Small Wind Turbine Blade Sensor for Condition Based Maintenance

A Thesis

presented to

the Faculty of California Polytechnic University

San Luis Obispo

in Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

CHUN KAU CHEUNG

Sep 2011

© 2011

Chun Kau Cheung

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Small Wind Turbine Blade Sensor for
Condition Based Maintenance

AUTHOR: CHUN KAU CHEUNG

DATE SUBMITTED: Sep 2011

COMMITTEE CHAIR: John Ridgely, Ph.D.

COMMITTEE MEMBER: Joseph Mello, Ph.D.

COMMITTEE MEMBER: Eileen Rossman, M.S.

Abstract

Wind Energy is a fast growing industry. However, wind turbines are expensive to build, so they need proper maintenance to prolong service life. Since blades are the most important components in the wind turbine, this project prototyped a low cost sensing system to monitor blades' health. The sensor board uses Atmel microcontrollers to monitor the blade's vibration frequency by sampling a data set of 128 acceleration measurements. An integer version of Fast Fourier Transform is used to process those data. The result is then sent via radio wirelessly to a receiver for future processing. Through calculation with collected data, theoretically, the system can run for 30+ years with lithium batteries if the accelerometer on the microcontroller board is replaced. The system has been tested on a vibration shaker, and it is also tested in single disturbance with wood and aluminum beam.

Table of Contents

	Page
List of Tables	ix
List of Figures	x
Definition and Terms	xi
Chapter	
1 Introduction	1
1.1 Project Statement	1
1.2 Project Aim	2
1.3 Project Scope	2
1.4 Project Organization	3
2 Background	4
2.1 Wind Turbine	4
2.1.1 Vertical Axis Wind Turbine	5
2.1.2 Horizontal Axis Wind Turbine	5
2.2 Wind Turbine Failure	6
2.2.1 Critical Components	6
2.3 Maintenance	8
2.3.1 Reactive Maintenance	9
2.3.2 Scheduled Maintenance	10
2.3.3 Condition Based Maintenance	11
2.3.4 Condition Based Maintenance with degradation model	11
2.4 Cal Poly Wind Turbine Project	12
2.5 Composites	12
2.5.1 Common Composite Blade Failure Modes	13
2.5.2 Frequency of Oscillation and Residual Life of Blade	14

2.6	Microcontroller	14
2.6.1	Atmel Microcontroller	14
2.7	Accelerometer	16
2.8	Computer Programming	17
2.9	Fourier Transform	17
2.9.1	Methods of Fourier Transform	18
2.9.2	Integer Fast Fourier Transform	19
2.9.3	Fast Fourier Transform Limitations	21
2.9.4	Fourier Transform Trick	21
2.10	Energy Source	22
2.10.1	Non-rechargeable Battery	23
2.10.2	Solar Power	23
2.10.3	Vibration	24
3	Theory and Process Layout	26
3.1	Failure Detection	26
3.1.1	Natural Frequency	27
3.1.2	Frequency of Oscillation	27
3.1.3	Failure Criteria	28
3.2	Finding Frequency of Oscillation	28
3.2.1	Acceleration to Fourier Transform	29
3.2.2	Integer Fast Fourier Transform	30
3.3	Accuracy of IFFT	30
3.3.1	Improving Accuracy	31
3.4	Communication	32
3.4.1	24L01	32
3.5	Power Supply	33
3.5.1	Supplying the Power	33
3.5.2	Managing the Power Usage	34

3.5.3	Using L91 battery	40
4	Programming	42
4.1	Fast Fourier Transform	42
4.1.1	Preliminary Programming Decision	42
4.1.2	Getting the Fast Fourier Transform to work	43
4.1.3	Getting the Integer Fast Fourier Transform to work	43
4.2	Embedded System Programing	44
4.2.1	CBM_sensor	45
4.2.2	CBM_receiver	45
4.3	Serial Port Terminal Program	46
5	Manufacturing	47
5.1	Cables and Connector Cables	47
5.1.1	Programming Cable	47
5.1.2	Radio Connector	47
5.1.3	Current Tester	48
5.2	Circuits	48
5.2.1	LED-button	49
5.2.2	Wein Bridge	49
6	Testing and Result	51
6.1	Test plan	51
6.1.1	Continuous Vibration Test	51
6.1.2	Disturbance Test- Metal Bar and Wood Bar	54
6.1.3	Continuous Sine Wave Test	56
6.1.4	Current Test	58
6.2	Results	59
6.2.1	Continuous Vibration Test	60
6.2.2	Disturbance Test- Metal Bar and Wood Bar	62
6.2.3	Continuous Sine Wave Test	63

6.2.4	Current Test	63
6.3	Uncertainty of the Testing	65
7	Conclusion	66
8	Unresolved Problems and Upgrades	67
8.1	Unresolved Problems	67
8.2	Upgrades	67
8.2.1	Programming	67
8.2.2	Mechanical Construction	68
8.2.3	Power Consumption	68
	References	70
Appendix		
A	Battery Comparison	72
A.1	Battery Specification	72
A.2	Weight	72
A.3	Current Usage	73
A.4	Self Discharge Rate	74
B	User Manual	76
B.1	Wireless Output	76
B.2	Control	79
C	Programming Code Description	80
C.1	CBM Sensor	80
C.2	CBM Receiver	89
D	Programming Code	91

List of Tables

6.1	Result from Continuous Vibration Test- Part 1	60
6.2	Result from Continuous Vibration Test- Part 2	60
6.3	Result from Disturbance Test	61
6.4	Summary from Disturbance Test	62
6.5	Result from Continuous Sine Wave Test	63
6.6	Minimum current of microcontroller for Swoop board that has an accelerometer and 4MHz crystal during power save mode	64
6.7	Minimum current of microcontroller for Swoop Board that does not have accelerometer and has 10MHz crystal during power save mode .	64
6.8	Current usage with microcontroller with 4MHz crystal	65
A.1	Batteries Comparison	73
B.1	MCU radio output at different stages	77

List of Figures

1.1	World total installed capacity of wind turbines in recent years	1
2.1	Vertical Axis Wind Turbine	4
2.2	Horizontal Axis Wind Turbine	5
2.3	Tower Collapse	7
2.4	Evolution of Maintenance Policy	9
2.5	$S - N$ Curve for AS4/3501-6 carbon/epoxy laminate	13
5.1	JTAG and ISP cable	48
5.2	Radio connector for the ET128 board	48
5.3	Current tester for ET128 board and Swoop board	49
5.4	LED button for troubleshooting	49
5.5	The actual Wein Bridge circuit	50
6.1	Set up for the Continuous Vibration Test	53
6.2	Set up for the Disturbance Test	55
6.3	Microcontroller with Wein Bridge attached	57
6.4	Microcontroller measuring current draw	59
A.1	Discharge curve of L91 lithium battery at 1ma	74
A.2	Shelf life of L91 with different temperature	75

Definitions and Terms

The following terms are defined as follows for use in this document;

- Atmel - A company that makes microcontrollers
- Bits - A binary digit that has value of either 1 or 0
- 8-bit - 8 binary digits. Used to represent values from 00000000 to 11111111 (0 to 255)
- 64-bit - 64 binary digits. Used to represent value from 0 to 18446744073709551615
- 8-bit processor - A processor that process 8-bit data at a time
- Clock cycle - A term in computer and microcontroller, refers to a duration of time it takes for a frequency generating element in the computer or microcontroller to oscillate a full period
- Crystal - A clock reference device in the microcontroller board, it is used to control how fast a microcontroller processes data
- Accelerometer- A device that measures acceleration and converts the acceleration value into electrical signals
- Look up table - A table that contains needed values that are all pre-calculated, so no extra calculation process is needed when using the look up table instead of computing numbers such as trigonometric functions
- Self discharging rate- A term from battery, it means the rate of battery loses charge by itself over time

- Analog to Digital Converter - A peripheral of a microcontroller that converts a continuous voltage reading to a discrete value that the microcontroller needs for processing
- ET128 Board- One model of microcontroller board that contains an ATmega128 microcontroller
- Swoop Board- One model of microcontroller board produced at Cal Poly that contains ATmega324P microcontroller
- Quiescent current- The minimum current use in an electronic device

Here are a list of abbreviations in the text

- Cal Poly - California Polytechnic State University- San Luis Obispo
- CBM- Condition Based Maintenance
- WWEA- World Wind Energy Association
- VAWT- Vertical Axis Wind Turbine
- HAWT- Horizontal Axis Wind Turbine
- FT- Fourier Transform
- DFT- Discrete Fourier Transform
- FFT- Fast Fourier Transform
- IFFT- Integer Fast Fourier Transform
- MCU- Microcontroller
- ω_n - Natural Frequency
- ω_{osc} - Frequency of Oscillation

Chapter 1

Introduction

1.1 Project Statement

There are many wind turbines in the world. According to the World Wind Energy Association (WWEA), there are already 160GW ($1.60 * 10^{11}$ Watts) of electricity generating capacity worldwide in 2009 as seen in figure 1.1. The number is predicted to keep increasing for years to come. Wind energy is a clean renewable energy source. However, it also comes with a big price tag. Megawatt class wind turbines cost millions of dollars to make and hours and hours of maintenance to prevent failures. Failures can endanger surrounding lives and the cost of repaying for property damages is often high.

Figure 1.1: World total installed capacity of wind turbines in recent years
World Wind Energy Association 2009.

Measures that reduce the chance of a drastic failure such as a blade broken off and flying to a neighborhood and get more usable life out of wind turbine blades can directly relate to huge cost savings.

The Mechanical Engineering Department of California Polytechnic State University-San Luis Obispo(Cal Poly-SLO) has been on a sequence of projects to build a small scale Wind Turbine which utilizes technology from large Utility scale models. As blades of the wind turbine have been made in 2009 by Brian Edwards, the next step is to look into preventative maintenance for the blades.

1.2 Project Aim

The purpose of this project is to prototype a low cost system that monitors composite blades's vibration in real time. It informs the user when a failure is detected from significant changes of frequency of oscillation in order to enable the user to perform condition based maintenance.

1.3 Project Scope

The project here will investigate how to use a low cost device like a microcontroller to monitor composite wind turbine blades. In order to accomplish the project, there are many fields of research which must be considered.

- Composites fatigue life and properties- Carbon Fiber, Fiberglass and epoxy composites specifically
- Environmental variables that will degrade the performance of the monitoring system
- Programming - microcontroller and computer programing
- Electronic systems -life and operational characteristics

- Frequency detection - Fourier Transforms
- Statistical data filtering
- Circuit and machine part design and fabrication

1.4 Project Organization

The report is arranged in 8 chapters. Each chapter has its own objective.

- Chapter 1 is the introduction that describes the overall project and what is in the report
- Chapter 2 discusses some background researches and some preliminary decisions made based on the researches
- Chapter 3 is an overview of how the problem is tackled in theory
- Chapter 4 describes the major steps in the programming and a description of all the relevant functions of the project
- Chapter 5 describes the components of the system which was designed and the reason why each particular component was made
- Chapter 6 describes the test plans and their results
- chapter 7 is the conclusion that describes how well the project meets the initial goal
- Chapter 8 is a discussion of unresolved problems and some possible improvements suggested for the system.

Chapter 2

Background

Figure 2.1: Vertical Axis Wind Turbine *Motherearthnews.com*

2.1 Wind Turbine

There are two basic types of wind turbines: Vertical Axis Wind Turbine (VAWT, see figure 2.1) and Horizontal Axis Wind Turbine (HAWT, see figure 2.2). Each has different advantages.

Figure 2.2: Horizontal Axis Wind Turbine *scienceprep.org*

2.1.1 Vertical Axis Wind Turbine

Vertical Axis Wind Turbine is basically a wind turbine that spins when wind hits the blades perpendicular to the rotating axis. Because the axis is vertical, the hub is generally installed towards the ground. Due to the location of the hub, it is really hard to install a pole to raise the wind turbine high up in the air. So generally, it is installed close to the ground where the wind is less strong. Overall, VAWT is considered as more aesthetically pleasing to the public compared to HAWT. VAWT has two major advantages over HAWT. One is that since the hub is near ground, it is very easy to service the hub when maintenance is required. Second, it does not require additional positioning to get the most efficient wind angle.

2.1.2 Horizontal Axis Wind Turbine

The design of the Horizontal axis wind turbine has improved over time, so it is a more developed technology. The HAWT spins when wind flow that is parallel to the rotating axis hits the blade. A support tower is a must in order for the blades not to

touch the ground. In addition, in order to get the most energy from the wind with the given amount of wind turbine cost, the tower is usually two to three times longer than the blade. A HAWT usually has a huge area to sweep through, so the shadow of the blades can cause light flickering to nearby neighborhoods. This is one of the reasons why some people don't want to live close to a HAWT. One advantage of the HAWT is that it is usually high up in the air where high velocity wind is located, so it can generate a lot more energy than the VAWT. Second advantage associated with being high up in the air is that it does not have to deal with the turbulent air flow that is caused by immediate environment on the ground, like trees or buildings. The turbulence in the air flow will cause a low efficiency power output from the wind as well as causing a high wear and tear on the system. Since the velocity of the wind in the higher altitude does not vary much, so the load imposed on the blades is near constant, with a small fatigue component, the life of the HAWT can be much longer.

2.2 Wind Turbine Failure

Because all machines will fail over time, so knowing the service life is very important. Typically, wind turbines are designed to last for 20 years. However, the designed life does not mean the same thing as the actual life because components can fail anytime. So the designed life of 20 years is just a guide line in the design process. There are many variables that lead to premature failure. This section will discuss what components are susceptible to failures and the cost of failure.

