic collapse of the bridge in Minneapolis [1], most bridge inspections had been visual with follow-up done only when visible damage is noticed. The event in Minneapolis sparked great interest in the area of computerized inspections that can provide better and more accurate understanding of the structural health of the bridges.

Wireless sensor networks (WSNs) offer a promising method to provide structural monitoring of bridges. A WSN is a collection of spatially distributed autonomous sensing devices also known as motes, which are capable of communicating with each other in a structured or ad hoc manner [2]. The motes are capable of monitoring physical conditions using sensors that are attached to them. A sensor network node's hardware consists of a microprocessor, data storage, sensors, analog-to-digital converters (ADCs), a data transceiver, controllers that tie the pieces together and an energy source [3]. Most WSN applications have to deploy a significant number of sensors into the field, hence slower and cheaper processors are mounted on these nodes to minimize cost. However, advances in the area of semiconductors promise a future with smaller and more efficient microcontroller circuits.

Research in the area of Structural Health Monitoring (SHM) has primarily involved wireless data collection at nodes with centralized data processing. Today, motes are built with faster processors, hence improving the prospect for transferring much of the data processing to the wireless network. The work in this thesis primarily examines the potential limitations of current sensor technologies (e.g., the IRIS mote) to implement a real-time structural health monitoring system with decentralized data processing. These limitations are examined using the performance changes due to the introduction of on-chip processing. The on-chip processing involves the implementation of both a Fast Fourier Transform and a lightweight data encryption mechanism. By processing data at the nodes, additional time will be needed for data processing and encryption. This introduces overhead into the system that impacts the data transmission time; this in turn affects the design of a real-time system. A real-time system is defined as a system that is capable of sampling data at a given frequency with a very small time margin in between sample blocks. The monitoring system should provide an adequate programming platform that can be used for built-in data security and on-chip data processing. It should also utilize existing technology in a way that will provide a thorough and cost-effective means of monitoring bridges in a real-time environment.

Most modern structures have a characteristic natural vibration frequency of about 10 Hz. However by the Nyquist theorem [4], the sampling rate should be at least twice that value to reproduce the original signal with adequate fidelity. In order to reduce the effects of the constant influx of noise, it becomes imperative to sample at rates much higher than the suggested Nyquist levels [5]. There are also other cases such as David

Culler's work that suggest sampling in the kHz range [6]. Prior research in this area suggests a minimum sampling rate of 100 Hz [5].

A secondary goal of the project is to determine a feasible range of relevant sampling frequencies for the IRIS mote. These range of frequencies should not introduce any latency into the system that is being implemented.

This thesis contributes to this research field in the following ways:

- It studies the potential of a structural health monitoring system with decentralized processing.
- It also studies the feasibility of using Wireless Sensor Networks for real-time data acquisition, processing and transmission in a structural health monitoring system.
- It determines a feasible range of relevant sampling frequencies on the specified hardware.

The organization of this thesis is as follows. Chapter 2 provides a brief description of the Structural Health Monitoring application domain and it also gives some perspective on prior work that has been done in this area. This background information will assist the reader in understanding the scope and direction of the work in this thesis. Chapter 3 gives a description of the hardware and software that were used for this project. Chapter 4 discusses the constraints of developing a real-time system, as well as some of the challenges of implementing a decentralized Structural Health Monitoring system. Since much of the decentralized processing will involve conversion of data in the time-domain data into its corresponding frequency domain, Fast Fourier Transforms will also

be discussed. This chapter also describes the different data security options that can be used for the proposed system. Chapter 5 discusses the research challenges that this project tackles by outlining the contributions of this work to the research community, while Chapter 6 provides the experimental setup. Chapter 7 discusses the results and provides an analysis of the collected data. The final chapter summarizes the results as well as provides some directions of possible research in this area.

CHAPTER II

STRUCTURAL HEALTH MONITORING

Structural Health Monitoring is an indirect way of detecting the level of damage that has been done to a structure via natural or human-induced disturbances [6]. Traditionally, Structural Health Monitoring was done using wired systems that collected and monitored data from these structures. This was an expensive and inflexible approach because the system could not be easily redeployed if better data collection points were discovered on the structure.

Wireless Sensor Networks became a good way to solve this problem, and thereby meet a major requirement for a viable SHM system. Autonomous motes could now be deployed over a field of interest while data was collected at a base station [7], [3]. The decision to use WSNs came with a significant trade-off; bandwidth had to be sacrificed for flexibility and price. The radios on the sensors were not capable of transferring data at very high speeds, which also limited the number of motes that can be part of a single network.

N. G. Shrive from the University of Calgary gives a description of the requirements for a SHM system from a civil engineering standpoint [7]. According to the work in that paper, an effective and deployable SHM system needs to have cheap, replaceable and durable sensors with some on-site artificial intelligence and low-power requirements. This proposed system's on-site intelligence should be capable of determining failure in the sensors and power sources. The work in this thesis recommends

the use of wireless data transmission in order to avoid the risk of disruption to wires; wireless systems can also be easily replaced as advances in both software and hardware are made. Lastly, the system should be designed in a way that allows sensibility to certain structural response parameters that are calculated for the particular structure on which it will be deployed.

Prior work done by Deepak et al. discusses the design of Wisden [5], a first generation wireless sensor network that is used for structural data acquisition. According to them, first generation network systems are likely to be used for data acquisition with much of the processing done at a base station node. Their main goal was to implement a reliable data transport as well as a wavelet based compression mechanism to deal with the issue of limited bandwidth. The work in this thesis focuses on implementing a second generation system with more data processing done in the network. Deepak's work implemented their system using the MICA mote, and its performance is compared to that of the IRIS mote which was used to implement this project.

D. Culler et al. describe the implementation of a first generation SHM system on the south tower of the Golden Gate Bridge (GGB) [6]. This work identifies the requirements that a SHM system imposes on a WSN and new solutions to meet these requirements are proposed and implemented. This particular work is one of the largest deployments of a WSN for SHM with data sampled reliably over a 64-node, 46-hop network. The work on the GGB was able to obtain data of sufficiently high quality by sampling at rates much higher than 200 Hz; the sampling rate for the previously mentioned Wisden system. Secondly, the work was designed for scalability, allowing for

dense sensor coverage. It is also important to note that this system was actually deployed on a real-world structure, hence it was able to solve the myriad of problems that come with a real deployment. This resulted in meaningful and reliable calibrated data which is usually missing in prototype implementations. This work also found out that smaller packet sizes tend to be a bottleneck for multi-hop network data transmission bandwidth, however, larger packets sizes do not necessary provide a solution to this problem due to the limited amount of RAM space on the motes.

Although the current work in this area involves the traditional implementation of a SHM system with a single data collection point, allusions have been made to a decentralized system as the future of research for bridge monitoring. With advances in semiconductor technology, it is becoming easier to manufacture motes with faster processors. This makes it possible to transfer the data processing tasks to the sensor nodes that make up these networks as shown in Figure 1. The main task that is transferred is the Fast Fourier Transforms and additional tasks like data encryption and compression can also be implemented on the sensors. This idea is currently mitigated by the issue of energy supply for such systems. Wireless Sensor Networks have autonomous nodes that have a limited amount of energy supply, hence limiting the amount of processing and transmission that can be done on these sensors. The use of Wireless Sensor Networks for structural monitoring also introduces the issue of data security, making the implementation of a data encryption mechanism a primary goal.

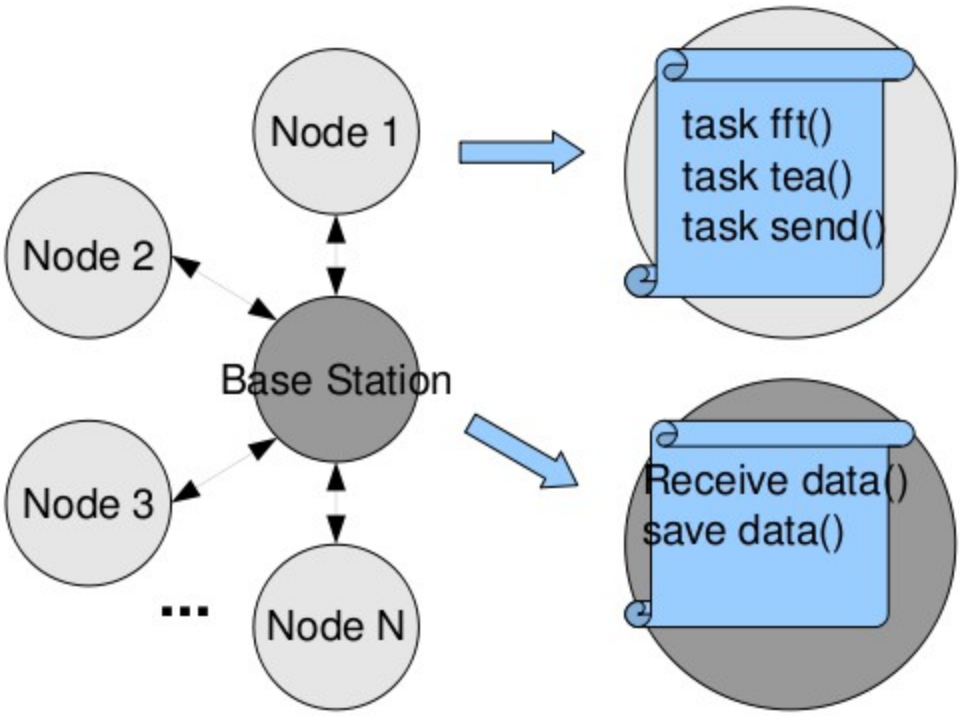

Fig. 1 – System Architecture

All these advancements in radio transmission rate and increased computational capacity in low-cost sensors make real-time structural health monitoring an interesting idea. Real-time data monitoring involves continuous data capture with a very small time margin between data sample blocks. The marginal time is represented as a percentage of total execution time, and the acceptable threshold will be set by the system designer. This idea forms the basis for this thesis work where a single-hop network will be observed and characterized for continuous data sampling and on-chip computation.