2.2.1 Critical Components

Wind turbines can fail in many ways. By different reports from a few countries in different sizes and makes of wind turbines, there seems to be a variability in which component fails the most. Below is a list of common components that fail:

Figure 2.3: Tower Collapse *www.stpns.net*

- Blades
- Nacelle and components inside
- Hydraulic/pneumatic system
- Tower
- Sensors
- Control system and electrical system

Among all those components, blades and tower usually receive the most attention because they are the most expensive components to fail. Wind turbine blades alone cause 15-20% of the total wind turbine installation cost[Flemming and Troels, 2003] and are the components that have the highest number of reported accidents¹. The tower collapse can totally destroy the wind turbine (see figure 2.3). Extra care is needed to minimize their failure rate and the ‘unwanted cost’ (see section 2.2.1) associated.

¹There are 199 reported accidents from the year 1990 to June of 2010 [Brenda, 2010], the common failure modes of the blades will be discussed in the section 2.5.1.

Reliability of sensors is also important as well. According to a report from Germany that studied 15 years of wind turbine operation (1500 wind turbines total), sensor malfunction accounts for 10% of the system malfunctions[Ciang et al., 2008], more than any single mechanical system malfunction listed in their report. Sensors let the user know when other components do not operate as intended. If a sensor fails, some important components will not be monitored. If a sensor repeatedly false triggers, wind turbine downtimes will occur. All these failures will not be cost efficient for the company.

Unwanted Cost

In a wind turbine, some physical component failures or a part breaks off may alter the balance of the system and cause another component to fail or breaks off in a chain reaction. For example, if a blade breaks off, it will cause immediate system imbalance. The force from the imbalance usually is many times over what the system is designed for, which may lead to other components breaking off due to overload altogether. The repair may involve blades, generator, or even a tower replacement, and those items are very expensive. In addition, the broken off component may result in danger to the surrounding. A broken blade traveled as far as 1300 meters away during a storm when the brake failed to operate[Brenda, 2010]. Things within this range could be damaged and life could be easily taken with such an impact. The cost of legal issues and system downtime may end up to be much more than the system's original cost.

2.3 Maintenance

After a wind turbine is built and in service, the concern now is how to optimize the cost to get the most out of wind energy. As discussed before, the designed life of wind turbines is about 20 years. This does not mean all parts will fail at the tick mark of

20 years; many parts will still be functioning for years after. It is best to know what needs to be overhauled and what needs to be replaced to optimize the cost. This is when maintenance becomes important. There are four basic maintenance strategies that are evolved over time to optimize the cost of wind energy. A diagram of the progress is shown in figure 2.4.

Figure 2.4: Evolution of Maintenance Policy [Ambani et al., 2009]

2.3.1 Reactive Maintenance

This maintenance methodology is basically after a machine is installed, no intervention will be done until it fails. This is the maintenance routine for many machines in the pre-1960s era [Ambani et al., 2009]. The only time to service the machines or replace them is when the machine is not functioning as designed. It could be a minor problem, or it could be a huge problem, whether it is a loose wire that requires a bolt tightening, or a blade broken and flew off to a nearby neighborhood.

The advantage of this maintenance is that only the installation cost is required initially. No more additional cost associated with maintenance is needed until something fails. The downside to this maintenance is that when the machine fails, it may cause a really long system downtime for it to be repaired. The reason for a longer

downtime than scheduled maintenance is that the unscheduled downtime sometimes involves failing of parts that are not anticipated to fail, so it will cause long waiting time from the manufacturer to supply parts. This is very undesirable compares to a scheduled downtime (maintenance), which is generally much shorter because most of the part replacements are anticipated. In addition, in the case of wind turbines, as mentioned in the section 2.2, it is quite expensive to have unanticipated component failures.

2.3.2 Scheduled Maintenance

This maintenance methodology is to perform maintenance and service at a fixed interval, so it will replace components before they fail. As time progress, some companies start to know reactive maintenance is not beneficial in terms of optimizing the cost of systems, so scheduled maintenance is developed. The advantage of this methodology is that it drastically reduces the unwanted cost (see section 2.2.1) due to parts failing unpredictably. Maintenance can be scheduled ahead of time and has a very predictable cost over the life time of the machinery. However, this method also has its disadvantages; scheduled maintenance uses too many man hours to maintain and replace components. During each service schedule, it often requires complete shutdown of the machinery. In wind turbines, it can take 12 hours to three days for a group of six to eight people to finish a typical maintenance process[Blankinship, 2008]. This will lead to days of system downtime, in which the company loses money. In addition, in order to avoid machinery breaking, sometimes components of the machinery are replaced well before the end of the components' life. This kind of early component replacements will also cause an increase in maintenance cost. So scheduled maintenance requires a lot more physical labor and has reduced service life of the designed parts.

2.3.3 Condition Based Maintenance

As the scheduled maintenance has areas to optimize, companies developed another maintenance strategy to optimize the cost of wind energy called condition based maintenance. This maintenance methodology emphasizes continuous monitoring of the health of machinery, and only do maintenance and service if a particular system parameter monitored reached the failure criteria. This active monitoring, if it functions as intended, drastically reduces the unwanted cost (see section 2.2.1) of maintenance. In addition, it also reduces physical inspection labor hours and minimizes system downtime. Minimum system downtime means maximum operation time in a given time interval, which is an important aspect for a power generation machine such as wind turbine. This is the preferred way of maintenance in this project.

2.3.4 Condition Based Maintenance with degradation model

As condition based maintenance has evolved in the recent years, the process has changed from passive to active. Companies have started to run mathematical degradation models to predict the time of failure. One of the most common way is to use Markov degradation model by Andrey Markov. Markov degradation model requires the system to have Markov property, in which the future states of the system only depends on the current state and is independent of the previous state². This project originally really looks into using this degradation modeling for condition based maintenance. However, the requirement of accurate quantifiable data for Markov degradation model is not available at the moment, so the project must use an earlier model of condition based maintenance, which reacts to failure criteria rather than predicting the time of failure.

²A good example can be seen in an event of dice rolling. The next rolling result does not have any association with previous rolling, so the whole event has Markov property.

2.4 Cal Poly Wind Turbine Project

The Cal Poly Mechanical Engineering Department is currently involved in a project to make a wind turbine. So far, the composite blades have been designed and built by Brian Edwards, and other components like tower and nacelle are being built now and almost ready to be put up as of this writing. The blades so far weigh about 10 pounds each according to Professor Patrick Lemieux from the Mechanical Engineering Department. The natural frequency still needs to be measured at this time, but it is roughly in the tens of hertz. The blades are measured 6 feet in length and they consist of steel hub, fiberglass (E-glass) and carbon fiber (AS4) in epoxy resin. These informations will serve as valuable resources in the rest of the project.

2.5 Composites

The word 'composites' is a name for a heterogeneous combination of materials. It can be a combination of many items. The main reason for combining materials is to take advantage of the desired properties from different materials to create a new combination that exhibits properties which cannot be found in one material. In the application of wind turbines, where strength in a particular direction and weight of the blade matter significantly, composites play a big role. As mentioned earlier, the blades consist of steel hub, fiberglass (E-glass), AS4 carbon fiber and epoxy. In order to improve the impact loading, E-glass was also added. It maintains the strength for relatively high fatigue cycle. The combination was chosen because of their great fatigue property. To demonstrate, a fatigue curve of AS4 Carbon fiber is shown in figure 2.5. For more information, please refer to the thesis by Brian Edwards [Edwards, 2009].

Carbon fiber retains 90% of its maximum strength when cycled for about 10^6 times; this makes carbon fiber really good for long cycle fatigue in the case of wind

Figure 2.5: $S - N$ Curve for $[0_2, 90_2]_S$ AS4/3501-6 carbon/epoxy laminate in comparison with experimental data from Lee, Daniel and Yaniv(1989), - -: 95% confidence level.[Harris, 2003]

turbine blades.

2.5.1 Common Composite Blade Failure Modes

Composites frequently fail in fatigue. Fatigue failure usually originates from microscopic cracks. As the stress from force exceeds the local yield stress, those cracks grow longer. This situation occurs even with a small force because stress is amplified at the tip of the crack, especially in brittle material, which will lead to crack propagation. One major difference between composite and regular brittle materials is that the initiation of cracks in a fiber does not directly cause a stress concentration onto another fiber nearby, because the high stress will reach the neighboring resin before arriving at another fiber. Since the resin is not as brittle as the fiber, it can yield enough to compensate and reduce the stress concentration.

There are a few common ways composite blades fail in fatigue. First, fiber

breaking due to high tension load. Second, delamination due to high compression load. Third, debonding of top and bottom skin of composite blades due to adhesion failure. All of these failures should have a direct impact on the natural frequency of the blade. In addition, another common failure is cracks in gel coat (sometimes due to lightning strikes or extreme weather), which will expose the composites to moisture and water. The exposure of moisture and water to the composite will significantly reduce the life of the wind turbine blades.

2.5.2 Frequency of Oscillation and Residual Life of Blade

Relating residual life to change in natural frequency is one of the subjects of interest in this project. However, researches that relate these two parameters could not be found. Future lab testing will be needed to determine this information.

2.6 Microcontroller

A microcontroller is a miniature computer with many peripherals in a chip. It can get data, process data, and control actuators. In this application of wind turbine, the main role of the microcontroller is to take measurements of the acceleration data via an accelerometer at a fixed time interval. After that, a Fast Fourier Transform (see section 2.9 for details) will be performed to convert the data into the frequency domain and then the microcontroller will calculate information needed to tell the user the health of a wind turbine blade.

2.6.1 Atmel Microcontroller

The microcontrollers (MCU) that are being used in this project are ATmega128 and ATmega324P. They are 8 bit microcontrollers from Atmel. Current modern computers use 64-bit processors to process, which means that at a given instruction

cycle, it can process a value up to 2^{64} , a gigantic number. But in the 8 bit system, the processor can only process a value up to 2^8 , which is one of 256 distinct values, at a given instruction cycle. So in order to increment a value of 260 by 1 to 261, the 8 bit processor has to take more than one instruction cycle. However, the 64-bit processor can easily process it in an instruction cycle. Due to the RISC³ architecture of CPU, a clock cycle is usually the same as the instruction cycle in the current 8-bit microcontroller. In the rest of the report, clock cycles will be used in place of instruction cycles because they virtually have the same value in the microcontrollers used in this project. The duration of the clock cycle is different in each system, and is defined by the frequency of the main oscillator. The frequency of the oscillators (called crystals) in this project are 32.768kHz⁴, 4MHz, 10MHz and 16MHz, meaning that they can process 32768, 4 millions, 10 millions and 16 millions instructions per second respectively.

The ATmega324P contains 32Kbytes (Kb) of programing memory and 2Kbytes (Kb) of RAM. 2Kb of RAM is very limited if it is used to process a large data set like in this project. So programing against the limited memory is a challenge here.

There are many components on the MCU. A few of them that are being used will be briefly explained here.

- Analog to Digital Converter (ADC)
- Timers
- Serial Peripheral Interface (SPI)

The first item that the microcontrollers have is an Analog to Digital Converter, which reads an analog signal (continuous value) and converts it to a discrete number

³RISC stands for Reduced Instruction Set Computing, is a CPU design strategy which increases the performance by using simplified instructions that can be executed in a short time (usually one clock cycle) rather than a complex instruction that uses many CPU clock cycles

⁴This is for an accurate sleep interval in power save mode

that the MCU can process. The timers keep track of the timing to make sure that the analog signal value is read at a fixed constant interval. The Serial Peripheral Interface communicates to a radio module on the microcontroller board to send communication information to the other radio module attached to a desktop or laptop computer via radio signal.

The microcontrollers also have a feature that is absolutely critical to the success of this project; it is the ability to sleep. During sleep, they use a very minimal amount of current. They could achieve at low as 100nA^5 ($1/100000000$ Amp) of current [Atmel, 2010]. For a good comparison, an LED generally considered as a low power device, and it draws about 10mA , which is about $1/100$ Amp. This ability to sleep is critical in a long term running environment. This is so because as a sensing component, the microcontroller needs to be functioning at all time, and drawing too much power will drain the battery a lot faster. With the ability to sleep, there is no need to stop a wind turbine frequently for battery replacement, which is a lot more cost efficient.

2.7 Accelerometer

An accelerometer is a component that measures acceleration. It converts a physical acceleration or gravitational acceleration to a voltage reading. From reading the voltage coming from an accelerometer, one can approximate the actual acceleration of the component. In this case, we can approximate the acceleration of the wind turbine blade to find the frequency of oscillation.

The accelerometer that is used in this project is ADXL330[AnalogDevices, 2006]. It is a three axis accelerometer that can measure up to $3g$ of acceleration. The accelerometer can detect up to 1600Hz of frequency and only uses up to 375uA

⁵In power down mode. However, this project uses the power save mode which draws 600nA of current

of current. However, the accelerometer does not have a power reduction mode, so it constantly draws 375nA of current. With a limited battery capacity, 375uA of current will be a challenge in the project that runs for years.

2.8 Computer Programming

Two programs are used in this project: one is for the sensor, one is for the receiver. The CBM sensor program in this project specifically gives instructions to the microcontroller to measure data, process the data, and communicate with another microcontroller wirelessly via the radio signal. The CBM receiver program is used to receive data from a remote microcontroller via the radio interface and then output the data to a personal computer. All the actual programming functions will be discussed in full in Appendix ??.

2.9 Fourier Transform

Fourier Transform converts data from time domain to frequency domain. There are two major types of Fourier Transform. Continuous Time Fourier Transform and Discrete Time Fourier Transform. Continuous Time Fourier Transform deals with functions that is continuous. Whereas Discrete Time Fourier Transform deals with sampling data sets. Discrete Time Fourier Transform will be used in this project due to the nature of the discrete data sets. So for the simplicity of the discussion, the rest of the report will refer Discrete Time Fourier Transform as just Fourier Transform.

Fourier Transform is like tallying a set of data and categorize them into different bins of frequency. If the input data that will undergo Fourier Transform is not varying, the output bin of zero frequency will be huge. If the input data is a pure sine wave, the output bin of the sine wave's frequency will be huge. In addition, if

the input data is a combination of different sine waves of different frequency, then the output will show a few large outputs at the frequency of the input sine waves. The importance of this Fourier Transform is that it analyzes a set of input data to see whether the input data has a set of dominant frequencies. By finding those dominant frequencies, the user can easily associate physical characteristics with them. In the case of this project, the frequency of oscillation (FO) is of interest.

2.9.1 Methods of Fourier Transform

Fourier Transform is a process that is quite tedious to perform. As the amount of data points becomes large, it will use a lot more processing power of the computer. Because of that, most people use a faster version called Fast Fourier Transform to perform the Fourier Transform nowadays. Both of the methods will be described in the following sections.

Classic Fourier Transform

Classic Fourier Transform (Discrete Fourier Transform) was the only method to do Fourier Transform before the development of Fast Fourier Transform. It takes a lot of time to calculate because it takes N^2 amount of arithmetical operations to complete the computation, where N is the amount of data points. As more precise resolution is required, more data points are used. As a result, it takes a much faster computer to do the classic Fourier Transform. With limited resources, if microcontrollers were used to compute the classic Fourier Transform, the resolution will really suffer.

Fast Fourier Transform

Fast Fourier Transform is a method to perform Fourier Transform. It drastically saves processing cycles of the computer. With Fast Fourier Transform, it can even

bring the process intensive Fourier Transform to an environment that does not have much processing power. A great example is microcontrollers. Because it saves processing power, the microcontrollers can use a much larger data set to get a more refined solution. Due to the limited processing power of the microcontrollers, only Fast Fourier Transform can fit the need of this project.

Besides saving processing power and processing time, Fast Fourier Transform also has an advantage in accuracy over the classic method as well. It is so because when a computer interacts with the surrounding environment to take data, it converts from a continuous analog number to a discrete number that computer uses. There are always small accuracy issues from converting an analog value to a discrete value. This is known as the round off error. This is usually a very small problem in a calculation because the process can easily use a higher resolution number to calculate values to minimize the uncertainty of the final result. But the problem with the classic Fourier Transform is that it uses values to add, subtract, multiply many times repeatedly before the final number is crunched, so even small accuracy issues can accumulate and affect the accuracy of the final result. However, the Fast Fourier Transform reduces that problem because it minimizes the amount of calculations, which in term reduces the tolerance stacking of the result.

2.9.2 Integer Fast Fourier Transform

As mentioned in earlier section, the processing power of the microcontroller is not great. Besides not having good processing power, the small and low power microcontrollers also do not have a floating point calculator⁶, which makes decimal calculations a lot slower than on a regular desktop computer. In order to reduce the amount of time to process, and still retain the desire resolution, a routine that straightly uses integer for the whole Fast Fourier Transform is then created. This

⁶A device that is dedicated to do decimal calculations. If it does not exist, it will take a long time to complete a calculation

routine treats values to have a higher resolution in an integer representation. In order to do this, the processor has to multiply a data's value by a given number (usually in the powers of two because microcontrollers are very efficient in handling multiplication or division in powers of two via bit shift⁷) and divides the number by the amount that it multiplied earlier to reduce the value back to a number of interest. This way, the computer can keep the proper amount of significant figures throughout the calculation.

Methods to improve the speed of Integer Fast Fourier Transform

The microcontrollers that are being used in this project are 8-bit, so the maximum number that they can process within one clock cycle is 255. This means that they will take longer to calculate things with resolution higher than $1/256$. However, the resolution is not needed a lot of times, so the program will use the smallest size of variables when applicable to reduce processing speed. In addition to reducing the size of variables, procedures that limit the calculation of variables to a certain minimal amount of bit length⁸ (the larger the bit length, the longer it takes to process) is also used.

In an attempt to use trigonometrical operations like sine, a process that speeds up the calculation called Look Up Table is used. This is a technique that converts all the values of sine into a pre-calculated form. A process to look up a value is a lot faster than to calculate a value every time. So by using this technique, it will drastically reduce the amount of time it takes to get a value of a trigonometric operation.

⁷This is a binary number operation in the microcontroller, which is used by computers for calculation, the two operations are: 1) dividing by power of two, it is accomplished by adding zeros to the bit that is the left-most of the number and discard the right-most bits. 2) multiplying by power of two, it is accomplished by adding zeros to the right-most of the number and discard the left-most bits of the number

⁸8-bit, 16-bit, 32-bit

2.9.3 Fast Fourier Transform Limitations

There are limitations associated with Fast Fourier Transform.

First, it can only take an amount of input data that is in the powers of two, for example: 64, 128, 256, 512, etc. This is limiting because the users can't input a number of data in between. However, there is a trick to resolve this issue (see section 2.9.4).

The second limitation⁹ is that it has to sample at least double rate of the frequency of interest (Nyquist frequency). For example, if the frequency of 20Hz is of interest, the frequency of sampling has to be at least 40Hz. Or else, the value may not be accurate. However, if the processing power is available, it is always a preference to sample the data 10x faster than the frequency of interest. This is due to a better capture of the wave form for accuracy.

The third limitation¹⁰ is that the resolution of the Fourier Transform is strictly determined by two factors, nothing more. The first one is how many data points used. The second one is how close the measured frequency is to the middle of the frequency spectrum¹¹. Since taking extra data points sometimes is impossible due to the processing limitation. To minimize the percent error of the result, it is quite important to have a good guess of the measuring frequency to start with.

2.9.4 Fourier Transform Trick

There is a trick which will make Fourier Transform easier to work with. It is called Zero Padding.

⁹also applies to regular Fourier Transform

¹⁰also applies to regular Fourier Transform

¹¹It is to the middle of the spectrum rather than the end is because the middle frequency is the maximum detection frequency due to the Nyquist frequency requirement

Zero Padding

As mentioned earlier, Fast Fourier Transform only works with data sets that are in the powers of two. This may cause some trouble in getting higher resolution as increasing the amount of data points may take too long to record. If a person only needs 300 data points and it takes 8 seconds to record them, but since the next power of 2 is 512, so it is forced to take 512 data points to get that resolution. This will use 14 seconds instead of 8, and sometimes it may run into some timing problems.

This problem can be resolved via a process called zero padding. Zero padding is a method to get higher resolution of the result without spending as much time taking all the data points. This is done by tricking the Fourier Transform that it has fully obtained all the data. But in fact, it has not. The method is mainly adding zeros to fill the whole data set to the powers of two. Adding zeros does not alter the result of the Fourier Transform, it only gets a more resolution detail instead¹². This is definitely a very useful tool to use, but the downside is that a larger set of data takes a lot longer to process and it does not really give any more information to the FFT, but only increases the resolution of the existing information.

2.10 Energy Source

All electronics consume electricity. Although the microcontrollers are very power efficient and use only a tiny amount of power, especially with the ability to sleep (low power standby) and save power¹³, they still consume quite a bit of electricity overtime. There needs to be a system that is able to supply the microcontrollers power in order to monitor the blades's health. Since there is not an electricity supply from the wind turbine blade directly for sensors, three energy sources are being

¹²Provided that it has taken enough data points for the Fourier Transform to know the frequency of a particular wave

¹³They consume down to a few micro Amps during deep sleep (in between each sample cycle)

considered in this project to supply the power for the microcontrollers. The three sources are: non-rechargeable battery, solar power and vibration power. Although each of them has it's own advantages and disadvantages, battery is selected at the end. A detail discussion is in the following sections.

2.10.1 Non-rechargeable Battery

Non-rechargeable Battery (it will be referred to as only 'battery' for the rest of the discussion) has been the choice for powering microcontrollers for many systems. There are a few advantages and disadvantages of using battery as the power source. One advantage is that it can output rather consistent voltage. They can generally supply the system for a good amount of time. However, there are a few draw back of using battery. First, they run out of charge, either via current draw of the system, or by self discharging¹⁴. Second, batteries don't have constant voltage towards the end of their life, so the voltage will slowly change overtime as it is draining. As a result, this will have a significant effect on designing parameters. Third, they are relatively heavy, so minimizing the amount of batteries used is a concern. Finally, they also need to be maintained for an optimal power draw from them. They can't be exposed to extreme temperature. Battery is not perfect, but it is still a very good source of portable energy. There is a list of batteries' comparison in Appendix A.

2.10.2 Solar Power

Solar power is a great renewable energy. It takes only solar cells to convert photons into electricity. The advantage of solar power is that it is an active source that generates electricity on the go as photons are available. So the user does not have to

¹⁴Self discharging is a term for batteries. A battery will lose charge overtime by itself. This self discharging rate depends on battery chemistry (for example: lithium, lithium-ion, alkaline, nickel metal hydride, etc...) and temperature of storage. See Appendix A for details

worry about the source of energy runs out overtime. Second, The solar power is free as long as the equipment is there to gather solar power. Although in a project like this, the energy consumption for the microcontroller is very minimal that this is not really a significant advantage. Third, the process to obtain electricity is completely clean and does not involve any processes that harm the surrounding environment.

However, there are a few disadvantages. One disadvantage is that the amount of energy gathered is not consistent. It fluctuates a lot from one second to another due to the incident light onto the solar cell is not always consistent, so it is very hard to use this energy without first being processed. Second, the amount of electricity being generated at night is close to zero. So in order for a project to use solar power, it will either have ways to supply energy at night, or storage like rechargeable battery is implemented to use energy stored in the daytime. Third, solar cells generate only a small amount of energy and therefore in order to generate an amount of electricity of interest, generally they require a rather large footprint of the solar cells array. At some circumstances, this is not viable. Fourth, solar cells require cleaning to stay at an optimal condition. Due to rain and moisture from the air, the surrounding dust can easily adhere to solar cells' surface and reduce their output. In addition, bird droppings landed onto the solar cells also reduce their output. Both of these situations will drastically reduce the amount of output from the solar cells, so to stay in an optimal condition, cleaning solar cells is required. Fifth, in order to implement solar cells, there are more electronics involved that will add to the failure rate of the entire system.

2.10.3 Vibration

Vibration energy is usually used in an environment that vibration exists at a constant frequency. For the ones that are being researched, they contain an element that resonates at the designed frequency that matches the vibrating source to generate energy. The frequency generally is either 50Hz, or 60Hz because they are the

typical frequencies of the AC motors. Since AC motor uses AC electricity, and the frequency of the AC electricity is usually either 50Hz or 60Hz, so that's why they are usually designed at those particular frequencies. The vibration of the motor is generally enough to give the sensing equipment energy. However, it is really hard to find a vibration energy gathering device that fits due to the relatively high natural frequency of the wind turbine blades. Even if ones that fit exist, they will still cost a lot of money because not many manufacturers custom made them.

Although it is hard to find one at a reasonable cost, there are some advantages of using vibration power. First, it is like solar power that it is an active energy source that generates electricity on the go, so it does not have to worry about losing electrical charge due to long time of storage. Second, it is a consistent source of energy, and it generates power as the machinery runs. As long as the vibrating source exists, there will be energy generated. Third, it is generally free because the vibration is usually a product that is not wanted in the first place. Fourth, it is a clean energy source, and it does not create anything that harms the environment during the process of generating energy. Fifth, when designed properly, it can also act as a damper that reduces vibration, which prolongs the machine's lifespan.

However, vibration power has a few disadvantages as well. First, it requires a consistent vibration of the designed frequency, any frequencies that vary from it will have a drastic reduction of energy generated. Second, when the vibration source does not have consistent vibration, it will not be generating a consistent amount of energy. Third, vibration power generally does not generate much energy just like solar power, so there are limited applications where vibration power can be used as the power source. Fourth, since it has mechanical and electronic components in vibration power, it is relatively easier to fail compare to other pure electronic component systems, so a well designed system is needed for a prolong usage of the system via vibration power.

Chapter 3

Theory and Process Layout

Condition based maintenance's goal is to minimize the cost associated with basic maintenance. It is accomplished by continuously monitoring parameters of importance, yet minimizing the amount of interruptions to the system that is monitored. After getting those parameters, a decision needs to be made to leave it as is, to overhaul it, or to replace the machinery. This monitoring and decision making take many man-hours to accomplish. But thanks to technological advancement, this process is now usually being done by electronics such as microcontrollers. However, the important parameters to monitor must be decided very carefully. Because when a human inspects a machine, the person sees the whole machine and sees how things go together; whereas when an electronic inspects the machine, it only sees the parameters that are programmed into the electronics, nothing else. When an unexpected failure occurs, the electronics may not be able to capture the failure at all if the parameter is not being monitored. As a result, the electronics will continue to let the machine run until a catastrophic result happens at the end. So, a careful consideration to decide what parameters to monitor is important.

3.1 Failure Detection

Failure of wind turbine blades is defined by defects (such as cracks, delamination, or things that affect the rigidity of the wind turbine blade) that affect the strength of the blades to below a certain threshold. There are many parameters to detect for determining failures: amplitude, acceleration, velocity, angular acceleration, moment

of inertia, frequency of oscillation¹ , etc. Frequency of oscillation was chosen as the source to determine failure because it is a source that displays a more consistent property over time.

3.1.1 Natural Frequency

Natural frequency is a frequency in which an object ‘naturally’ vibrates in. This is directly related to the material and it’s shape. When an object is subjected to a disturbance, such as an impact or changes in forces applied onto the object that moves the object away from the equilibrium point, the object will try to return to the equilibrium point along an axis. During the process of returning to the equilibrium point, it may oscillate for awhile before getting back to the equilibrium point. In the case that damping does not exist, that oscillation is generally referred to as being the natural frequency of the object among a certain axis.