CHAPTER III

HARDWARE AND SOFTWARE RESOURCES

Hardware Resources

A Wireless Sensor Network (WSN) is a collection of sensors that are capable of communicating with each other in a structured or ad hoc manner. Each sensor has a processor for data processing, a radio interface for wireless communication and a dedicated memory for storing data and programs. The project characterizes the feasibility of Crossbow technology's products for the purpose of real-time data collection and processing. The products used are listed below:

- MTS310CB which is a basic sensor board that plugs into the mote. It features a light and temperature sensor, a microphone, sounder and an accelerometer [8].

Fig. 2 – MTS310CB

- XM2110CA (aka Iris mote) which is a mote module used for enabling low power, wireless sensor networks. It is capable of sending data at 250 Kbps and also has node routing capabilities used by multi-hop networks [9].

Fig. 3 – XM2110CA

- MICA is an 868/916 MHz mote that transfers data at 38.4 kbaud and is used extensively in wireless sensor networks. This hardware is introduced here because its performance will be compared to that of the IRIS mote later in this paper.

Fig. 4 – Mica Mote

- MIB520CB which is a base station for the entire wireless network. It allows for a wired connection with the PC via the USB port. The port is used for programming the motes and communicating with them as well [10].

Fig. 5 – MIB520CB

Software Resources – TinyOS And NesC

TinyOS [11], [12] is a platform independent operating system that was used as the software interface for the hardware products listed in the previous section. It allows for wired or over the air configuration of the motes. Programming in the TinyOS environment is done in a language called nesC (network embedded system C) which is a variant of the C programming language. Although it is a C variant, there are very marked differences in the programming styles. The nesC language is an event-driven programming language that is used to directly control the hardware on the sensors. It is also a modular language with applications being built from other smaller components that are able to provide the necessary functionalities. These components are called Modules, and they need to be joined together via a process called "Linking" in order to build a larger working application. The modules are very similar to objects conceptually; they encapsulate and couple state with functionality. However, the principal distinction lies in the naming scope. Unlike C++ and Java objects, which refer to functions and variables in a global namespace, nesC components use a purely local namespace. This means that in addition to declaring the functions that it implements, a component must also declare the functions that it calls. The name that a component uses to call these functions is completely local. When a component "A" declares that it calls a function "B", it is essentially introducing the name A.B into a global namespace. A different component, C, that calls a function B introduces C.B into the global namespace. Even though both A and C refer to the function B, they might be referring to completely different implementations. Every component has a specification, a code block that declares the functions it provides

(implements) and the functions that it uses (calls). For example, the specification for a fictional component SampleC is given as:

```
module SampleC {
provides command uint8_t send_data(uint8_t* array, uint8_t len);
uses command uint8_t get_data(uint8_t* array, uint8_t len);
}
```

Because SampleC provides the function "send_data", it must define it so that other components can call it. Conversely, because SampleC uses "*set_data*", it can reference the function and so depends on some other component to define it. Components can always reference functions that they define, meaning SampleC can call function "*send_data*" on itself. In practice, components very rarely declare individual functions in their specification. Instead, NesC has interfaces, which are collections of related functions. Component specifications are almost always in terms of interfaces. For example, some applications require stop and start services. The StdControl interface is a common way to express this functionality:

```
interface StdControl {
command error_t start();
command error_t stop();
}
```

A component representing an abstraction or service that can be turned on or off provides StdControl, while a component that needs to turn others on and off uses StdControl. This is often a hierarchical relationship. For example, a routing layer needs to

start a data link packet layer, which in turn needs to start and stop idle channel detection:

```
module RoutingLayerC {
provides interface StdControl;
uses interface StdControl as SubControl;
}

module PacketLayerC {
provides interface StdControl;
}
```

Connecting providers and users together is called wiring. For example, RoutingLayerC's code has function calls to SubControl.start() and SubControl.stop(). Unless SubControl is wired to a provider, these functions are undefined symbols, meaning they are not bound to any actual code. However, if SubControl is wired to PacketLayerC's StdControl, then when RoutingLayerC calls SubControl.start(), it will invoke PacketLayerC's StdControl.start(). This means that the reference RoutingLayerC.SubControl.start points to the definition PacketLayerC.StdControl.start. The two components RoutingLayerC and PacketLayerC are completely decoupled, and are only bound together when wired.

For computationally intensive programs, TinyOS allows the use of "tasks" which are synonymous to functions in any other programming language. These tasks can be triggered inside an event. For example, it is possible that a programmer wants to perform some operation on a set of data collected from the sensors. Once the readings arrive, an

event is triggered, and the programmer can then place a call to a task to perform some arithmetic operation on the newly gathered data. Unlike the C language, tasks need not be declared, but they still need to be implemented before they are called.

CHAPTER IV

SYSTEM CONSTRAINTS AND APPLICATIONS

System Constraints

Real-Time Constraints

The concept of real-time means different things to different system designers, but for the scope of this project, it refers to a system that is capable of sampling data at a given frequency, with a very small time margin in between sample blocks. The major constraints of such a system for structural health monitoring are discussed below:

- Energy: wireless sensors have a limited power source and this limits the amount of data computation and transmission that can be done. Real time systems require continuous data sampling, processing and transmission, and this energy constraint poses a major restriction to the implementation of such a system
- Data transmission and storage: Motes are very small computers that have limited memory on them. This limits the amount of data that can be stored intermediately prior to processing and transmission. Intermediate data storage is a major part of real-time systems because information needs to be buffered in order to support the idea of continuous data sampling and transmission.
- Sampling frequencies: The frequency at which data is sampled is critical to the successful implementation of a real-time system. The sampling frequency has

to be high enough to detect the natural vibration frequency. However, sampling at a high frequency poses a few challenges. It results in more data points, hence requiring more intermediate data storage. It also means that more data will be transmitted over the radio hence utilizing more power.

Although there are many factors that limit the implementation of a real-time system, it is still important to pursue scholastic ideas that explore the potential of the components that make up the system. This work can be categorized as a probe into the potential implementation of a real-time monitoring system.

Message Passing Specifications

TinyOS 2.x is only capable of sending a maximum of 128 bytes. The default payload size in each message is set to 28 bytes, but this value can only be altered at compile time as stated in the programming manual. This poses a problem for applications that are designed to send packets of varying size. TinyOS 2.x uses a data structure called "message_t" to organize the data that will be sent across the network. In the "message.h" file under the *"tinyos2.x/tos/types"* directory, the variable TOSH_DATA_LENGTH is used to store the length of the data field. Its default size is 28 bytes, but its value can be changed at compile time with the command line option: DTOSH_DATA_LENGTH = x.

Since this value is reconfigurable, it is possible that two different versions of an application can have different packet sizes. If a packet layer receives a packet whose payload size is longer than TOSH_DATA_LENGTH, it must discard the packet. This work addresses the limitations of fixed packet sizes and the maximum length of a packet. This is mainly relevant for cases where particularly long data sets are needed for signal

processing computations.

Applications

Fast Fourier Transforms

Fast Fourier Transforms are integral in representing and analyzing the information gathered from the bridge structures. This project focuses mainly on the vibration frequency parameters that are gathered using the accelerometer device on the sensor. This data will then be separated into its frequency component using a Fast Fourier Transform. A Fast Fourier Transform (FFT) is an efficient algorithm used to compute the Discrete Fourier Transform (DFT) of a discrete signal. A DFT decomposes a sequence of values into components of different frequencies. For the case of this thesis, a series of time domain data are passed through a DFT algorithm to get the frequency domain components of the input data. An FFT is essentially a faster implementation of a DFT computation. Computing a DFT of N points takes $O(N^2)$ arithmetical operations, while an FFT can compute the same result in only $O(N \log N)$ operations. The difference in speed can be substantial for long data sets. The computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to $N/\log(N)$. Let $x_0, \ldots, x_{N-1}$ be complex numbers, the DFT is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} nk} \quad k = 0, \ldots, N-1$$

Fig. 6 – DFT Algorithm

This project adapts the Cooley Tukey [13] algorithm which is a very popular computation of the DFT. It uses a divide and conquer algorithm that recursively breaks down a DFT of size X into two pieces of size X/2 at each step, limiting the length of the input sequence to powers of 2. This method is generally referred to as the Radix-2 method.

For illustrative purposes, Figure 6 depicts the computation of an 8-point DFT (**N** = 8). Observe that the computation is performed in three stages, beginning with the computations of four two-point DFTs, then two four-point DFTs, and finally, one eight-point DFT.

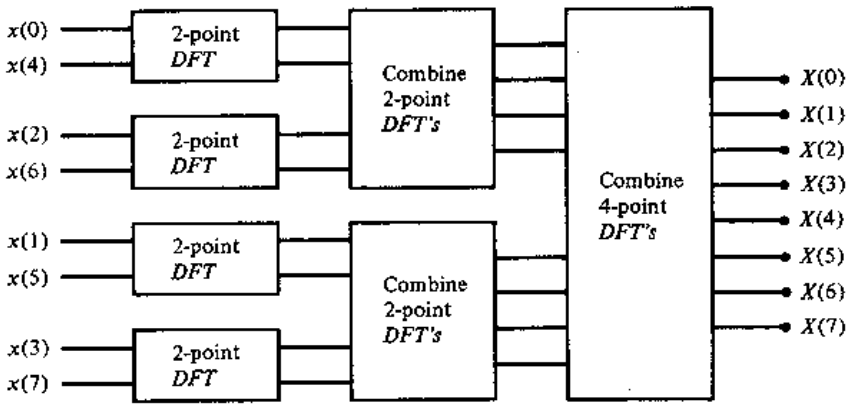

Fig. 7 – Computing an 8 Point DFT [14]

Once all the 2-point FFTs have been calculated, the values are then recombined in a butterfly combination as illustrated in Figure 8.

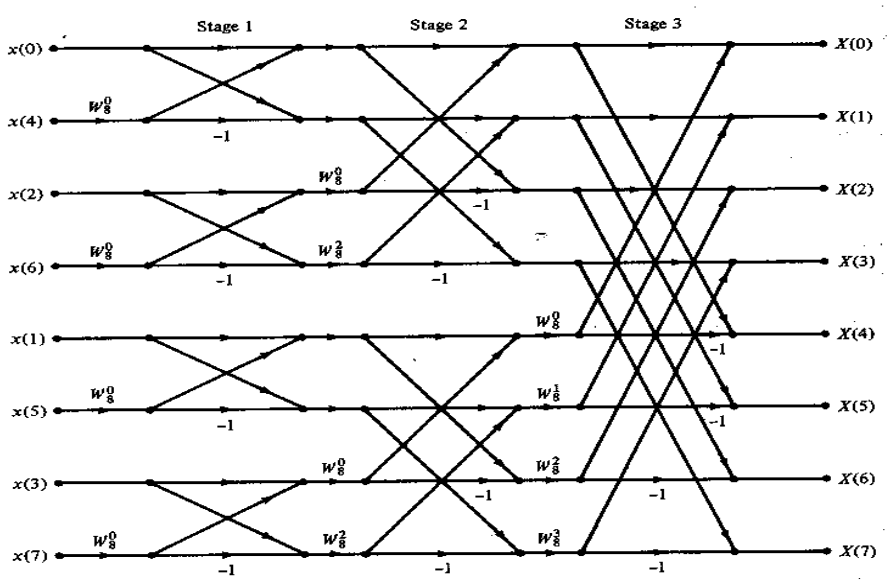

Fig. 8 – Eight-point decimation-in-time FFT algorithm [15]

Tiny Encryption Algorithm

There are many encryption algorithms that have been developed since the early 70s, but most of them are usually computationally intensive. This makes them unsuitable for deployment on sensors nodes due to the low processing capabilities of these motes. The Tiny Encryption Algorithm (TEA) [16] was used in this project because of its simplicity and ease of implementation. Prior to providing the details about TEA, it is important to mention TinySec [17], a link layer encryption mechanism which is meant to be the first part in a suite of security solutions for TinyOS devices.

TinySec

TinySec is a link layer encryption mechanism that was created to address security issues in devices where energy and computational power present significant resource

limitations. Just like the TEA, TinySec also implements a block cipher with a single symmetric keying system, hence providing a similar level of security. The main goals for TinySec are to provide the following:

Access control: by ensuring that only authorized nodes are able to participate in the network because only authorized nodes have access to the shared group key.

Integrity: by making sure that messages are only accepted provided they have not been altered in transit.

Confidentiality: by ensuring that outside parties are not able to infer the content of messages.

Simplicity: by providing a security stack that provides the three goals mentioned above which is no more difficult to use than the traditional, non-security aware communication stack.

The Tiny Encryption Algorithm was chosen over the TinySec for the two reasons listed below:

- Tiny Encryption Algorithm is platform independent.
- Unlike the TinySec implementation, Tiny Encryption does not increase the size of the original packet.

The TEA is a block cipher that was developed by David Wheeler and Roger Needham at the Computer Laboratory of Cambridge University. It is one of the fastest and more efficient cryptographic algorithms in modern times. It is a Feistel cipher which uses logical operations like - XOR, ADD and SHIFT. It encrypts 64 data bits at a time

using a 128-bit key. The key schedule is extremely simple, with all the parts of the key used in exactly the same way for each cycle. It seems highly resistant to differential cryptanalysis, and achieves complete diffusion (where a one bit difference in the plain text input will cause approximately 32 bit differences in the cipher text) after only six rounds. Performance on a modern desktop computer or workstation is very impressive, and it can also be implemented on a mote. The simplicity of the TEA algorithm makes it vulnerable to certain attacks. One of its main weaknesses is the problem with “equivalent keys”; each key is equivalent to three others which means that the effective key size is only 126 bits. However, it provides the level of security needed for the scope of this work.

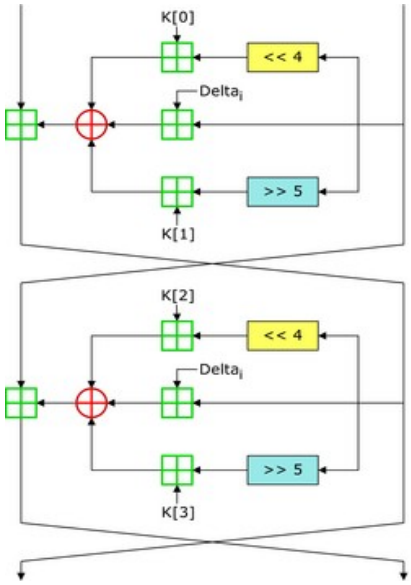

Fig. 9 – A Feistel structure [18]

According to Simon Shepherd, Professor of Computational Mathematics at Bradford University, the TEA is a relatively safe algorithm with no known successful cryptanalysis. It uses the same mixed algebraic group technique as the International Data

Encryption Algorithm (IDEA), but it is much simpler making it faster as well. The code is very lightweight and portable enough to be used on small computers such as those on sensors. The minor weaknesses identified with the TEA algorithm are rectified in a new variant of the algorithm called Block TEA or XTEA[16]. TEA fits the bill for a low-overhead end-to-end cipher encryption.

CHAPTER V

RESEARCH CHALLENGE

Using Wireless Sensor Networks for Structural Health Monitoring has become the prevalent consensus amongst researchers in this area [3]. Most current systems implement sensors that are able to send raw information to a base station. This base station mote is typically connected to a computer that is capable of processing large volumes of information. Also, little work has been done in the area of real-time data collection for structural health monitoring. Most systems are either event driven, or collect data at a delayed time interval.

The work in this thesis studies the potential of a structural health monitoring system with decentralized data processing. This monitoring system will have nodes that are capable of processing data prior to sending them over the network. This also allows for the possibility of implementing data security and/or compression techniques on the sensor nodes.

Some of the system implementation challenges involve exploring the effects of sending data sets of multiple sizes across the network, as well as sending an aggregate data size of more than 128 bytes in cases where it is needed. This project naturally exceeds this since it involves the calculation of Fast Fourier Transforms which at times may require up to 1024 points of data.

By measuring the accuracy of the results sent across the network for varying packet lengths, the computational feasibility (i.e. capacity) of the IRIS mote is

characterized for real-time and computationally-intensive processing.

Most computers have a MIPS (millions of instruction per second) specification that is used by researchers as a benchmark for selecting the right machine for their projects. This work characterizes the computational efficiency of the sensors by measuring the performance based on the computation of Fast Fourier Transforms and the implementation of a data encryption algorithm on multiple lengths of data in real time at different data sampling frequencies. It is worthwhile to pursue these concepts because sensors are used for a variety of applications, and hence an adequate characterization of their computational capacity may be helpful in selecting a particular mote to match a user's need.

The resonant frequency of a structure is that frequency which causes the structure to vibrate at maximum amplitude. This in turn causes the structure to be under tremendous stress. Although most structures have a natural resonant frequency of less than 10 Hz, it is important to sample at a rate much higher than that in order to increase the fidelity of the signal as well as eliminate problems caused by noise. This is the reason why this project characterizes the performance of the motes for multiple frequencies. It is possible to implement a filtering algorithm to solve the issue with noise, but that will be additional tasks that will have to be implemented on the sensor.

CHAPTER VI

EXPERIMENTAL SETUP

This project uses the Oscilloscope program [6] that sends periodic data gathered from a sensor to a base station. This algorithm was altered to implement the solutions to the problems that were discussed in earlier sections. In order to solve the problem of having a limited packet size, an algorithm was implemented that is best described as a *"Burst Mode"* application for the sensor. It is a way to change the amount of data that is sent across the network, and it also serves as a foundation for characterizing the effectiveness of performing large-scale computation on a relatively large set of data. A buffer is created which holds the values collected from the sensor prior to sending them across the radio. There are some control parameters that monitor the accumulation of data in the buffer. These parameters are used to initiate a rapid succession of data being sent across the radio once the buffer is full. This is useful when data processing has to be completed before the final values are sent.

Characterizing the performance of the motes requires the gradual increase in the size of the buffer which holds the values that need to be processed within the TinyOS tasks. This approach is useful for identifying the buffer sizes that result in an excessive computation time; excessive computation time is unacceptable for real-time applications.

The following subsections will discuss the approach used in this experiment for collecting useful data; these data are used to characterize the feasibility of the given

hardware for use in a real-time structural health monitoring system.

The following sections provide a description of the components that are integral to setting up this project. It also explains the main concept that was used to tackle the issue of propagating large data sets across the network. The penultimate section discusses the methods used to verify the integrity of the data produced from the FFT and TEA algorithms. The final section provides a description of how the computational performance of the processors will be analyzed.

Oscilloscope Application

The Oscilloscope application (APPENDIX) is a simple data-collection program that periodically samples the default sensor and broadcasts a message over the radio once a specified number of readings have been reached. These readings are received by a base station mote, and the values can be collected in a variety of ways. The original version gathers the data by using a Java program that shows the values in a Graphical User Interface as shown in the figure 10.

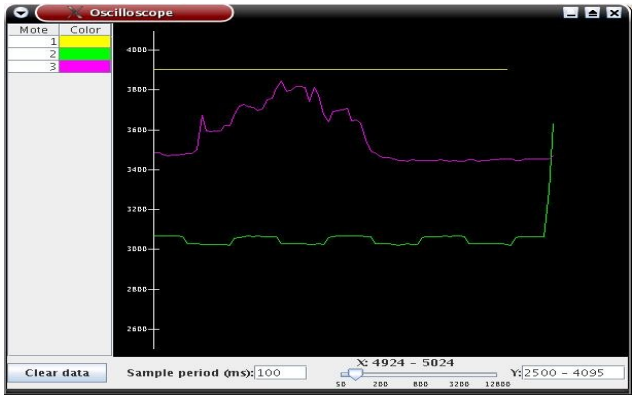

Fig. 10 – Java GUI for data collection at the base station

The lines represent values that are being sent from independent motes that have been programmed to report to this particular base station. For the purpose of this experiment, a packet sniffer is used to collect all data sent over the radio from the sensor node to the base station. The packet sniffer intercepts the raw data that is sent over the radio to the basestation, and this data is used for analysis, verification and graphing. More details about this application is provided in the next section.

Network Data Monitoring

It is very important to be able to monitor the values that are passed between the sensor node and the base station. TinyOS has a Java tool that listens to the network port of the computer for incoming packets sent by the sensor node. The command line option used to initialize this listening device is:

"java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB1:iris > output.txt"

The "-comm" option specifies the target port on the computer, the ":iris" portion specifies the type of sensor sending the data, and the "output.txt" is the name of the file

that stores the collected data. The packets are displayed in their raw form with all the additional meta data that accompany the payload information.