3.1.2 Frequency of Oscillation

Frequency of oscillation is a parameter directly associated with natural frequency with a damping factor that differs from materials and shapes. If the damping is found, the natural frequency can be obtained from equation 3.1 :

$$w_o = \frac{w_d}{\sqrt{1 - \zeta}} \quad (3.1)$$

where ζ is called the log decrement, which is directly related to δ , see equation 3.2.

$$\zeta = \frac{1}{\sqrt{1 + (\frac{1}{\delta})^2}} \quad (3.2)$$

the δ can be found using the following eqn. 3.3 with peak magnitude ratio of subsequence cycle of vibration

¹Detecting the natural frequency is impossible because it requires the system to have no damping

$$\delta = \frac{1}{n} \ln \frac{x_0}{x_n} \quad (3.3)$$

where n is the number of periods away from the current peak.

The reason why frequency of oscillation is important in this project is that frequency of oscillation is a property that is easier to measure than the natural frequency. It is also easy to get natural frequency from frequency of oscillation by experimentally determining the log decrement. Then with eq. 3.1, 3.2, and 3.3, natural frequency can be calculated. In a vibrating system, the damping is usually small, so the frequency of oscillation does not vary from the natural frequency much, and sometimes since they are so close to each other, many people use the term interchangeably.

3.1.3 Failure Criteria

Composite members generally behave very well for fatigue life (see figure 2.5). They can generally operate near ultimate strength at high fatigue cycle, which means that the blade can maintain at high strength even when it is damaged. However, since there is not any data that compares the residual life to change in natural frequency, so a failure criteria of 8% from the reference frequency of oscillation is used. This percentage has no theoretical basis, and future analysis that reexamine this failure percentage should be done.

3.2 Finding Frequency of Oscillation

It is critical to find the frequency of oscillation correctly since this is the only measure to determine failure in this project. The way that is used in finding frequency of oscillation is to use a set of acceleration data within a time interval and see how frequently the peaks occur by IFFT, then a filtering process is used to narrow down

the selection down to one, which is the frequency of oscillation. At a given axis, the frequency of oscillation does not change as long as the material and its shape property remain the same. So finding a frequency that occurs the most during the whole vibration cycle should yield the frequency of oscillation of the blade. With all these in mind, the whole process starts with finding the acceleration.

3.2.1 Acceleration to Fourier Transform

The acceleration data is taken via an accelerometer. It converts acceleration that it detected to a discrete voltage reading for a microcontroller. Since the accelerometer only creates a continuous set of analog data, while the microcontroller only processes discrete data, a process call analog to digital conversion is required for each data point. The conversion can be done via a peripheral of the microcontroller called the Analog to Digital Converter. The ADC has a resolution of $1/256$ ². After the conversion of the whole set of data is completed, the microcontroller will use the converted data for processing.

The most important thing at this point is to get data at a precise time interval because accurate precise time interval data will generate precise IFFT results. The microcontroller can respond within tens of microseconds in getting the data via a process called interrupt³. This use of interrupt will ensure the precision of the timing. With a good set of data, now the microcontroller can execute a sequence call Integer Fast Fourier Transform to find the frequency of oscillation.

²The actual max resolution is $1/1024$, but due to not having enough RAM for larger data set, the resolution of the ADC is sized down for better IFFT resolution

³A feature of the microcontroller that when the interrupt is triggered, the microcontroller will immediately stop processing the current operation and process the code inside the interrupt; this is useful for time precision data gathering

3.2.2 Integer Fast Fourier Transform

The Integer Fast Fourier Transform (IFFT) developed here is a processing sequence designed specifically for the 8-bit microcontrollers (In this case, the ATmega324P). It uses integers to process and convert the whole data set from the time domain to the frequency domain. It is a process that is catered to the processing power of the microcontroller that is in the project, yet balancing speed and resolution requirements. With this in mind, the acceleration data now can easily be processed to find the frequency of oscillation.

3.3 Accuracy of IFFT

The accuracy of the IFFT is heavily influenced by the resolution of the frequency spectrum. The resolution of the IFFT is defined by the eq.3.4.

$$Resolution = \frac{SamplingFrequency}{NumberOfDataPoints} \quad (3.4)$$

$$PercentResolution = \frac{Resolution}{MeasuringFrequency} \quad (3.5)$$

The current system (Swoop board with ATmega324P) can at max sample 128 data for a total of 256 data points⁴ used in the FFT at every sampling cycle. When sampling at 300 Hz to measure a frequency up to 150 Hz, the resolution is 1.17Hz. At the same sampling rate, when measuring frequency of 100Hz will imply a percent resolution of 1.2% between frequency bins. With some margin for error, this is only good enough for a wind turbine blade that fails at 5% or more of frequency changes. So if taking samples at the same rate to detect 30Hz of frequency, this will mean a resolution of 3.5% per resolution bin, which is not desirable to measure failure that

⁴The process of FFT requires complex numbers as input, so only 128 real number can be input into the system, while the other 128 data points are all 0 in the imaginary axis

is around 8%. This shows the importance of detection frequency being closer to half of the sampling rate will yield a more desirable frequency resolution.

3.3.1 Improving Accuracy

There are a few ways to improve accuracy. Besides getting close to half of the sampling frequency that is mentioned in the previous section, here is a list of them:

- Using more data points
- Determining the in-between data.
- Filtering unwanted trials or data that is the outliers and find the average.
- Using higher resolution numbers in all cases.

Using more data points increases the resolution significantly. However, there is a hardware limit. As more and more data points are used, the processing time also increases drastically. So a good balance is needed. In this case of using ATmega324P, the maximum amount of data points is significantly limited by the RAM, because of that, a process to reduce RAM usage is implemented. The process involves using the minimum size of data type to declare variables. By doing this, it not only frees up some RAM for extra data points to be recorded, but it can also increase the processing speed. In addition, by an increase of RAM size from 2K to 4K, the number of data points can go up to as high as 1024 data points (512 sampled data), this is used in the ET128 board.

The second method is to determine the in-between data. Sometimes an actual peak in the frequency domain could be in between two adjacent frequency bins. In order to have a better estimation of the actual natural frequency, a method of using adjacent bin's magnitude is used to evaluate the actual peak frequency. The method assumes normal distribution. It compares the two neighboring frequencies'

magnitude and shifts up to half of the resolution depends on which side has higher magnitude.

Averaging results will usually improve some accuracy. However, sometimes there are outlier data that will affect the average of the data. Because of that, a process is used to filter out unwanted data. Since a frequency of oscillation is a property of material and it's shape, the frequency of oscillation won't change from trial to trial. Since it always exists, it should be the prominent peak in each of the IFFT computations. So all it takes is to find the frequency that occurs the most, and filters out all other frequencies that do not occur frequently. This will generally further improve the accuracy.

Increasing the resolution in all calculation procedures will give higher precision numbers. If there is enough MCU processing power, this is definitely the process to be done. But in an environment that competes for resources, the gain in resolution outweighs the precision at this point, so this is why 10-bit ADC is not used.

3.4 Communication

In order to do CBM, one of the requirements is that the status of the system needs to be periodically monitored and checked for problems. In order to do that, it requires means of reporting the frequency of oscillation from the microcontroller. Since a direct line of communication is not feasible in this scenario because it is on a rotating part, the simplest solution is to use wireless communication.

3.4.1 24L01

There are different kinds of wireless communication. Due to equipment constraint, the project uses 24L01 from Nordic Semiconductor. This is a 2.4GHz module that is capable of sending data remotely at about 20ft. It is a low power module, so it suits the project quite well. In order to use the 24L01 for communication, it requires

two modules: one is for sending, and one is for receiving. The goal is to be able to get all the data wirelessly from the wind turbine and send them to another location nearby that uses a much more powerful long distance transmission. The range is required because the data needs to be received back in California Polytechnic State University, San Luis Obispo (Cal Poly) for condition based maintenance monitoring. One of the most important parts of the project is to successfully receive the signal from the sensor because it means absolutely nothing to the condition based maintenance if no data is received. A successful communication requires accurate processing and sending of the data.

3.5 Power Supply

Another challenge of this project is to build a self-contained module because there isn't any physical electrical wiring to supply power. As mentioned previously, an accurate processing and data sending are essential to the success of the system. Both of these requirements are directly associated with power supply. And it will be discussed in the next two sections in how to supply power, and the power usage.

3.5.1 Supplying the Power

As mentioned, a self-contained power source is required. Besides the fact that it needs to be self-contained, the power source also needs to satisfy three requirements:

- Be reliable
- Have good voltage stability
- Provide enough power at all time

Although research was done for solar power and vibration power, the project at the moment uses pure battery to power the device. This is due to complications

involved with the charging system. Overcharging protection, maintaining power availability and maintenance of the power source, all these will increase weight, yet lower the reliability of the system. The current design is based on simplicity of obtaining power by a battery pack and replacing batteries when necessary. This has two advantages. First, higher reliability due to simplicity of the system. Second, without the charging system via the solar or vibration power, primary batteries can be used⁵. The advantage about primary batteries is that they usually have very low self-discharge rate and as well as higher battery capacity for use. A good example is the Ultimate Lithium Battery L91. The mass is only 14.5 grams (half of what nickel metal hydride battery weighs), yet it is rated for about 3000mah and has a rated life of 15 years. At constant 40 degrees Celsius, even after 15 years, it will still retain 60%⁶ of the charge[Energizer, 2010b]. Since this system is designed to be long lasting, so low battery self-discharge rate is very important. This is why lithium battery is selected for this project.

3.5.2 Managing the Power Usage

The second issue is power usage. There are a few processes in the sensing system that consume the majority of the power, for example: sampling data, processing data and sending data via radio signal. Although at max the microcontroller system uses about 40mA, even if a 9V battery is used, this is only consuming 360mW. This is 1000 times less than what normal computer draws while running. It may seem like very minimal amount of power usage, but in the embedded system, this is a lot of power that is consuming. Consider a typical 9V alkaline battery that is rated at about 300mAh. By using the eq. 3.6:

⁵Batteries that are not rechargeable, like: alkaline, lithium (not lithium-ion)

⁶It could even remain 90% charge if the temperature is kept at 20 degrees, see Appendix A for more detail on battery comparison

$$TimeDuration = \frac{Capacity}{PowerUsage} \quad (3.6)$$

The time it takes to completely drain the battery is about 7.5 hours with this equation. This is reasonable for maybe a laptop, but for a continuous monitor system, this is not acceptable. It means a person will have to stop the wind turbine every 7.5 hours to replace the battery, or to charge it. In order to compensate for this problem and increase the time duration of running. We have two methods, they are:

1. Increase capacity
2. Reduce power consumption

The first method is to increase the capacity of the battery, which is mainly increasing the size and weight of the battery, or adding more batteries to supply the power. The second method is to reduce the power consumption of the system. Both of these methods are described in the following two sections.

Increasing the Capacity

Increase the capacity of the battery is feasible only up to a certain point. The mass of the sensing system is still quite manageable when the battery is small and light (like a 9V battery), but when multiple batteries are used, the mass of the system will increase drastically. However, the amount of time the batteries are operating will still not be close to what is wanted from the sensing system. Assuming the battery is designed for at least 5 years of usage, at the rate of power usage from the system (from eq. 3.6) even if the self-discharged rate of the battery is ignored, this will still require 5000 9V alkaline batteries (45gram of mass each). If the mass is calculated, this will be equivalent to about 225kg of mass, 50 times the mass of the blade. So this clearly shows that it is not possible to solve the problem of

duration of operation by purely increasing the capacity. Instead, the system needs a big reduction in power consumption.

Reducing power consumption

There are a few ways of reducing power consumption. In order to minimize the power consumption, it is very important to identify the components that use the most power and reduce the usage or shut them off to minimize the overall power consumption. As mentioned earlier, the major power consumer in the system is sampling data, processing data and sending data via radio signal. Shutting their power off when it is not being used will save a lot of power consumption.

The microcontroller has six modes of power reduction sleep mode. Two sleep modes work perfectly in this project: they are power save mode and idle mode. The process here is to see the minimal amount of power the whole system uses and starts to add components to make the system functions as designed. Since different settings have power consumption differences, all systems that are not specified in terms of setting will automatically be assumed to run at the following specification:

- Crystal frequency - 4MHz
- Voltage - 3.6 Volt (without regulator and directly powering the system)
- Sampling frequency- 200Hz
- Data points - 256 per sample set used by the FFT
- Time between each set of sampling - 30mins

Power Save Mode- minimum baseline usage The baseline usage involves using only components that are needed to be on for the entire sensing duration. The bare minimum components here are the core of the microcontroller to retain the data and the Real Time Clock (RTC) crystal (32.768kHz) to keep track of

timing. The RTC also enables the microcontroller to sample at a fixed interval (in this case 30 minutes per sample set). The ATmega324P is capable of only using 0.6uA (0.0000006A of current) when it is not processing anything in that particular power reduction mode.

Normally, the voltage to keep the microcontroller running steadily can be as low as 1.8V⁷, so the minimum power usage of the microcontroller for just keeping track of time is 1.08uW. So based on 100% efficiency conversion and without self-discharging, a 300mAh 9V battery can theoretically last for 285 years. This is the bare minimum power usage. This number is actually very unrealistic as self-discharge rate affects the life of a battery a lot. But by knowing the current drain, it is easy to approximate the life of the microcontroller powered by battery.

Limitation and waking up from power save mode Although the microcontroller is capable of using 0.6uA as the power save mode current, since it is using 3.6V as the power source instead of 1.8V, the current draw is about 0.7uA.

In order to keep track of time for a fixed time set data sampling, a continuously running clock is required, and Real Time Clock is used in this place. It does not stop even when the microcontroller is going into the power save mode. The way it keeps track of time is to run a counter inside the microcontroller until it overflows⁸. When the number overflows, it will wake up the microcontroller. Then the microcontroller will increase a counter inside the program. As the program counter reaches a certain user defined number, it will start the process of taking a set of acceleration data for the Fourier Transform. Since the microcontroller during normal operation uses so

⁷This voltage is a bare minimum to run all the process smoothly, any fluctuation that drops below this voltage can cause a risk of microcontroller error that can ruin data. This voltage can guarantee operation up to 4MHz. Any faster processing requires a higher voltage as specified by the Atmega324P datasheet

⁸Exceeded the maximum counting number of the microcontroller. In this case for this particular microcontroller, 255 is the max number

much power, so the best way to do this is to increase the max possible wait time between microcontroller wakes up. In this case, eight seconds is the longest that microcontroller can set before each wake up. Since that duration of time to wake up the microcontroller also uses power (at full processing power usage), this amount of power usage is needed to be factored into the calculation for the maximum allowable operation time.

Page 33 of the ATmega324p datasheet states that it takes about 16k clock cycle for the crystal to be used. With processing time (including incrementing the programing counter) needed before the microcontroller goes back to sleep, a good approximation would be about 20k clock cycles of the microcontroller processing time for each eight second period. The actual time used is about 5 milli-second (assuming a 4MHz crystal) for each wake up time (Assume the microcontroller is directly powered and is running at 4 volts, and 4 MHz), this means the microcontroller use 2.8mA of current⁹ for 5ms. For every hour, it will wake up 450 times. At the rate of 2.8mA for 5ms, it is equivalent to using about 1.8uA continuously on average.

Sampling Data - power usage Sampling data draws a lot of power as well, page 341 of the ATmega324P datasheet mentions that the microcontroller uses about 0.6mA when the system is at the other power reduction mode- idle mode. The microcontroller samples 256 data points. If the process is sampling at 200Hz, it will take 1.280 sec to complete sampling 256 data points. Assuming each sampling uses about 500 clock cycles between sleeps. For 256 times, the whole sampling time will take about 128000 clock cycles (32ms). The total time of sleep is the total time subtracts the sampling time, which is 1.248 sec. The total time of sampling is 32ms. So the average power being used is 0.5uA over an hour.

⁹p.339 of ATmega324P datasheet

Processing data- power usage In data processing, the time it takes for processing data to do FFT and other operation is about 200ms. And at the rate of 2.8mA, on average, it is using about 0.3uA of power on average.

Radio sending- power usage The radio operation uses about 12mA of current and it uses about 50ms per each sending. On average, it uses about 0.3uA of current from sending. However, during power down mode of the radio module, it uses 0.9uA of current. So on average, the radio operation uses about 1.2uA of power.

Accelerometer The accelerometer does not have any power down mode, so it consistently uses about 375uA of current.

Total power usage So the total current draw of the system is very minimal if the system only turns on the required components when it is needed. The total power can be calculated from summing all the previous section's current draw results:

- Power-Save current - 0.7uA
- Power-Save wake up current - 1.8uA
- Sampling data - 0.5uA
- Processing data - 0.3uA
- Radio - 1.2uA
- Accelerometer - 375uA
- Total - 380uA

If 100% efficiency is assumed for a 9V 300mAh alkaline battery to supply the power, this battery now can last for 82days rather than 7.5 hours. If assuming the accelerometer can be shut down and draws only 1uA of power on average, the total

power draw is down to 5.5uA. The total amount of time that the system can last for now is about 15.6 years, which is very close to the designed life of the wind turbine blade. If additional batteries are used, it is very possible that the system can last for the whole duration of the wind turbine blade designed life if the battery self discharged rate is low enough.

3.5.3 Using L91 battery

Although in previous section, 9V alkaline battery is assumed to be the power source, but there is a better battery for this application, called L91 from Energizer. It has two main advantages. First, the chemistry of lithium is better than alkaline in a long term usage because it has a much lower self discharge rate. That's why the shelf life of alkaline batteries is only about 7 years compared to 15 years from lithium batteries. Second, 9V batteries require an additional electronic component called voltage regulator, which tends to waste a lot of power. If L91 (AA size) batteries are used, they can directly power the microcontroller without an additional voltage regulator. With this in mind, as long as the self discharge rate of L91 is found, a more precise battery life estimation can be calculated.

Self Discharge Current of L91

An assumption of a temperature environment that the battery pack is operating in is described as follows: half of the time is spent at day (40 degrees Celsius), and half of the time is at night (20 degrees Celsius). The datasheet of L91 said it will retain 60% at 40 degrees Celsius and 90% at 20 degrees Celsius after 15 years at a linear relationship[Energizer, 2010b]. Assuming that is true for the whole battery's life time. This is equivalent to continuous drain of 9.1uA during the day, and 2.3uA during the night. So, on average, it is draining at a rate of 5.7uA.

Battery Life with 2AA L91 battery

Since the system needs to have at least 1.8V to operate, this will mean two 3000mah 1.5V L91 batteries are needed. With the assumption in section 3.5.2 (including replacement of the accelerometer), the equivalent power loss is 11.2uA on average. So the life of the battery is about 30.6 years. At 29 grams of weight, this is definitely a viable solution.

Chapter 4

Programming

This chapter describes the stages of the programming process in this project.

4.1 Fast Fourier Transform

The programming started with researching Fast Fourier Transform. With knowing that it is a very time consuming process, many decisions have to be made to cater to the microcontrollers used in the project. Energy consumption is a major factor with many decisions made because the system needs to operate long enough. For example, quadrupling the sampling and processing time can roughly translate to five years versus 20 years of operation time. Some decisions can even save energy in an order of hundreds times. The following sections will explore the process that the project went through to get a successful IFFT computation.

4.1.1 Preliminary Programming Decision

To start programming, a programming language needs to be selected. Although assembly implementation should be used for a more efficient computation process for the least amount of power consumption, C++ is used because it is easier to read and easier to adapt to the code for use in the future. In addition, it is also easier to upgrade in the future because of the modularity of C++.

There are a lot of implementations of FFT in the Internet, so choosing one to use was a hard task. Microsoft Visual Studio¹ was used to test run the different

¹An Integrated Development Environment for programming

FFT found on the Internet. Matlab² from MathWorks was used to plot the results to make sure the FFT worked and to generate sample data.

4.1.2 Getting the Fast Fourier Transform to work

Adaptation to other people's code is quite troublesome. After different test runs, the FFT code from drdobbs.com is adapted. It is a C++ implementation of Cooley-Tukey Fast Fourier Transformation[Myrnyy, 2007]. One of the hardest parts of using the code from Dr. Dobbs is that the code is not well commented. It is already hard enough to understand the code; it is a nightmare to convert data from floating numbers to integers. In addition, there is a function in the FFT that does not show the definition. This leads to some extra processes on researching about how FFT operates. At the end, the function is re-created.

In order to successfully test the FFT, two Matlab files are generated. One file is called `input_dat.m` that generates a time sequence data at a fixed interval and output to a text file. The other one is called `plot_fft_result.m`, which takes a file generated from the FFT and plots it in Matlab. These two files are repeatedly used to test and to make sure the FFT works.

4.1.3 Getting the Integer Fast Fourier Transform to work

After the FFT works, the next step is to get the FFT to work with pure integers. Besides changing all the decimal numbers into integers, there are two main problems that the programming encounters: resolution and processing power.

Trigonometric functions like the sine cannot be represented by integers, as its values only range from -1 to 1, and this does not have enough resolutions. In order to preserve the significant figures, the sine values are multiplied by 32768 and rounded to the closest integer. However, the problem with this is now after some processing,

²A program that processing mathematical operations

numbers may become so big that even 32-bit numbers are not sufficient. In order to work around this issue, numbers are multiplied and divided at times to ensure the number is small enough, yet still retain the proper significant figures.

Since the input and output of the sine are continuous numbers, calculation of sine values may take a long time with these microcontrollers. As a result, a sine table is created for a faster sine output for input numbers ranging from 0-360 degrees. This method significantly reduces the amount of processing time. Although this method also reduces the resolution of the initial input number, the value is still good enough for this project.

4.2 Embedded System Programing

The project overcame a big milestone when the Fast Fourier Transform can operate with pure integers. This enables a much faster Fast Fourier Transform with the microcontroller because the microcontroller now does not have to deal with floating point numbers. The next step is to port it to the embedded system. Two major problems occurred.

One problem is that some functions that worked on the Microsoft Visual Studio do not work on embedded microcontrollers, so some codes needed to be discarded. In order to keep proceeding, many old codes from Mechatronic classes were used. However, many of the files are updated and no longer compatible with the ones used, so a process of understanding the new code, and modified the old ones correspondingly took quite some time.

The second problem is that many code limitation now arise in the embedded system. Due to the limitation of the processing power, it's not efficient to run everything in 32-bit integer when the microcontrollers are only efficient in processing 8-bit data. This may cause slow down of up to 20x in certain processes, so many code optimizations were used. And signed and unsigned data types are used to

increase range of processing while keeping the processing time at the minimum. As a matter of fact, many of the "bugs" in the code that took weeks to resolve are involved with data types and signed and unsigned values. At the end, the codes are successfully written and work well in the testing environment. The following two sections briefly describe the final program in each of the microcontrollers. The actual codes are explained more in depth in Appendix B thru Appendix D.

4.2.1 CBM_sensor

The whole sensor programing is the most critical part of the project. It involves getting data at a fixed interval, compute Integer Fast Fourier Transform, and then sends them out via radio signal. In addition, it also constantly filters data for more accurate results.

The whole code is coded in a format called 'blocking code', meaning that it will not process multiple items simultaneously. It executes a function by a function in an orderly manner. This is usually not the way to program modern microcontrollers. But 'blocking code' can be implemented because all functions here require a previous function to work, so it doesn't really help doing this processing in any other ways. In addition, the program runs a little more efficient in the blocking code manner. The CBM_receiver in the other section, however, is the opposite of using 'blocking code' called the 'state machine'. It will be discussed more in the next section.

4.2.2 CBM_receiver

CBM_receiver is a program to receive data from the CBM_sensor It takes the data that is received wirelessly from the radio modules, processes and sends them via a serial port. The data is then received via USB port of the computer through a serial to USB converter (FT232RL³ is used in this project). Afterwards, a computer will

³From Future Technology Device International Ltd.

process the data received from the serial port terminal.

The program from the CBM_receiver is programmed in a format called 'state machine'. This is a technique that enables processing of multiple items at the same time. How it works is that it processes a very small portion of the code from different section of the program, so it creates a seemingly multiprocessing ability. First, it divides a program into tasks. Within individual tasks, it is future divide into states. Each individual state has to be designed such that it abides by the timing requirement (1ms or less in this case), meaning that it can't take too long to finish. Afterwards, the program will run in a round robin fashion among the various tasks. If all the tasks run fast enough, users will feel that the microcontroller is processing multiple items at the same. Because of this round robin style of code running, programs written with 'state machine' is very easy to add in additional tasks without much alternation to the main code. This is important to the wind turbine project because it will allow processing of multiple radio signals coming from different sensors.

4.3 Serial Port Terminal Program

The final part of the programing comes back to the personal computer. After the data are received in the computer, it needs a good way to view the data. Currently, there are two programs used to view the data in this project: one is a serial port terminal program called TeraTerm, and the other one is called Matlab. TeraTerm just displays the serial port data as it arrives, while Matlab can store the data for future processing. Matlab is better if future processing is needed. However, Matlab requires a little bit more additional programing to view the data, so at the current state, using TeraTerm is an easier option.

Chapter 5

Manufacturing

Manufacturing is a process that makes a material into a useful form. In this project, many of the manufacturing are involved with making cables and circuits to enable different tests.

5.1 Cables and Connector Cables

There are a few cables and connectors made for this project, they are described in the following sections.

5.1.1 Programming Cable

There are two common ways to program the Atmel ATmega microcontrollers, and both are used in this project. So two programmer cables were made: one is for JTAG¹ programming for ET128 board and the other is for the In-Circuit-Serial Programming² (ISP) for the Swoop boards (see figure5.1).

5.1.2 Radio Connector

A radio connector is made for programming the microcontrollers for their radio tasks. The connector is specifically made for ET128 board (see figure5.2) because the Swoop board has a connector on it already, so no extra connector is needed.

¹A very widely used programming interface, and it also supports on chip simulation for debugging

²Default programing interface for all ATmega microcontrollers

Figure 5.1: The JTAG programming cable for ET128 board (left) and the ISP programming cable for the Swoop board (right), they are attached to the programmer called AVR Dragon

Figure 5.2: Radio connector for the ET128 board

5.1.3 Current Tester

A circuit tester is made for each of the microcontroller boards (see figure 5.3). The function of the two circuit testers is to break up the connection, so that a multimeter can be connected in between to measure current draw.

5.2 Circuits

There are two circuits needed for the programs. One is a LED-button for debugging, and the other is a Wein Bridge for generating sine wave.

Figure 5.3: Current tester for ET128 board and Swoop board

5.2.1 LED-button

An LED-button is made for the programs to halt a device as an indicator (see figure 5.4). This is very useful at times when timing requirement does not allow debugging via printing in serial port and JTAG debugging is not available.

Figure 5.4: LED button for troubleshooting

5.2.2 Wein Bridge

Ever since the program can sample acceleration data, a vibration source is always a desirable tool rather than hand shaking the accelerometer. A Wein Bridge is made for this reason (see figure 5.5). It generates a sine wave that will be fed into the microcontroller that simulates a constant vibration signal. In addition, the Wein

Bridge is also made because there is a need to measure the accuracy of IFFT.

Figure 5.5: The actual Wein Bridge circuit

Chapter 6

Testing and Result

Testing is a procedure to confirm a theory. The whole sensing system needs to be tested to ensure it is operating as designed. There are many tests that are done to ensure proper functioning of the system. The tests are explained in the following sections. Afterwards, the results will follow.

6.1 Test plan

The following test plan includes a list of tests. The purpose of the test plan is to state what equipment was used, and the parameters of the tests. The goal of these tests is to simulate vibration of the wind turbine blades and the power usage of the microcontroller.