However, it is important for the programmer to know what section of the raw data represent the actual sensor readings that are sent across the network. A Perl script "payload.pl" (see APPENDIX) was written to parse the incoming values sent across the network in order to extract the useful payload information. The script can be altered easily to fit any packet size increase up to the maximum allowed.

A message sent through serial ports from the motes to the PC will typically look like Figure 11:

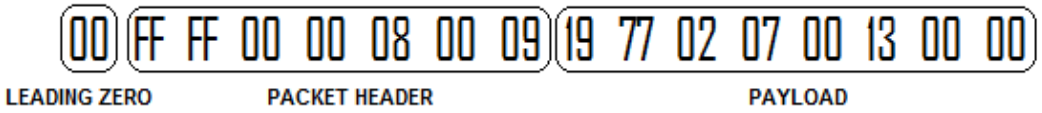

Fig. 11 – A sample message

The first byte "00" is a typical leading zero to denote the beginning of the message. After that, the whole packet is of the type "serial_packet_t". Inside the serial_packet_t structure, the first 7 bytes are of the type "serial_header_t", which is a data structure that holds the meta data or descriptive information of the message that is being sent across the network. In Figure 11, the meta data will be the next 7 bytes that immediately follow the leading zero.

The structure of "serial_header_t" is given below:

- nx_am_addr_t dest **Destination (2 bytes)**
- nx_am_addr_t src **Source (2 bytes)**

- nx_uint8_t length **Payload Length (1 byte)**
- nx_am_group_t group **Group ID (1 byte)**
- nx_am_id_t type **Active Message Handler Type (1 byte)**

After the 7-byte header comes the payload, which can be up to TOS_DATA_LENGTH (28 bytes) long. The content is determined by the payload and has no restrictions other than the length. In Figure 11, the payload is represented by the last 8 bytes. The length of this section is denoted in the "Payload Length" field in “serial_header_t”.

The structure of the message is such that the header and the payload constitute one packet. The Oscilloscope application described in the appendix sends out these messages one packet at a time. The next section describes a method that sends a larger data set over the radio.

Propagating Large Data Sets

Sending large quantities of information from the sensor node to the base station requires a mechanism that rapidly sends chunks of the data set in a sequential manner. Due to the limitation on the maximum packet size that could be sent across the network, a system had to be devised that could send parts of the larger data set in smaller groups until the entire set was successfully sent. This implementation, called the “burst mode” allows data of varying sizes and lengths to be sent across the network. It also provides the possibility of performing operations on a larger data set prior to sending the resulting data

across the network. This idea will make it possible to perform Fourier Transforms on much larger data points.

The size of the input array is crucial when performing Discrete Transforms of a sequence of digitized data. For example, a 256-point FFT requires an array that is as many points long. This is a relatively large dataset when compared to the packet size limitation of 28 bytes, as well as the current implementation of the Oscilloscope application. Every packet in this application has a payload portion that holds eight 2-byte values that represent the actual readings that are taken from the sensor boards. Therefore, a dataset with 256 points will require the program to rapidly send 32 packets across the network in order to begin the collection of new data into the auxiliary buffer space. The efficiency of the send sequence ultimately determines how much time elapses until the collection of sensor data in subsequent batches.

The original code (see APPENDIX) initiates the sending of data across the network when the preset number of readings has been taken from the sensor. A buffer is created to store the data collected from the accelerometer prior to initiating the send sequence. The size of the sample dataset is controlled by another value that is set within the header file. Once the buffer has been filled, a sequence of tasks are initiated to process the stored information. A flag mechanism was implemented to trigger the send sequence, hence ensuring that all the values in the buffer are completely processed before the send sequence is initiated. The event-driven nature of TinyOS is then exploited for the "burst mode" implementation. An event

"event void AMSend.sendDone(message_t* msg, error_t error)" is triggered every time

a packet is sent across the network. Another send command can then be implemented inside this event to initiate a recursive send pattern. This happens because the send command will trigger the same event ***"event void AMSend.sendDone(message_t* msg, error_t error)"*** again, hence creating a loop. This loop sequence is terminated once the predetermined number of packets have been sent across the network.

The entire application will be compromised if the values being stored in the buffers are incorrect or inaccurate. For this reason a method was devised to ensure the accuracy of the data stored in the buffers from the FFT and TEA implementations. This verification process and its results are explained in the next section.

Verification Of FFT And TEA Implementations On IRIS Mote

It is important to verify the correctness of the values that are being gathered on the processors on the motes. In other words, there needs to be a way to make sure that the values sent across the network are valid. In the case of a computationally intensive FFT, it is imperative that the final values being sent across the network actually represent the Fourier transform of the initial signal stored in the buffer. For this reason, a program was written to implement the same dataset on a local PC using a readily available language such as C++. Sample signals were generated using sine functions on both the sensor and the PC as inputs into the buffers that are used for calculating the FFT. The results of both applications are represented in Figure 12. Note that the Perl script described earlier was instrumental in gathering and plotting the data sent from the motes.

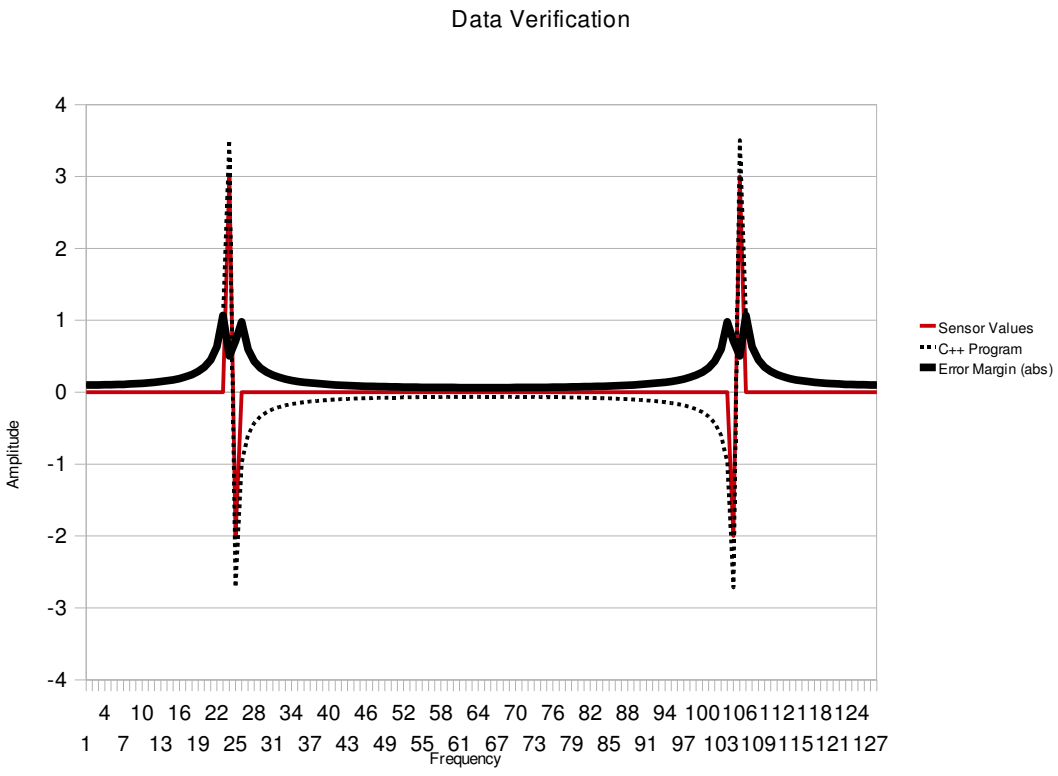

Fig. 12 – Data Verification with error margin

Figure 12 shows the 128-point FFT computation for a signal represented by the equation 10*sin(20). Much of the errors observed are as a result of the level of precision available on the sensor. Although the FFT computation is performed and stored in floating precision, the value is stored as an integer prior to transmission, thereby reducing the amount of data sent over the radio by 75 percent. This is the reason for the error margin that is displayed along the thick black line. The same techniques were used to verify the correctness of the TEA algorithm on the motes.

The FFT and TEA algorithms are the tasks that are implemented in this experiment. One of the goals of the project was to document the marginal time that is

introduced into the system due to executing these algorithms. The next section describes the concept of measuring time in the TinyOS environment and how this is used to characterize the performance of the processors on the motes for different data sizes and frequencies.

Characterizing The Computational Performance Of The System

Most operating systems and microcontrollers come equipped with a robust system-time implementation, offering feature like:

- Multiple counters and clocking options
- One or more comparison registers for counters with interrupt triggers and
- Time stamps for sent and received data

This provides programmers with an easy way to measure the execution time of a particular program or application. However, TinyOS does not attempt to capture all this diversity in a platform-independent fashion. Instead, each processor is able to expose its applications to a system-like timing mechanism through the implementation of the functionality of the underlying components and interfaces at the hardware level. The Local.timer() component is used in this experiment to provide a pseudo system timing mechanism. These measurements will be used to compare the marginal time required for different data set sizes.

The elapsed time and the amount of memory used by the sensor are the two integral parts to performing a feasibility characterization of the motes. Memory space is a

crucial metric because the amount of space on the sensor is very limited, and this in turn affects the size of the information that can be processed and stored within the application. Time is also a major factor in this case because of the requirement to process information in a real-time manner.

The amount of data memory used was monitored for different FFT lengths and the limit was recorded. The percentage of time for performing the FFT and TEA algorithms will also be studied for different frequency values. This will help in making recommendations as well as concluding the feasibility of IRIS motes for real-time data monitoring.