6.1.1 Continuous Vibration Test

The Continuous Vibration Test examines the microcontroller's ability to detect vibration. This is a very rudimentary way to test the performance of the microcontroller to capture vibration. There are three main reasons why it is different than the real system. First, the vibration magnitude, due to damping, will decrease over time, but this test does not capture that part. Second, the noise level of the signal coming from an accelerometer in a windy environment would be much greater than a fixed frequency vibration in a controlled system. Third, the nature of the vibration is different too. The vibration in this test is forced vibration¹, where as

¹A vibration that is driven by an external force with a particular frequency

the vibration in the actual blade is free vibration². This is important because force vibration may cause resonance, where as the free vibration won't.

Description of procedure

The test will be performed with a shaker in the Vibration Lab in the Mechanical Engineering Department of Cal Poly. The frequency will be set at around 5Hz, 20Hz, 100Hz and 500Hz. It will run until the system gathered 10 results. The collected data will be sent via a radio interface of the microcontroller.

Equipment used

There are three equipment used:

- Shaker
- Swoop board with mount, radio and battery installed that act as the transmitter
- Swoop board with radio and is connected to a laptop for data logging

The shaker is used for generating vibration that simulates natural frequencies. The shaker³ generates its vibration signal from the function generator⁴ through an amplifier⁵. Then the Swoop board (Transmitter) will record the data and send the data via radio wirelessly. After that, the other Swoop board with the radio receiver will get the data and send it to the laptop for data logging.

Mounting and fixtures

The microcontroller will be securely bolted and taped to a mounting plate to reduce the chance of things fall apart under vibration. Then the whole system will

²A vibration due to internal force

³Model VG100-6 from Vibration Test Systems Inc.

⁴Model 182A from WaveTek Corp.

⁵Model CE2000 from Crown

Figure 6.1: Set up for the Continuous Vibration Test

be bolted down to the shaker to get an accurate transmission of vibration to the microcontroller (see figure 6.1 for the setup).

Testing parameter

Here are the testing parameters.

- Input Voltage: 4V that will be regulated down to 3.3V
- Vibration frequency(Hz): 5, 20, 100, 500
- Number of results: 10
- Number of data points: 256
- Sampling frequency(Hz): 7, 56, 280, 1400
- Actual input frequency(Hz)⁶: 5, 20, 100, 550

Expected Outcome

The measured frequency from the Continuous Vibration Test should be very close to the input frequency.

⁶This is the frequency that is input into the system

6.1.2 Disturbance Test- Metal Bar and Wood Bar

The Disturbance Test measures the performance of the sensing system on a cantilever flat beam. This beam is to mimic the actual wind turbine blade. This test simulates a real turbine blade vibration much closer. In an actual system, the wind blown onto the blade will be acting as a force. As the wind changes in speed and direction relative to the blade, the applied force due to the wind will change as well. This change in the force will create vibration of the blade. So by sending a disturbance onto the blade, this will create a good free vibration that the Continuous Vibration Test can't provide.

The second part of the testing is to confirm the fact that frequency of oscillation changes as blade shape change. This is simulating the case for a blade with cracks in it. By purposely reducing the cross section of the blade at a fixed location and then measuring the result, the test will be able to show that a detectable amount of changes in frequency will occur before the blade fails.

Description of Procedure

The test will have a metal bar and wood bar clamped down onto a metal table⁷. The microcontroller will be mounted at the end of the blade with battery attached. A downward point load (a finger push) at the tip will be applied and released and see how the blade vibrates. It will be done five times at each of the three lengths. The lengths will be selected based on whatever the normal vibrating frequencies that seem fit. After the test, the two bars will be shaved off a certain thickness to vary the frequency of oscillation and the test will be repeated again.

⁷Using a metal table to ensure rigidity so that the vibration of the blade will not be affected much

Equipment used

The equipment used is as followed:

- Metal bar and wood Bar
- 2 C-clamps
- Swoop board connected to a laptop for data logging

The metal bar and wood bar will be used in simulating the wind turbine blade. The two C-clamps are used to mount the blade onto the table and be able to use it to shift the length of the bar to create different frequency of oscillation. The Swoop board will get the data and send it to the laptop for data logging.

Mounting and fixtures

The two bars will be clamped down by two C-clamps (see figure 6.2). The Swoop Board will be bolted down to the bars securely to prevent falling off while the system is under testing.

Figure 6.2: Set up for the Disturbance Test

Testing parameter

The test parameters for the Disturbance Test:

- Number of test for each case: 10
- Wood bar dimension (cross section): 1.5in X 1/4in
- Aluminum bar dimension (cross section): 1in X 1/8in
- The 3 lengths of the wooden blade: 8, 10, 15in
- The 3 lengths of metal blade: 7, 9, 12in
- Cut location wooden blade: 7in
- Cut location metal: 6in
- Depth of cut wooden blade: 1/16" , 1/8"
- Depth of cut metal blade: 3/32"

6.1.3 Continuous Sine Wave Test

The continuous sine wave test is a test to measure the precision and accuracy of the FFT results. Getting a result of the final frequency with high resolution is important, but if the machine does not give a precise narrow range of numbers from a given fixed source, the resolution does not mean much. The reason why the Continuous Vibration Test (see 6.1.1) is not sufficient is that mechanical systems generally have a lot of noise. This noise sometimes is from system imbalance. In addition, when the system goes through an accelerometer, the accuracy now is even harder to be measured because extra problems may be caused by the accelerometer. So the test is constructed to test just the accuracy of the FFT process, which in itself is very hard to calculate the accuracy.

Figure 6.3: Microcontroller with Wein Bridge attached

Description of procedure

The test uses sine waves generated from the Wein Bridge (see section 5.2). The output is a pure sine wave that simulates the input of the IFFT. The data will be logged and sent to the computer via the serial interface of the microcontroller.

Equipment used

Equipment only consists of three component:

- Wein Bridge
- Swoop board with radio and battery installed that act as the transmitter
- Swoop board with radio and connect to a laptop for data logging

The function of the Wein Bridge is to send a voltage signal to the Swoop board. Then the transmitter Swoop board will transmit the result via radio. At the end, the receiver will receive the data and send it over the serial interface to the laptop for data logging.

Mounting and fixtures

There is no mounting needed, just a direct connection, see figure 6.3.

Testing parameter

The testing parameters are as followed:

- Voltage input: same as the regulated input to the microcontroller (3.3V)
- Frequency: 4.5Hz and 1000Hz
- Number of data sets: 50

6.1.4 Current Test

The Current Test is a test to measure how much current the microcontroller draws. It is used multiple times to reduce the current usage. This test is very important as some current draw that is not eliminated will significantly reduce the battery life time of the microcontroller. As seen in section 3.5.2, the elimination of one simple component that uses only 375uA will change from battery life time of less than three months to over 30 years. So some current testing is very crucial to the maintenance procedure.

Description of procedure

The process to take current measurements only involves connecting the current tester and the multimeter in current measuring mode. The current draw should be shown on the multimeter.

Equipment used

The equipment used are as followed:

- Current tester
- Multimeter-TaiTan VC97

Figure 6.4: Microcontroller measuring current draw

- Swoop board with various component attached.

The current tester is there to provide a connector for the multimeter to measure the current while the Swoop board is operating with various components turning on and off.

Mounting and fixtures

There is no mounting needed. Only a direct connection is needed, see figure 6.1.4.

Testing parameter

Testing parameter:

- MCU Input voltage: Direct voltage from battery at 3.6V

6.2 Results

The results of the tests performed in according to the above test plan are recorded in the following sections.

6.2.1 Continuous Vibration Test

The following (see Table 6.1 and 6.2) is the result of the Continuous Vibration Test. The input frequency is the frequency measured with the TaiTan VC97. The dial of the frequency generator is set to all 4 of the initial input frequency (500Hz, 100Hz, 20Hz and 5Hz), the sampling frequency is the value from the microcontroller.

Table 6.1: Result from Continuous Vibration Test- Part 1

Input Frequency (Hz)	Sampling Frequency (Hz)	Frequency Detected (Hz)
550.1	1400	593, 593, 592, 592, 592, 592, 592, 592, 593, 592
104.9	280	106.4, 106.4, 106.4, 106.4, 105.6, 105.5, 105.5, 105.5, 105.5, 105.6
20.16	56	20.4, 20.3, 20.4, 20.4, 20.4, 20.3, 20.3, 20.4, 20.3, 20.4
5.508	14	2.976, 2.978, 2.979, 2.979, 2.977, 2.978, 2.981, 2.981, 2.981, 2.981, 2.978, 2.981

Table 6.2: Result from Continuous Vibration Test- Part 2

Input Frequency (Hz)	Average (Hz)	% difference
550.1	592	7.6
104.9	105.9	1.0
20.16	20.4	1.0
5.508	2.979	-45.9

There are two results that are unexpected. One is at 550Hz and the other one is at 5.5Hz. During the testing of the 550Hz, the machine has a rather high pitch sound. This is probably due to mechanical friction, so it could affect the result. However, at the 5.5Hz test, even though the input sine wave is at 5.5Hz, the machine visually does not output 5.5Hz. So the result of around 3Hz actually makes sense.

Table 6.3: Result from Disturbance Test

Material	Test length(in)	Depth of Cut(in)	Data	Average
Wood	15	0	44 45 47 45 46 45 45 45 45 45	45.2
Wood	15	1/16	44 47 46 45 46 45 45 45 47 44	45.2
Wood	15	1/8	44 44 43 44 45 44 42 46 43 45	44.0
Wood	10	0	78 78 79 81 80 79 79 80 81 79	78.4
Wood	10	1/16	79 79 78 79 79 79 78 78 78 77	78.4
Wood	10	1/8	74 75 78 74 77 75 79 73 75 74	75.4
Wood	8	0	94 94 98 93 94 92 92 94 95 92	93.8
Wood	8	1/16	95 95 94 94 94 93 95 95 94 95	94.4
Wood	8	1/8	90 95 90 90 87 127 86 86 89 89	89.1 (eliminated 127)
Aluminum	12	0	70 70 70 69 68 70 68 63 70 70	68.8
Aluminum	12	3/32	70 70 71 75 70 71 63 73 71 74	70.8
Aluminum	9	0	56 56 57 56 57 63 56 59 58 57	57.5
Aluminum	9	3/32	55 54 52 52 53 55 52 54 55 52	53.4
Aluminum	7	0	38 39 38 38 39 36 38 38 39 39	38.2
Aluminum	7	3/32	40 38 38 39 39 39 38 38 39 38	38.6

6.2.2 Disturbance Test- Metal Bar and Wood Bar

Disturbance Test results are recorded below in Table 6.3. All the cuts are smooth to minimize stress concentration. And on the wood, it was only planned for a single cut originally, but no frequency changes are observed, so another cut was made. On the data column, a scaling factor of 0.156 or $40/256$ is needed to convert the number into Hertz.

Table 6.4: Summary from Disturbance Test

Material	Test length(in)	Depth of Cut(in)	% Frequency Difference
Wood	15	1/8	-2.7
Wood	10	1/8	-3.8
Wood	8	1/8	-5.0
Aluminum	12	3/32	+2.9
Aluminum	9	3/32	-7.1
Aluminum	7	3/32	+1.0

Overall, this table shows depth of cut does not have significant effect on frequency of oscillation. Although there is a perceivable amount of frequency differences, two problems associated to make a call on the fact that depth of cut being related to frequency of oscillation. First, the depth of cut is very deep of up to $1/2$ of the blade's thickness⁸ to yield these small amount of frequency changes. Second, there is an oscillation increase even the depth of cut is quite deep. This can only be a result of the fact that the differences are so small that it is still within the margin of error. So even a decrease in 7% of frequency of oscillation still is not justified to claim the frequency of oscillation is associated with depth of cut.

⁸For the wood blade, the cut is $1/2$ of the blade's thickness, whereas for the aluminum blade, the cut is $3/8$ of the blade's thickness.

6.2.3 Continuous Sine Wave Test

Although the test originally specified 4.5Hz and 1000Hz as the frequency, due to component malfunctioning, only one trial test is done at around 50Hz (see-table 6.5.

Table 6.5: Result from Continuous Sine Wave Test

Number of Data Points	Average (Hz)	Standard Deviation (Hz)
50	56.8	0.39

Although the two extreme cases can't be tested, but the data here shows the output of the Integer Fast Fourier Transform is pretty high in precision.

6.2.4 Current Test

The Current Test results are followed. There are two sets of data for minimum power usage because there is an unknown 482uA of current draw somewhere, so another board is used to investigate the problem. Interestingly, the current draw from the second board got down to 1.3uA with the same code, which means that there is a 481uA of current difference. The two boards only differ in two minor components. One is the crystal frequency ⁹ and the other one is the accelerometer ¹⁰. The frequency of crystal should not affect the current draw because in power save mode, the crystal is not in used. This points to a problem with the accelerometer, even the datasheet for accelerometer said 375uA of current is used. Later on, the problem is found to be having a short circuit of the accelerometer to ground. This makes sense because a short circuit will draw a lot more current than designed.

After switching to a board without accelerometer, as seen in Table 6.7, the current draw significantly reduces to only 1.3uA. The next step is to get the minimum current with the radio chip, the result is in Table 6.8.

⁹The first one is 4MHz, and the second one is 10MHz

¹⁰First board has an accelerometer, but the second board does not

Table 6.6: Minimum current of microcontroller for Swoop board that has an accelerometer and 4MHz crystal during power save mode

Action	Current (uA)
Remove radio	6520
Remove all LED resistors	2098
Pull all unused pins to ground	1952
Power supply at 3.3V and bypass FT232R USB/regulator chip	1690
Disable USART receive and transmit	1615
Disable DIDR1	1596
Disable DIDR0	1454
Disable JTAG (3.3V minimum configuration)	482

Table 6.7: Minimum current of microcontroller for Swoop Board that does not have accelerometer and has 10MHz crystal during power save mode

Action	Current (uA)
Use the same code from the (3.3V minimum configuration)	1.3
Change voltage to 1.8V	0.8

The radio chip has a specification of using only 0.9uA of current during power down. However, it shows that the number is a lot bigger than 0.9uA in the test. As a matter of fact, it seems like it uses about 130uA. This is probably due to the quiescent current¹¹ of the regulator used on the 24L01 board, so for a minimum current draw, the voltage regulator of the radio chip needs to be removed.

¹¹Minimum current use of an electronic

Table 6.8: Current usage with microcontroller with 4MHz crystal

Action	Current (uA)
3.3V minimum configuration	482
Plug in Wein Bridge and radio chip	13000
Disable radio chip (CE=0)	1994
Power down radio chip	1981
Unplug Wein Bridge (minimum configuration with radio)	610

6.3 Uncertainty of the Testing

One of the few things that is not being tested by the microcontroller is the ability to measure the acceleration of free vibration at a high frequency. This is due to the inability to set up the test due to lack of resources. As the frequency of oscillation increases to a high number (around 300Hz), the vibration becomes extremely hard to be visible. In addition, that high of frequency can be fully damped out easily before the microcontroller can register any data.

Although the test shows that the microcontroller can measure frequency even at 590Hz, but this is from pure force vibration. Since measurement of free vibration at a high frequency is really hard to measure, there is a concern of whether the microcontroller can register acceleration of the wind turbine blade at such frequency. Especially in the condition that magnitude of vibration will damp out extremely fast in high frequency of oscillation, so the microcontroller may easily miss the vibration.

Chapter 7

Conclusion

All the tests together have shown the project has successfully produced a monitoring device for vibration detection. It has shown to be able to detect vibration even up to 600Hz. However, since no high frequency disturbance test is done, the project will not have any knowledge of performance on free vibration over 20Hz. One concern is the outcome from the disturbance test, which states that reduced cross section on a specific place on a solid wooden, and aluminum blade will not significantly change the frequency of oscillation. Further investigation on this subject matter is needed in the future projects.

Chapter 8

Unresolved Problems and Upgrades

There are a few items that can be developed further in the future. They are categorized either into unresolved problems or future upgrades.

8.1 Unresolved Problems

There is still a problem that hasn't been resolved by the end of this project, it is finding the correlation between the residual life and natural frequency. Hopefully future development will be successful in getting such information.

8.2 Upgrades

There are many upgrades that are thought of as the project progresses. They are in three different categories: programming, mechanical construction and power consumption.

8.2.1 Programming

In programming, there are four items that could be worked on in the future:

- Use a more efficient radio code by sending bytes of information instead of a hex number for each data

- Change some existing code to enable auto-ranging of finding the natural frequency to optimize the resolution, so guessing of natural frequency can be eliminated
- Use faster FFT routine
- Enable Zero Padding (see section 2.9.4) for the IFFT. This will allow user to freely use the amount of data points to get the optimal detail from the IFFT rather than being limited by the 'powers of 2' requirement

8.2.2 Mechanical Construction

In mechanical construction, there are two items that could be worked on:

- Shielding the battery, so it won't be heated up during the day, yet don't fully seal the battery because the batteries need vents to operate safely. This is important because lowering heat can drastically elongate the amount of time the battery can be in service.
- Avoid getting rain/water onto the batteries.

8.2.3 Power Consumption

There are five items that could be researched to further reduce the power consumption of the device:

- Remove the regulator in the 24L01 chip.
- Use an accelerometer that has a shut off mode, or relocate the accelerometer so that it is powered by a port output, so it can power up the accelerometer from the output pins.

- Power the 24L01 chip via the output pins. Currently it is statically drawing 0.9uA of current at shut down mode, so if the microcontroller directly gives power to the radio chip, it will save a lot of power.
- Use ATxmega instead of ATmega microcontrollers. ATxmega has longer count of sleep and thus the microcontroller does not need to wake up as often, so it will save a lot of power consumption. Currently, this is the second highest current draw operation.
- If lithium battery is used, a diode can be placed in the circuit in line with the battery, so it will reduce voltage going into the microcontroller and all other components. Since diodes do not consume energy besides causing a voltage drop, this method can save power consumption because many of the components here will draw a lot less power when the voltage is lowered. The down side is that the battery will consider fully drained at about 2.5Volt (this is because the minimum voltage required for the microcontroller to be in operation is 1.8v and with the 0.7 volt of turn on voltage of diodes). But from the discharge curve of L91, the voltage 1.3V is at the last 5% of the capacity at 20mA (the current maximum drain is about 15mA during full operation) of drain rate. The current saved from various components are way more than 5%, so this is a worthwhile consideration.

References

- [Ambani et al., 2009] Ambani, S., Li, L., and Ni, J. (2009). Condition- based maintenance decision-making for multiple machine systems. *Journal of Manufacturing Science and Engineering*, 131.
- [AnalogDevices, 2006] AnalogDevices (2006). *ADXL330 Accelerometer datasheet*. Analog Devices.
- [Atmel, 2010] Atmel (2010). *ATmega324P/V datasheet*. Atmel.
- [Blankinship, 2008] Blankinship, S. (2008). Keeping wind turbines spinning. *Power Engineering*, August:50–52.
- [Brenda, 2010] Brenda (2010). Summary of wind turbine accident data to 2010. Technical report, Caithness Windfarm Information Forum.
- [Ciang et al., 2008] Ciang, C., Lee, J., and Bang, H. (2008). Structural health monitoring for a wind turbine system: A review of damage detection methods. *Measurement Science and Technology*, 19.
- [Edwards, 2009] Edwards, B. (2009). Composite manufacturing of small wind turbine blades. Technical report, Mechanical Engineering Department of California Polytechnic University- San Luis Obispo.
- [Energizer, 2009] Energizer (2009). *Energizer E91 datasheet*. Energizer.
- [Energizer, 2010a] Energizer (2010a). *Energizer NH15-2300 datasheet*. Energizer.
- [Energizer, 2010b] Energizer (2010b). *Lithium Iron Disulfide Handbook and Application Manual*. Energizer Battery Manufacturing Inc., li3.01 edition.

- [Flemming and Troels, 2003] Flemming, M. L. and Troels, S. (2003). New lightning qualification test procedure for large wind turbine blades. In *Int. Conf. Lightning and Static Electricity (Blackpool, UK)* pp 36.1-10.
- [Harris, 2003] Harris, B. (2003). *Fatigue in Composite*. WoodHead Publishing Ltd.
- [Myrnyy, 2007] Myrnyy, V. (2007). A simple and efficient fft implementation in c++.
- [Panasonic, 2008] Panasonic (2008). *Panasonic CGR18650CG datasheet*. Panasonic.

Appendix A

Battery Comparison

As technological improvement progresses, more and more portable equipment are getting used everyday. This leads to a strain in the portable power storage device. Battery is the major device that people use because of its size, weight and energy density. This appendix will compare a few of the battery technologies and explain why lithium battery is selected as the power supply in the sensing system.

A.1 Battery Specification

There are two types of battery: rechargeable and non-rechargeable (primary) battery. The common high capacity rechargeable batteries are lithium-ion and nickel metal hydride batteries, where as the common high capacity primary batteries are lithium and alkaline batteries.

Table A.1 is a list of some very basic characteristics of the batteries. The values are only valid when draining are occurred at about 25mA or lower.

A.2 Weight

Lithium battery is very light. Two AA batteries weigh less than one AA NiMH battery. In addition, the output capacity is extremely well. So it is very advantageous to use a lithium AA battery because it has a much higher capacity/weight ratio.

Table A.1: Batteries Comparison

Battery Name	weight(g)	Voltage (V)	Capacity (mAh)	Shelf Life (years)	Rechargeable?
Lithium [Energizer, 2010b]	14.5	1.5	3000	15	no
Alkaline [Energizer, 2009]	23	1.5	2800	7	no
Lithium-ion (18650) [Panasonic, 2008]	45	3.6	2250	n/a	yes
Nickel Metal Hy- dride [Energizer, 2010a]	30	1.2	2300	n/a	yes

A.3 Current Usage

The capacity is greatly affected by the rate of current drain. The higher the current drain rate, the more loss in capacity for a given battery. The equation from Ohm's law $P = I^2/R$ states that the higher the current draw, the power dissipation by the internal resistance is higher by a factor of squared. In addition, the voltage that is available will decrease as the current draw is high. This is due to Ohm's law $V_{internal_resistance} = IR_{internal_resistance}$ as some of the voltage will be lost due to the internal resistance of batteries.

On the other hand, with a low current drain rate, the voltage of a battery will remain quite high. The voltage would go to as high as 1.8V for a lithium battery (see figure A.1), 1.6V for an alkaline battery, 1.4V for a NiMH battery and as high as 4.2V for a lithium-ion battery.

Figure A.1: Discharge curve of L91 lithium battery at 1ma

A.4 Self Discharge Rate

Self discharge rate is the rate of how fast a battery loses its capacity. Over years, most batteries will lose almost their whole charge even when they are not being used. However, Energizer claims L91 batteries for 15 years of shelf life. The self discharged rate at 40 degrees Celsius is about 20% capacity lost in 15 years (see figure A.2). This is definitely a quality that this project wants.

Figure A.2: Shelf life of L91 with different temperature

Appendix B

User Manual

The following sections describe the output results of the sensing system and what users can control in this program.

B.1 Wireless Output

There are three main stages that the wireless module sends data to the receiver. The first stage (Initialize) is in the beginning of the program to let the user know that the sensing system started. The second stage (Getting w_{ref}) is at the end of a sampling set to find a specific w to set as a reference frequency of oscillation. If a proper reference frequency of oscillation is found, it'll go into the third stage. If a proper one is not found, it will go back to re-sampled again. The third stage (Normal) is the normal operating stage. It will keep comparing to the reference w to see whether it has exceeded the requirement (currently it is set at 8%). The following table B.1 shows different stages and its outputs.

Here is a quick explanation to everyone of them. A value of '0' means that the slot is not being used yet, and is available to use for further expansion.

- status- it has 6 status total:
 - 0- Initializing
 - 1- Getting reference frequency of oscillation to set it as the blade being 'new'
 - 2- In healthy regular operation (what it should be in most of the time)

Table B.1: MCU radio output at different stages

Stage\Info	0	1	2	3	4	5	6	7
Initialize	status	#FFT/ cycle	#data_fil	#min err	freq sample	#samples	noise	0
Getting w_ref	status	w_ref	next_time	#init_ref	#min_ref	%ref_tol	0	0
Normal	status	w_avg	next_time	w_ref	%failure	%data_tol	#cycle	0

- 3- Need attention because it has mixed natural frequency and is hard to determine whether it has failed or not
- 4- Blade failure, the frequency of oscillation is below the failure criteria percentage
- 5- 0Hz frequency of oscillation, the blade could be broken off
- #FFT/cycle - The amount of FFTs there are in a given cycle. It is used to determine a cycle average frequency of oscillation. The higher the number, the more data points to determine the cycle average frequency of oscillation, but it also uses a lot more system resources to calculate it. A set that contains a few of these average cycle frequencies will be used to determine whether the blade is healthy or not
- #data_fil- The amount of average cycle frequencies used to determine whether the blade is healthy
- #min err- The amount of average cycle frequencies that needs to be within a certain tolerance to consider whether the blade is healthy or not
- freq_sample- The sampling frequency, due to the Nyquist frequency require-

ment, this number needs to be at least double of the frequency the user wanted to measure

- `#samples`- The number of data to sample in each Integer Fast Fourier Transform
- `noise`- Noise filtering, a frequency needs to be higher than this threshold in order to be considered as a candidate as the frequency of oscillation
- `w_ref` - The reference frequency of oscillation, or the frequency of oscillation of the blade being new.
- `next_time`- The next time mcu will send out a radio signal in seconds
- `#init_ref`- The amount of average cycle frequencies used to determine the reference frequency of oscillation, or the frequency of oscillation of the blade being new
- `#min_ref`- The amount of average cycle frequencies that needs to be within a certain tolerance to be used to determine the reference frequency of oscillation, or the frequency of oscillation of the blade being new.
- `%ref_tol`- The percentage (in 0.1%) tolerance or precision that the cycle average frequency of oscillation needed to be in for it to calculate reference frequency of oscillation
- `w_avg`- the average frequency of oscillation of the whole set of data
- `%failure`- the percentage (in 0.1%) that the blade reaches to consider it being failed
- `%data_tol`- The percentage (in 0.1%) tolerance or precision that the cycle average frequency of oscillation needed to be in for it to calculate the frequency of oscillation of the blade that determines whether the blade is healthy or not

- `#cycle`- A running counter to keep track of how many cycles have the system run already

B.2 Control

There are a few things users can alter in the program to suit the need. All the user controllable constants are located in the header file called `CBM_v1_0.h` (The file is in Appendix D). The header file is broken up into sections according to different files. Within each file, there is a section in which the pre-processor constants¹ can be altered and there is another section that constants cannot be altered. An explanation of each pre-processor constant is commented on a side inside the header file.

¹Preprocessor constants are constants that are given meaningful names. The values can be altered before compilation, but remain the same throughout the run time of the mcu

Appendix C

Programming Code Description

The following sections describe the program in each of the microcontrollers. One is referred to as CBM Sensor while the other is referred to as CBM Receiver. In addition, individual files within each microcontrollers and their main functionalities are explained. The actual file is located in the Appendix D.

C.1 CBM Sensor

There are a lot of major files in CBM Sensor:

- `CBM_v1_0.cpp`
- `CBM_v1_0.h`
- `fft_int.cpp` and `fft_int.h`
- `sensor.cpp` and `sensor.h`
- `avr_sleep.cpp` and `avr_sleep.h`
- `nRF24L01_base.cpp` and `nRF24L01_base.h`
- `nRF24L01_secure.cpp` and `nRF24L01_secure.h`

Together they enable the sensing system to function.