Due to the lack of an output stream interface such as “cout” in C++ or “print” in Java, the timer values had to be incorporated into the message packet structure. The TinyOS timer interface provides a 32-bit timer value that is synonymous to a counter; it counts from “0” to “2^{32}” before starting over at “0”. The message packet structure looks like this:

```
typedef nx_struct oscilloscope {
  nx_int16_t version;
  nx_int16_t interval;
  nx_int16_t id;
  nx_int16_t count;
  nx_int32_t timer;
  nx_int16_t readings[NREADINGS];
} oscilloscope_t;
```

The goal is to compute the total time it takes to collect and process one batch of data. This information will then be broken into parts constituting the Buffering time, Task

processing time and Data sending time. The total execution time is calculated by assigning the value of the timer to a variable at the beginning of the buffering sequence. The difference between the timer values after two consecutive batches provides a rough estimation of the execution time. The execution time for the tasks, FFT and TEA, are also derived by extracting the timer value right before the task is called, and right after it performs its last operation. The following equation describes the total time and its constituents:

$$Time_{TOTAL} = Time_{N} - Time_{N-1} + Time_{FFT} + Time_{TEA} + Time_{SEND}$$

Although it is possible to calculate the amount of time it should take to send data over the radio from the product specifications, this information will be quite misleading. The recursive nature of the “Burst Mode” implementation is responsible for this. For each section of data sent over the radio, an event has to be triggered upon successful transmission. This event in turn prompts the transmission of the subsequent batch. This creates additional overhead time which is almost impossible to calculate explicitly. Therefore, the transmission time is then derived by rearranging the previous equation.

$$Time_{SEND} = Time_{TOTAL} - (Time_{N} - Time_{N-1}) - Time_{FFT} - Time_{TEA}$$

CHAPTER VII

RESULTS

Timing Analysis

The collected data are displayed and discussed in a way that supports the contributions of this thesis work to the research community. These contributions are restated below:

- A study of the potential of a structural health monitoring system with decentralized processing
- A study of the feasibility of using Wireless Sensor Networks for real-time data acquisition, processing and transmission in a structural health monitoring system.
- Provides a feasible range of relevant sampling frequencies on the specified hardware.

The sensors are capable of operating at different sampling frequencies and this affects the feasibility of the sensor for different data collection scenarios. Timing information was gathered while operating the mote at different sampling frequencies, starting from 32 Hz. Although this frequency may be insufficient for preserving the fidelity of the signals in the presence of noise, it does satisfy the Nyquist theorem for sampling at a frequency that is at least twice the value of the expected signal; most structures have a natural vibration frequency of no more than 10 Hz. The sampling frequency of the mote is then doubled for subsequent data points and the efficiency of the

sensor is analyzed for higher frequency values. Data collection was stopped at 1024 Hz because it represented a threshold for efficient on-chip computation. At this frequency, an increase in the buffering time was observed.

After taking the time measurements, it was verified that the amount of time that was required by the processor to compute the FFT and TEA algorithms were constant regardless of the frequency that was used. This is consistent with expectations because the task execution time should not depend on the sampling frequency. This information will be used to analyze the performance of the sensor at the different sampling frequencies.

An approximation of the overall execution time for one batch of data is determined by subtracting time stamps between consecutive batches. The time information is retrieved from the same point within the program; right before the buffering begins. The total execution time is broken down conceptually as follows:

$$Time_{TOTAL} = Time_{N} - Time_{N-1} + Time_{FFT} + Time_{TEA} + Time_{SEND}$$

The percentages of the amount of time spent computing the FFT and TEA are provided in the following graphs:

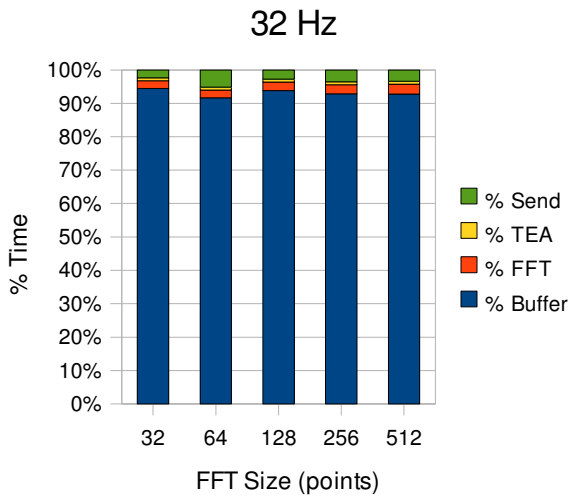

Fig. 13 – 32 Hz time Graph

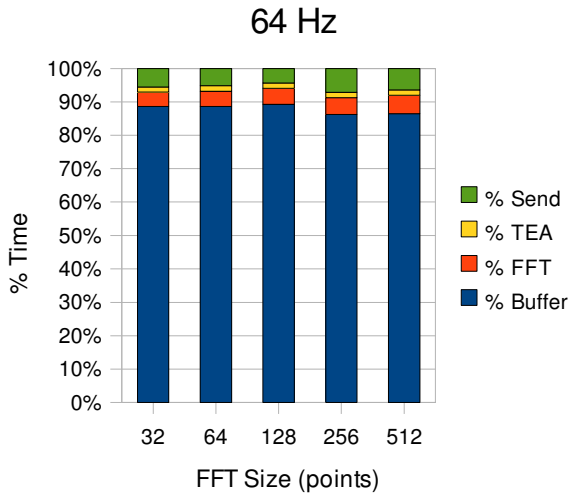

Fig. 14 – 64 Hz time Graph

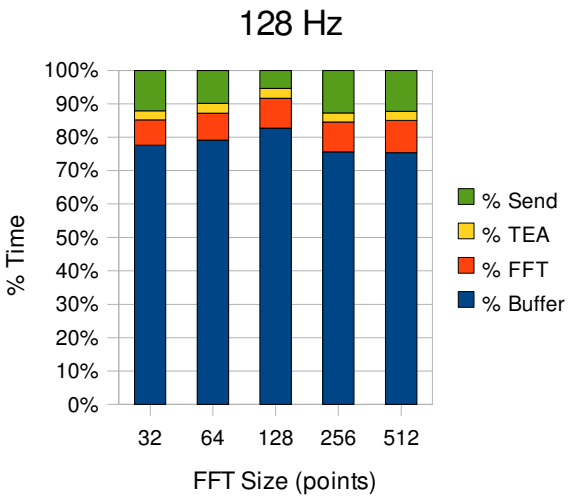

Fig. 15 – 128 Hz time Graph

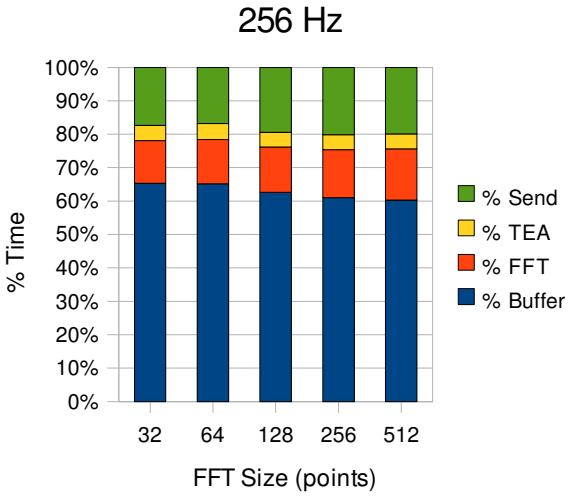

Fig. 16 – 256 Hz time Graph

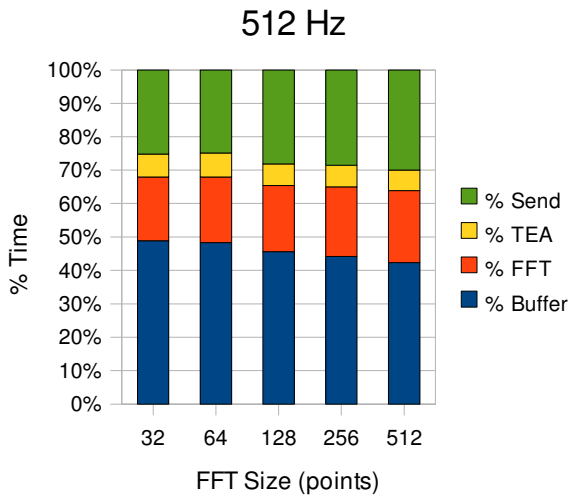

Fig. 17 – 512 Hz time Graph

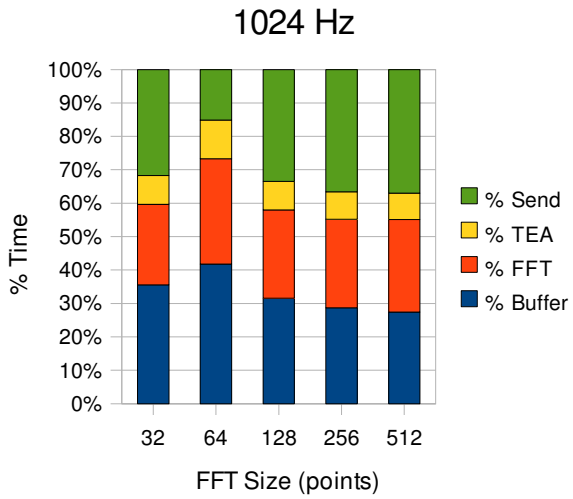

Fig. 18 – 1024 Hz time Graph

The collected results are divided into three sections and an analysis of the observed trend is given for each case. The buffering time readings were collected beginning at 32Hz and the sampling rate was increased gradually until any discrepancy

was observed.

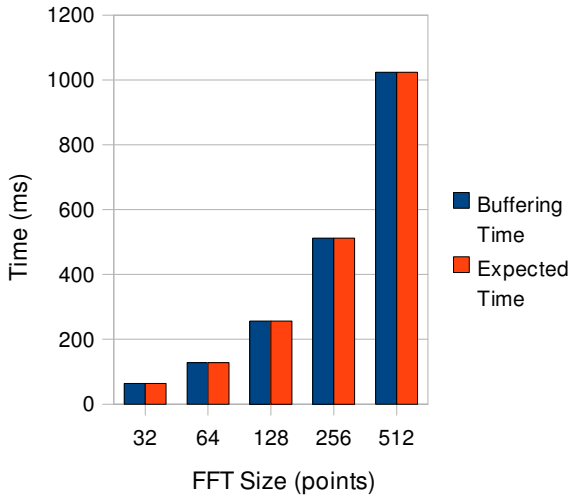

Fig. 19a – Buffering time Graph at 512Hz

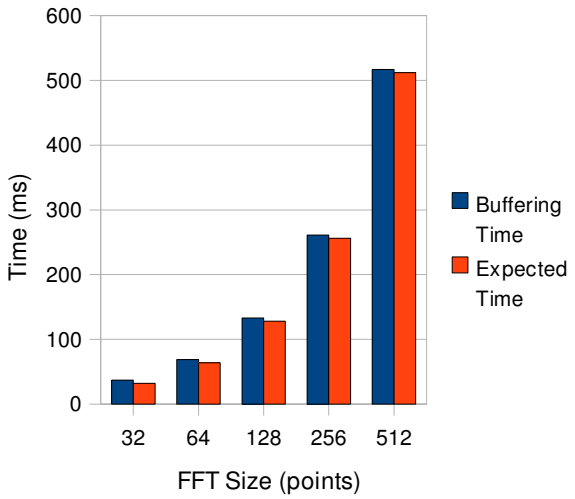

Fig. 19b – Buffering time Graph at 1024Hz

Buffering Time

This is the amount of time it takes for the application to fill the intermediate buffers. The recorded buffering time matched the expected times for the observed frequencies from 32Hz to 512Hz; Figure 19a shows the expected and actual buffering

times for different FFT sizes at 512Hz. However, there was an increase in the recorded time at 1024Hz as shown in Figure 19b. This increase in time is attributed to the introduction of an overhead, however, the source of the overhead is yet to be properly identified.

Task Execution Time

This is the amount of time it takes for the FFT and TEA algorithms to complete. It was found that these values remained the same regardless of the data sampling rate. An increase in the execution time was only observed when the FFT size was increased. This increase in time followed the expected trend, NLog(N), for FFT computations where N is the length of the input buffer. As seen in Figure 20, the percentage of time spent performing the tasks increases as the sampling frequency increased. This is attributed to the fact that an increase in the sampling frequency decreases the buffering time significantly, which was a major portion of the total execution time. The amount of data that was processed on the sensor supports the idea of a decentralized SHM system. Implementing a full fledged decentralized SHM system may not be practical at the moment due to power supply constraints, but the idea still remains promising as the future of SHM systems.

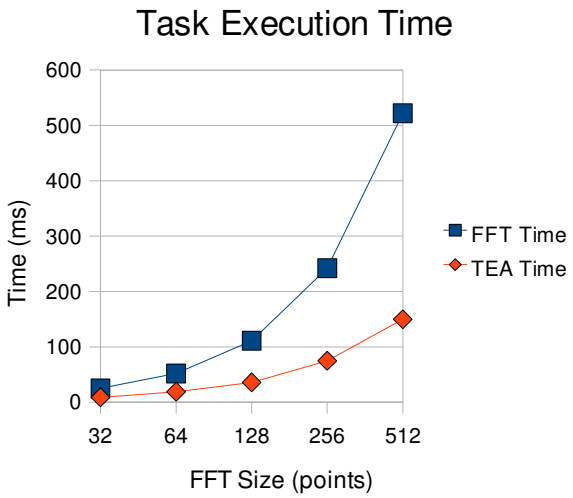

Fig. 20 – Task Execution Graph

Data Transmission Time

This is the time it takes to send data across the network from the sensor mote to the base station. The IRIS sensor mote is capable of sending data at 250 kbps. The table below shows the expected transfer times for the various FFT sizes used during this experiment. These data show the disparity between the expected transmission times and the actual measured times. However, the expected transmission times are simply theoretical, and they only represent a maximum threshold.

Table 1 – Expected Data Transmission Time

FFT Size	Data Size (Bits)	Expected Transfer time (ms)
32	512	2.048
64	1024	4.096
128	2048	8.192
256	4096	16.384
512	8192	32.768

Figure 21 shows the amount of time it took to send multiple packets of data across the network at 256Hz. This trend is expected because the transmission time should be directly proportional to the size of the data. The transmission time information can also be used to calculate the number of nodes that can be supported by the network as discussed in the next paragraph.

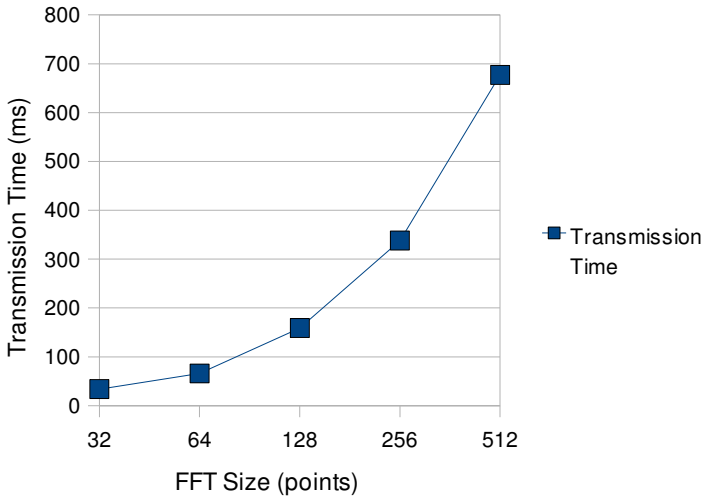

Fig. 21 – Transmission Time as a function of FFT Size at 256Hz

The number of nodes that can be supported by a one-hop network with continuous data transmission depends on the sampling frequency. This is as a result of the dependency between the sampling frequency and the buffering time. To estimate the number of nodes that can be supported by the implementation, the total execution time for each data transmission batch will have to be divided by the amount of time it takes to send the same batch of data over the radio:

$$(Time_{TOTAL}) / (Time_{SEND})$$

Figure 22 shows the average number of nodes that can be supported by the network for the different sampling frequencies that were monitored. It is also important to mention that these values are approximations because only one sensor was used for this experiment; a multi-node network will have to be adjusted to avoid collisions during data transmission.

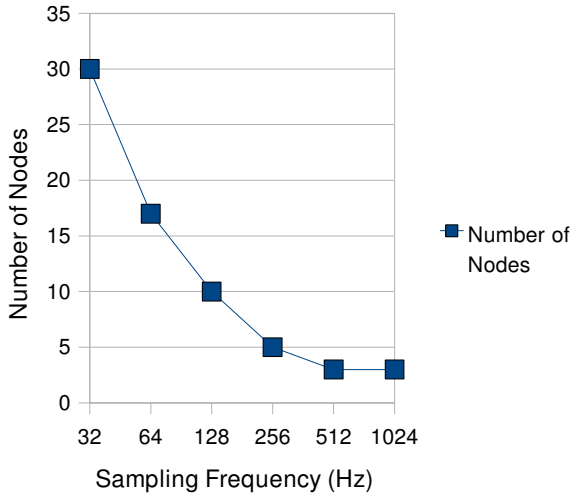

Fig. 22 – Number of Nodes for different Sampling Frequencies

Memory Utilization

One major setback for the IRIS sensor is the amount of memory (RAM) available for programming. Although its processing power is quite good, the sensor is unable to deal with large data sets mainly because of the amount of space needed to store intermediate data.

For this experiment, the largest data set that fits the memory restraint was 512 points of data. The RAM space utilized by the program is shown below for the different FFT sizes. An attempt to process a 1024-point FFT resulted in a program size that exceeded the amount of space available on the sensor (8Kilobytes).

Table 2 – Memory Utilization

FFT Size	RAM (bytes)	Marginal Memory Increase	ROM (bytes)
16	814	N/A	17890
32	974	160	17892
64	1294	320	17892
128	1934	640	17892
256	3214	1280	17898
512	5774	2560	17898
1024	10894	5120	17898

The Memory Utilization graph shows the amount of space used by the program as the number of packets increased. The important piece of information is the trend that is observed when the marginal increase is observed. This trend follows that of the aggregate

packet size. The measurements were taken starting from a 16-point FFT, and the size was doubled for subsequent measurements. This is why the amount of additional space required was expected to increase in multiples of 2 as shown in Figure 23.

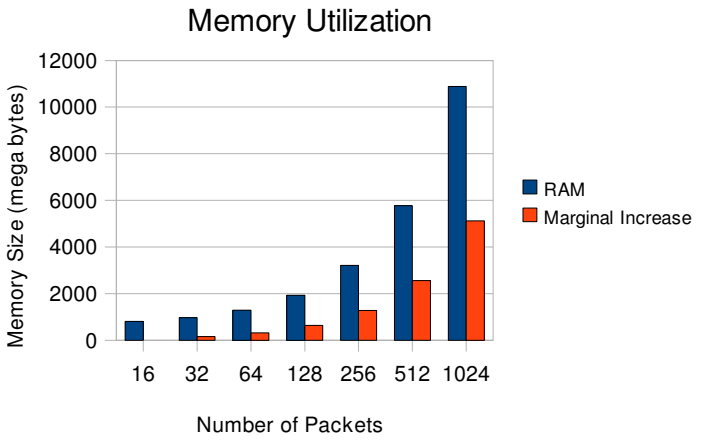

Fig. 23 – Memory Utilization

It is possible to process larger FFT sizes; however, a trade off with computational accuracy will be necessary especially when computing Fast Fourier Transforms. Floating point FFT's require the input data to be stored as a “double” variable which is represented by 8 bytes as compared to the 2 bytes that are used for transferring data over the radios; the values transmitted over the radio are of the type “nx_int16_t”. This is the main reason for the large amount of memory space consumed by the program

Continuous Data Sampling

In order for this application to sample data continuously, the data sample collection and sampling will have to be implemented in parallel using interrupts. This can be accomplished by adding another buffer to hold intermediate data samples that will be processed by the tasks while the initial buffer is being refilled by the sampling protocol. However, this approach can only be feasible as long as the task processing time remains less than the data collection time. The equation "(1/sampling frequency) x (# of samples)" provides the sampling time. This value can then be compared to the sum of the task processing time and the transmission time.

It is important to notice that the formula above translates to a 50% threshold for the system that was implemented in this work. This means that data will have to be sampled no less than 50% of the entire execution time. Figures 13 – 18 show that sampling at approximately 512 Hz and above will result in a case where the data collection time drops below this 50% threshold.

CHAPTER VIII

SUMMARY

The idea of a real-time application depends on the definition of “real-time”. It is commonly used to describe applications that have a deadline associated with them. There may be a requirement for the amount of time allowed to pass before either data is being recaptured, or a system response is initiated. The IRIS mote is ideal for real-time decentralized structural monitoring because its processing power is sufficient for performing computationally intensive and repetitive operations such as FFTs.

When compared to a Mica sensor, the IRIS sensor spends much less time transmitting data over the radio. When sampling between 200 Hz and 250 Hz, data transmission accounts for about 20% of the total execution time on the IRIS as opposed to a staggering 85.7% for the Mica mote.

Although the availability of memory is a major setback for on-chip processing, the available memory space allows for up to 512 data points to be captured at relevant sampling frequencies. The work in this thesis also shows a successful implementation of both a secure data encryption algorithm and the calculation of a 1024-point FFT algorithm on the IRIS sensor.

Finally, a range for feasible computation was determined to be between 32 Hz and 1024 Hz. Although most structures have natural frequency of less than 10 Hz, it is important to sample at a rate much higher than 10 Hz in order to preserve the fidelity of

the signal, as well as eliminate problems caused by noise.

This project can be extended by deploying a WSN on a small scale bridge prototype in order to take actual readings. The characterization can also be extended to measure stress,strain and temperature variations. Also more work can be done in terms of programming the sensor nodes to transfer data for predetermined threshold values.

REFERENCES

1. Minnesota Department of transportation, Program Support Division, Technical memorandum No. 07-10-B-02, July 19, 2007

2. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “A Survery on Sensor Networks,” IEEE Comm. Mag., Aug. 2002, pp. 102-114

3. D. Culler, D. Estrin, and M. Srivastava, "Guest Editors' Introduction: Overview of Sensor Networks," Computer, vol. 37, pp. 41-49, 2004.

4. C. E. Shannon, "Communication in the presence of noise", Proc. Institute of Radio Engineers, vol. 37, no.1, pp. 10-21, Jan. 1949.

5. N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. The Proceedings of the ACM Conference on Embedded Networked Sensor Systems, November 2004.

6. Sukun Kim, Pakzad, S. Culler, D. Demmel, J. Fenves, G. Glaser, S. Turon, M., "Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks," Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on , vol., no., pp.254-263, 25-27 April 2007

7. N. G. Shrive. “Intelligent Structural Health Monitoring: a Civil Engineering Perspective”. University of Calgary, Alberta Canada.

8. www.xbow.com/support/Support_pdf_files/MTS-

MDA_Series_Users_Manual.pdf

9. www.xbow.com/Products/Product_pdf_files/Wireless_pdf/IRIS_Datasheet.pdf
10. www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB520_Datasheet.pdf
11. http://www.tinyos.net/
12. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in Ambient Intelligence: Springer Berlin Heidelberg, 2005, pp. 115-148.
13. Cooley, J. W. and Tukey, J. W. (1965). An Algorithm for the Machine Calculation of Complex Fourier Series. Math. Computat., 19, 297–301.
14. www.cmlab.csie.ntu.edu.tw/.../transform/fft.html
15. www.cmlab.csie.ntu.edu.tw/.../transform/fft.html
16. TEA, a Tiny Encryption Algorithm
17. C. Karlof, N. Sastry, and D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks," in 2nd International Conference on Embedded Networked Sensor Systems Baltimore, MD, USA, 2004, pp. 162 – 175.
18. http://en.wikipedia.org/wiki/File:XTEA_InfoBox_Diagram.png
19. Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). Numerical recipes in C: the art of scientific computing. Cambridge University Press, Cambridge, England.

20. Straser, E.G. “A modular, wireless damage monitoring system for structures”. Ph.D. Thesis, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA. 1998

21. K. C. Lu, C. H. Loh, Y. S. Yang, J. P. Lynch and K. H. Law. “Real-Time structural damage detection using Wireless Sensing and Monitoring System” (2007)

22. S. E. Lee, S.H. Shin, G. D. Park, K. Y. Yoo. “Wireless Sensor Network Protocols for Secure and Energy-Efficient Data Transmission”. 7th Computer Information Systems and Industrial Management Applications, 2008

23. Health Monitoring of Bridge Structures and Components Using Smart Technology http://www.ctre.iastate.edu/reports/health_monitor_wi_vol1.pdf

24. V. Plessi, F. Bastianini, S. Sedigh. “A Wireless system for real-time environmental and structural monitoring”. University of Missouri-Rolla, Rolla, MO

25. http://www.bec.iastate.edu/

26. J.P. Lynch. Overview of wireless sensors for real-time health monitoring of civil structures. Proceedings of the 4th International Workshop on Structural Control (4th IWSC), New York City, NY, June 10-11, 2004.

27. M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. The Proceedings of ACM Second International Conference on Embedded Networked Sensor Systems (SenSys 04), pp. 39-49, Baltimore, MD, November 3, 2004.

28. S. Liu, O. V. Gavrylyako, P. G. Bradford. Implementing the TEA on Sensors.

ACM Southeast Regional Conference, Proceedings of the 42nd annual Southeast regional conference, pp. 64 – 69, Huntsville, AL, 2004.

APPENDIX

Oscilloscope.h

```
/*
 * Copyright (c) 2006 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

// @author David Gay
// Revised by Bode Ajiboye

#ifndef OSCILLOSCOPE_H
#define OSCILLOSCOPE_H

enum {
  /* Number of readings per message. If you increase this, you may have to
     increase the message_t size. */
```

```
  NREADINGS = 8,

  /* Default sampling period. */
  DEFAULT_INTERVAL = 1,

  AM_OSCILLOSCOPE = 0x93,
  packets = 4
};

typedef nx_struct oscilloscope {
  nx_int16_t version; /* Version of the interval. */
  nx_int16_t interval; /* Samping period. */
  nx_int16_t id; /* Mote id of sending mote. */
  nx_int16_t count; /* The readings are samples count * NREADINGS onwards */

  //timing variable
  nx_int32_t timer;

  nx_int16_t readings[NREADINGS];
} oscilloscope_t;

#endif
```

Oscilloscope.nc (Original)

```
/*
 * Copyright (c) 2006 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/**
 * Oscilloscope demo application. See README.txt file in this directory.
 *
 * @author David Gay
 */
#include "Timer.h"
#include "Oscilloscope.h"

module OscilloscopeC
{
  uses {
```

```
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Leds;
  }
}
implementation
{
  message_t sendBuf;
  bool sendBusy;

  /* Current local state - interval, version and accumulated readings */
  oscilloscope_t local;

  uint8_t reading; /* 0 to NREADINGS */

  /* When we head an Oscilloscope message, we check it's sample count. If
     it's ahead of ours, we "jump" forwards (set our count to the received
     count). However, we must then suppress our next count increment. This
```

```
   is a very simple form of "time" synchronization (for an abstract
   notion of time). */
bool suppressCountChange;

// Use LEDs to report various status issues.
void report_problem() { call Leds.led0Toggle(); }
void report_sent() { call Leds.led1Toggle(); }
void report_received() { call Leds.led2Toggle(); }

event void Boot.booted() {
  local.interval = DEFAULT_INTERVAL;
  local.id = TOS_NODE_ID;
  if (call RadioControl.start() != SUCCESS)
    report_problem();
}

void startTimer() {
  call Timer.startPeriodic(local.interval);
  reading = 0;
}

event void RadioControl.startDone(error_t error) {
```

```
  startTimer();
}

event void RadioControl.stopDone(error_t error) {
}

event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  oscilloscope_t *omsg = payload;

  report_received();

  /* If we receive a newer version, update our interval.
     If we hear from a future count, jump ahead but suppress our own change
  */
  if (omsg->version > local.version)
    {
        local.version = omsg->version;
        local.interval = omsg->interval;
        startTimer();
    }
  if (omsg->count > local.count)
    {
```

```
        local.count = omsg->count;
        suppressCountChange = TRUE;
    }

  return msg;
 }

 /* At each sample period:
   - if local sample buffer is full, send accumulated samples
   - read next sample
 */
 event void Timer.fired() {
  if (reading == NREADINGS)
    {
        if (!sendBusy && sizeof local <= call AMSend.maxPayloadLength())
          {
            // Don't need to check for null because we've already checked length
            // above
            memcpy(call AMSend.getPayload(&sendBuf, sizeof(local)), &local, sizeof
local);
            if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf, sizeof local) ==
SUCCESS)
              sendBusy = TRUE;
```

```
        }
      if (!sendBusy)
        report_problem();

      reading = 0;
      /* Part 2 of cheap "time sync": increment our count if we didn't
         jump ahead. */
      if (!suppressCountChange)
        local.count++;
      suppressCountChange = FALSE;
    }
  if (call Read.read() != SUCCESS)
    report_problem();
}

event void AMSend.sendDone(message_t* msg, error_t error) {
  if (error == SUCCESS)
    report_sent();
  else
    report_problem();

  sendBusy = FALSE;
```

```
  }

  event void Read.readDone(error_t result, uint16_t data) {
    if (result != SUCCESS)
      {
        data = 0xffff;
        report_problem();
      }
    local.readings[reading++] = data;
  }
}
```

Oscilloscope.nc (Modified)

```
/*
 * Copyright (c) 2006 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/**
```

```
 * Oscilloscope demo application. See README.txt file in this directory.
 *
 * @author David Gay
 * Revised by Bode Ajiboye
 */
#include "Timer.h"
#include "Oscilloscope.h"

module OscilloscopeC @safe()
{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Leds;

    interface Timer<TMilli> as Timer2;
    interface LocalTime<TMilli>;
```

```
  }
}
implementation
{
  message_t sendBuf;
  bool sendBusy;


//"lc" is used to check for first sampling iteration
uint8_t lc = 1;


  /* Current local state - interval, version and accumulated readings */
  oscilloscope_t local;

  uint32_t reading; /* 0 to NREADINGS */
  uint16_t timing; /* timer variable*/
  uint32_t start, end,overflow_check;


 /* Buffer control variable
  when index2 = 1, the send sequence is initiated
```

```
 index3 is used to track the number of packets that have been sent*/

uint8_t index = 0;

uint8_t jj,ii;

uint8_t index2 = 0;



/* index3 is used to control data accumulation into "local.reading" variable prior to
sending */

uint8_t index3 = 0;



/*buf[] is an array used to hold temporary sensor readings... "packets" is used to set the
number of packets to be stored/sent

  "packets" is declared in header file      */



nx_int16_t buf[NREADINGS*packets];

nx_int16_t buf2[NREADINGS*packets];



/* buf_index is used to track the section of buf[] that the new readings from local.reading
need to go into
. For example, if buf[] has "NREADINGS*2" positions, the first stream of data from
local.reading fills the first NREADINGS positions,
 when buf_index is 0. Incrementing buf_index by one allows the next set of data from
local.reading to be place in the next NREADINGS positions
 in buf[] */



uint32_t buf_index,d;
```

```
/* bcomp is set to TRUE when the number of packets specified by the variable "packets"
have been sent over the radio  */

 bool bcomp = FALSE;

 bool flag = FALSE;



        uint8_t dir;

        uint16_t m;

/* FFT variables */



           long n,i,i1,j,k,i2,l,l1,l2;

           double c1,c2,tx,ty,t1,t2,u1,u2,z;

           double x[NREADINGS*packets];

           double y[NREADINGS*packets];



           uint16_t ind;



/*tea algorithm variables*/

           uint16_t v0, v1, sum, loopi;

           uint16_t delta=0x9e3779b9;                  /* a key schedule constant */

           uint16_t k0, k1, k2, k3;   /* cache key */

           uint16_t kk[4];



           uint16_t tealoop = 0;
```

```
//used to calculate m=log_2(NREADINGS*packets)
  int p = 1;


task void tea(){


tealoop = 0;
//while(tealoop < 64)
while(tealoop < packets*NREADINGS)
{
   kk[0] = 0x1234;
   kk[1] = 0x5678;
   kk[2] = 0x90AB;
   kk[3] = 0xCDEF;


   v0 = buf2[tealoop],v1 = buf2[tealoop+1], sum = 0;




   k0=kk[0], k1=kk[1], k2=kk[2], k3=kk[3];
```

```
  for (loopi=0; loopi < 32; loopi++) {                    /* basic cycle start */
     sum += delta;
     v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
     v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
  }                                          /* end cycle */
  buf2[tealoop]=v0; buf2[tealoop+1]=v1;
  tealoop = tealoop + 2;
}

//local.timer = call LocalTime.get(); //used to get execution time for TEA algorithm
index2 = 1;


}

task void fft(){
        for (d=0; d < NREADINGS*packets; d++)
                {

                        buf2[d] = 10*sin(20*(d/packets)); //generate a sine signal.;
                    x[buf_index] = buf2[buf_index];

                }
```

```
//Calculate m=log_2(NREADINGS*packets)
m = 0;

  while(p < NREADINGS*packets)
  {
    p *= 2;
    m++;
  }

        dir = 1;

        /* Calculate the number of points */
        n = 1;
        for (i=0;i<m;i++)
           n *= 2;

        /* Do the bit reversal */
        i2 = n >> 1;
```

```
j = 0;
for (i=0;i<n-1;i++) {
  if (i < j) {
    tx = x[i];
    ty = y[i];
    x[i] = x[j];
    y[i] = y[j];
    x[j] = tx;
    y[j] = ty;
  }
  k = i2;
  while (k <= j) {
    j -= k;
    k >>= 1;
  }
  j += k;
}

/* Compute the FFT */
c1 = 1.0;
   c1 = -c1;
c2 = 0.0;
```

```
l2 = 1;
for (l=0;l<m;l++) {
  l1 = l2;
  l2 <<= 1;
  u1 = 1.0;
  u2 = 0.0;
  for (j=0;j<l1;j++) {
    for (i=j;i<n;i+=l2) {
      i1 = i + l1;
      t1 = u1 * x[i1] - u2 * y[i1];
      t2 = u1 * y[i1] + u2 * x[i1];
      x[i1] = x[i] - t1;
      y[i1] = y[i] - t2;
      x[i] += t1;
      y[i] += t2;
    }
    z =  u1 * c1 - u2 * c2;
    u2 = u1 * c2 + u2 * c1;
    u1 = z;
  }
  c2 = sqrt((1.0 - c1) / 2.0);
  if (dir == 1)
```

```
        c2 = -c2;
      c1 = sqrt((1.0 + c1) / 2.0);
    }

    /* Scaling for forward transform */
    if (dir == 1) {
      for (i=0;i<n;i++) {
        x[i] /= n;
        y[i] /= n;
      }
    }
```

```
for (ind=0; ind < NREADINGS*packets ; ind++)
        {
            buf2[ind] = x[ind];

        }
```

```
//local.timer = call LocalTime.get(); **used to get execution time for FFT algorithm
```

```
index2=1;

//post tea();

}



 /* When we head an Oscilloscope message, we check it's sample count. If
    it's ahead of ours, we "jump" forwards (set our count to the received
    count). However, we must then suppress our next count increment. This
    is a very simple form of "time" synchronization (for an abstract
    notion of time). */
 bool suppressCountChange;

 // Use LEDs to report various status issues.
 void report_problem() { call Leds.led0Toggle(); }
 void report_sent() { call Leds.led1Toggle(); }
 void report_received() { call Leds.led2Toggle(); }

 event void Boot.booted() {
```

```
  local.timer = 0;
  local.interval = DEFAULT_INTERVAL;
  local.id = TOS_NODE_ID;
  if (call RadioControl.start() != SUCCESS)
    report_problem();
}


void startTimer() {
  call Timer.startPeriodic(local.interval);
  call Timer2.startPeriodic(2);
  reading = 0;

}

event void RadioControl.startDone(error_t error) {
  startTimer();
}

event void RadioControl.stopDone(error_t error) {
}
```

```
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  oscilloscope_t *omsg = payload;

  report_received();

  /* If we receive a newer version, update our interval.
     If we hear from a future count, jump ahead but suppress our own change
  */
  if (omsg->version > local.version)
    {
        local.version = omsg->version;
        local.interval = omsg->interval;
        startTimer();
    }
  if (omsg->count > local.count)
    {
        local.count = omsg->count;
        suppressCountChange = TRUE;
    }

  return msg;
```

```
  }



event void Timer2.fired(){







}



 /* At each sample period:
    - if local sample buffer is full, send accumulated samples
    - read next sample
 */
 event void Timer.fired() {



   if (index2 == 1)
     {

                    index2 = 0;
```

```
		for (jj=0; jj < NREADINGS; jj++)

		{

			buf_index = jj + (index3*NREADINGS);

			local.readings[jj] = buf2[buf_index];



		}

		index3++;




		if (!sendBusy && sizeof local <= call AMSend.maxPayloadLength())

		{

			// Don't need to check for null because we've already checked
length

			// above




			memcpy(call AMSend.getPayload(&sendBuf, sizeof(local)),
&local, sizeof local);

			if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf,
sizeof local) == SUCCESS)

				sendBusy = TRUE;



		}
```

```
            if (!sendBusy)
            report_problem();

            /* Part 2 of cheap "time sync": increment our count if we didn't
            jump ahead. */
            if (!suppressCountChange)
            local.count++;
            suppressCountChange = FALSE;
  }
if(!flag)
      {



            if (call Read.read() != SUCCESS)
            {

                  report_problem();

            }
      }
```

```
  }


  event void AMSend.sendDone(message_t* msg, error_t error) {
    if (error == SUCCESS)
      report_sent();
    else
      report_problem();


    sendBusy = FALSE;


          if(!bcomp)
          {
                  for (jj=0; jj < NREADINGS; jj++)
                  {
                          buf_index = jj + (index3*NREADINGS);
                          local.readings[jj] = buf2[buf_index];
                  }



              memcpy(call AMSend.getPayload(&sendBuf, sizeof(local)), &local, sizeof
local);
```

```
            if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf, sizeof local) ==
SUCCESS)

              sendBusy = TRUE;



            if(index3 == packets - 1)

            {

                  index2 = 0;

                  index3 = 0;

                  bcomp = TRUE;

                  timing = 0;

                  flag = FALSE;

            }

            index3++;



          }

 }



 event void Read.readDone(error_t result, uint16_t data) {




   if (result != SUCCESS)

     {

         data = 0xffff;
```

```
        report_problem();
    }

                buf2[reading++] = data;



        if(lc == 1)
                {
                        local.timer = call LocalTime.get();
                        lc = 0;
                }


if(reading == ((NREADINGS*packets)+1))
  {
        local.timer = call LocalTime.get() - local.timer;
        reading = 0;
        bcomp = FALSE;
     index2=1;
        lc = 1;
```

```
            flag = TRUE;
            //post fft();



        }


    }
}
```

OscilloscopeAppC.nc

```
/*
 * Copyright (c) 2006 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/**
```

```
 * Oscilloscope demo application. Uses the demo sensor - change the
 * new DemoSensorC() instantiation if you want something else.
 *
 * See README.txt file in this directory for usage instructions.
 *
 * @author David Gay
 */
configuration OscilloscopeAppC { }
implementation
{
  components OscilloscopeC, MainC, ActiveMessageC, LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE), new
AMReceiverC(AM_OSCILLOSCOPE);


  OscilloscopeC.Boot -> MainC;
  OscilloscopeC.RadioControl -> ActiveMessageC;
  OscilloscopeC.AMSend -> AMSenderC;
  OscilloscopeC.Receive -> AMReceiverC;
  OscilloscopeC.Timer -> TimerMilliC;
  OscilloscopeC.Read -> Sensor;
  OscilloscopeC.Leds -> LedsC;
}
```

Perl.pl

```
#!/usr/local/bin/perl

print "Hi there!\n";


open (LOGFILE, "output.txt") or die "I couldn't get at log.txt";

open (PROCESSED, ">processed.txt") or die "$! error trying to overwrite";



print PROCESSED "Mote Data\n";

for $line (<LOGFILE>) {


@fields = split(/\s/, $line);



print PROCESSED "TIMER VALUE = ";

print PROCESSED hex("$fields[16]$fields[17]$fields[18]$fields[19]") ;

print PROCESSED "\n";

;


for ($count=20; $count<35; $count++)

{
```

```
 $hexval1 = $fields[$count];
 $hexval2 = $fields[$count+1];

if(hex($hexval1) == 255)
 {
        print PROCESSED hex("$fields[$count]$fields[$count+1]") - 65536;
 }
else
 {
        print PROCESSED hex("$fields[$count]$fields[$count+1]");
 }


 print PROCESSED "\n";
 $count++;
}

}
close LOGFILE;
close PROCESSED;
```