CBM_v1_0.cpp

The CBM_v1_0.cpp is a file that is the backbone of the whole sensing program. For simplicity of describing individual portions of what the program does, the whole file will be discussed in four sections: initialization, to obtain reference frequency of oscillation, to obtain frequency of oscillation for failure detection, and data filtering.

Initialization The initialization involves a process that setup modules and functions of the program for used later. First, five conditions of the blade are defined there. Second, it sets up all the input and output pins of the microcontroller and pulls them to output, so no pins are floating¹ to ensure the lowest possible power draw from the microcontroller. Third, it defines instances² of the classes used, so the functions of all those classes can be used later. Finally, it displays debugging information about the setting of the microcontroller via a serial interface (This function will not be used in the actual sensing system, but it is a good function to retain for future debugging purposes).

To obtain reference frequency of oscillation After initialization, The microcontroller executes a routine that is very similar for the rest of the program.

It starts to routinely sample the acceleration data points that the user specified. Then, it processes Fast Fourier Transform to get the data into the frequency domain. From that point, the program picks out the top peaks and store three of them into an array. This process repeats until the array reaches the selected amount of times. The array will be filled and needed to be filtered and evaluated to find a good average. This average is the set average. It should give the user a rather consistent number that depicts the frequency of oscillation.

¹A state that has a fluctuating voltage because no voltage is defined. Floating pins usually draws more power than non-floating pins when it is not used

²A variable that contains a structure of functions

The program will stop taking data until eight sets (user selectable) of averages are obtained. If there are at least five (user selectable) of the eight sets of data that are close to each other within tolerance, the program will take an average of those number and that will be the cycle average. If there are less than 5 of them within tolerance, the program will retake those 8 sets of numbers again due to accuracy issues (for more information of that particular process, please see section below called 'cycle filtering averaging function').

A number that is high in precision is desirable for the reference because that average will be use as the reference frequency for 100% healthy blade. After the cycle average is found, the whole preface of the program is done, and will go into the main loop. The program will then send a radio signal to let the user know the status.

To obtain frequency of oscillation for failure detection The process of obtaining the frequency of oscillation for failure detection is very similar to the process to get the reference frequency of oscillation. The only difference is that now it will compare to the failure percentage every time it gets a good average from the cycle filtering and averaging. If the number is above the failing criteria, then it is healthy. If the number is 0, then the blade could be broken off or some malfunction of the wind turbine blades. If the number is below the failure criteria, then the blade needs to be serviced or replaced.

Data filtering function The filtering function does the filtering and averaging when a certain amount of sets of frequency of oscillation is found. The function mainly determines how consistent are those set averages.

First, The process finds a highest and lowest number among the sets of numbers (8 if the process is in the preface to find the reference frequency). Those two numbers will be compared to the average and see which one is further away, and set that

number as the index (a number to delete if needed). Now individual set numbers will be compared to the average, if the number is out of tolerance, the index will be deleted (not the number that is out of tolerance). The new average is found again, and the process will restart with one less number. By this process, the remaining numbers should have higher and higher precision because one of the extreme numbers is deleted every time. If all numbers are within tolerance and above the minimum, the program will use the average of these numbers as the number for either reference frequency of oscillation or number to compare to the failure criteria.

CBM_v1_0.h

CBM_v1_0.h is a header file that is used by all the major files of the CBM sensing program. The header file contains many important constants for each individual file. Those constants are very important because they are user selectable to alter the program to fit a particular need. For example, assuming the wind turbine blade's natural frequency is 10Hz. It can be adjusted in this header file so that the program can sample at a rate appropriate to find the actual frequency of oscillation. If the program only needs 128 data points as the resolution, then it can change the value and make the processing even faster. In addition to constants, there are also macros³. So this header file is very important for user to fine tune the program for specific uses.

fft_int.cpp and fft_int.h

fft_int.cpp and fft_int.h contain a list of functions that is used by the CBM_v1_0.cpp to process FFT and getting precise numbers for frequency of oscillation, it has seven major functions.

³Values that are preprocessed by the computer before downloaded into the microcontroller

go() The most important function is the `go()` function. It processes a set of integer time domain data through binary inversion⁴ and then through Danielson-Lanzcos routine⁵ to find the FFT, which is a set of data in frequency domain.

get_sintab() A scaled up sine table (by a factor of 32768) with only integer as values is used in place of regular sine function to save processing time. In situations where the table is small, it can be saved in default location, which is RAM. However, 2048 data points are rather huge, and this will overflow⁶ the RAM, so all 2048 values are saved into the programming memory (see section 2.6.1). The function `get_sintab` is used for getting those values.

top_freq() `top_freq()` is a function that gets two peak frequencies for future evaluation. First, it converts all values of the FFT to positive numbers, then the program goes through the whole frequency spectrum to get the top 10 frequencies from the FFT. After that, the function isolates the peaks by deleting the neighboring peaks that have lower magnitudes in order to avoid duplication of the same peak. The function deletes data that is smaller than a threshold (usually noise) as well as the value at 0 (which is a DC constant voltage value). In order to let the main function to know when to take averages, the program also increment a counter to count the number of FFT done at the end of finding the top two frequencies.

store_fft_cycle() `store_fft_cycle()` is a function to store two peaks' value from the `top_freq()` function as an array for future calculation of average frequency of oscillation. In addition, this function also stores the values that are in the direct neighborhood of the peak frequencies of each IFFT, which will be used later to refine the cycle average.

⁴A data swapping process for an efficient process for FFT

⁵A process for calculating FFT

⁶Exceed it's capacity

get_avg_freq() `get_avg_freq()` is a function that uses the array of frequencies from `store_fft_cycle()` to calculate the set average frequency of oscillation of the system. The process starts with dividing the whole spectrum of frequency domain into as much as 128 bins. Then the program sorts the array of peak frequencies into different bins. The bin with the most data, along with the two neighboring bins, will be used to calculate the average frequency of oscillation. If there are more than one bin ties for having the most data, this function will start a special process to determine which is the bin to calculate the frequency of oscillation. In addition, standard deviation can also be determined in this function.

cycle_reset() `cycle_reset()` is a function that resets the FFT counter, so that the main function can have controls to when the program restarts taking data for the cycle.

isqrt() `'isqrt()'` is a long integer square root function that determines the square root of integers, this is important for calculating standard deviation in the above `get_avg_freq()`.

sensor.cpp and sensor.h

`sensor.cpp` and `sensor.h` are files that contain functions for used by `CBM_v1.0.cpp` to record data for the IFFT. All functions can be separated by four parts: Setup, read sensor, flags and interrupt service subroutine.

Setup The main part here is to set up both the analog to digital converter to get data from the accelerometer and the timer for counting at a fixed interval for time precision sampling of the accelerometer data.

The ADC is set to read a 8-bit number for calculation. The ADC is capable of getting 10-bit resolution data (4x more in resolution). But in order to use 10-bit resolution, either the program will not be able to process as many data points, or

the speed of processing will be significantly reduced in order to process the same amount of data. So a 8 bit sampling is used.

The timer is set up to interrupt at an interval automatically determined by the sampling frequency. The trigger source of the interrupt is an output compare match⁷ with the value set by the sampling frequency.

read sensor Read sensor is a set of functions that start the reading of the three axis accelerometer. The project currently is designed for using only one axis, but since the price of the 3-axis accelerometer is very close to the single axis variety, the project uses a 3-axis one instead. Future upgrade may use all 3-axis, so three functions to read all three axis are created. The functions start the reading process and reset values of all variables used in the interrupt.

flags There are three flags that are used from the CBM_v1.0.cpp to get the state or reset the state of the sensor. They are the only means (besides the header file) that other files in this project can have effect on these sensing files.

interrupt service routine The interrupt service routine in the sensor files is very important. It gets called at a fixed time interval to wake up the microcontroller from the power reduction idle mode to sample data. It is a very short routine that either records data, and or sets up for recording data. The recording stops once the recording counter reaches the number of data points required for the IFFT.

avr_sleep.cpp and avr_sleep.h

The avr_sleep.cpp and avr_sleep.h are files that implement power reduction sleep routines and shut down components to get the minimal amount of power usage during sleep. Since only idle mode and power save mode are used, these two modes

⁷When the number of the continuous incrementing timer and the number set by the user become the same

are specifically catered to the CBM sensing program for a minimum power draw. The files consist of three parts: setup, sleep modes and interrupt service routine.

Setup The set up of the `avr_sleep` sets the microcontroller to enable the use of external Real Time Clock (RTC) crystal for an accurate time measurement and low power draw during the power save mode. It is set to wake up at every eight seconds.

sleep modes The microcontroller has six sleeping modes. They are modes that turn off different amount of peripherals in order to achieve the minimal power usage in sleep while keeping proper peripherals in operation. It is critical to turn off as many components as possible during sleep. As seen in the power supply section (see section 3.5.2) , a 375uA (0.000375A) current draw can reduce the amount of time of operation from 15years to less than three months. So many of the codes are here to ensure components are really shut off during sleep.

interrupt service routine The interrupt service routine in the `avr_sleep` files is used to count every eight seconds. This number is used to compare with the interval set by the `CBM_v1_0.cpp`. In this case for this project, at 225 counts of eight seconds (30minutes), the program will start getting and processing the data to get the frequency of oscillation.

nRF24L01_base.cpp and nRF24L01_base.h

The `nRF24L01_base.cpp` and `nRF24L01_base.h` are files that operate the major functions of the nRF24L01 radio chip that is used in this project. It was originally created by Minh Nhat Le, later being modified by Samuel Hoffman and Professor John Ridgely. This project modified it further and added the functions: `power_down()` and `power_up()`. The function `power_down()` turns off the radio modules to the lowest possible state (in this case, only drawing 0.9uA of current). The function

`power_up()` turns the modulus back on and does nothing. The reason why the function `power_up()` is there because there is a start up delay time for the radio modules to warm up in order for it to function properly. As a result, `CBM_v1_0.cpp` needs to call this function ahead of time, so that it will work later to send or receive data.

`nRF24L01_text.cpp` and `nRF24L01_text.h`

The `nRF24L01_text.cpp` and `nRF24L01_text.h` are files that do the major sending and receiving of the radio module. It is a child class of the parent `nRF24L01_base` files that uses all the functions in the base class for sending and receiving radio signals. The files are originally created by Minh Nhat Le and is modified later by Samuel Hoffman and Professor John Ridgely. Now, it is further modified in this project to make it works with the ATmega128⁸ for testing up to 1024 data points. This is for the future adaptation when ATmega644P is available.

`nRF24L01_secure.cpp` and `nRF24L01_secure.h`

The `nRF24L01_secure.cpp` and `nRF24L01_secure.h` are files that add functionality to the radio sending process. It automates three things: packet making⁹, packet sending and packet confirmation¹⁰. These two files were originally created by Mario Garcia, and it is adapted to be used in the sensing system project. Two functions are added, they are `quick_packet_send()` and `quick_send()`.

⁸ATmega128 has 4KB of RAM instead of 2KB. An identical chip with 4KB of RAM named ATmega644P is considered as an upgrade. But due to unavailability of the chip, so ATmega128 is used in place for testing.

⁹A process to send a large set of data at once rather than sending individual data out every time. It saves processing power and time by eliminates sending repeated identifying information that is required by the receiver

¹⁰Via check sum- a process that first adds up all the numbers and characters in hex code and then sends that information out. The accuracy of the data will be checked by the receiving side by the same process. Check sum drastically increase the accuracy of the data sending and receiving

quick_packet_send() and quick_send() `quick_packet_send()` is a function that sends information as a packet out. This function requires all eight integer data (maximum number of data in a packet) to be available. The function makes them into a packet and calls `quick_send()` to send the packet out. One advantage of this method over what Mr. Garcia created is the ability to send a packet immediately that is created on the spot without the need to wait.

C.2 CBM Receiver

There are four major files (along with their corresponding header files): `CBM_Base_Receiver`, `task_receiver`, `nRF24L01_base` and `nRF24L01_text`. Both the `CBM_Base_Receiver` and `task_receiver` were originally created by Mario Garcia named `receiver_test` and `task_receiver` (same name). Both are slightly modified to adapt to new files and change the way how they use RAM in some functions. The two receiver files will be discussed in a little more in the two subsections. The other two codes from radio modules are the same as the one used in the `CBM_v1.0` files (see section C.1).

CBM_Base_Receiver.cpp

`CBM_Base_Receiver.cpp` is the backbone main program that gets all the data wirelessly and sends the result to the serial port. The major two parts are setting up and running tasks in the format of finite state machine.

During the setup, the program defines the pins for the radio modules and defines objects needed to get the main loop running. The program also displays some useful debugging information via a serial port as well. After the set up, the microcontroller will be continuously running through the main loop. However, since there is only one task currently, namely `task_receiver()`, it will just keep executing that task.

task_receiver

The `task_receiver.cpp` and `task_receiver.h` are files that contain a task that runs the radio receiver. The task has three states and along with another function that converts data from the radio to integers. The 3 states are: `GET_DATA`, `CHECK_DATA`, and `UPDATE_GUI`. `GET_DATA` packs every four data from the receiver together. The `CHECK_DATA` converts the four data into an integer via the integer conversion function at the end of the file. In addition it also checks whether the check sum gets a correct value or not. `UPDATE_GUI` is a state that sends the data to the serial port for logging and condition based maintenance.

Appendix D

Programming Code

The following pages contain codes that make this sensing system work. If interested in how the code generally functions, it is explained in Appendix C.

```

//=====
/** \file   CBM_setup.cpp
 *    This is the main file for CBM
 *
 *    Revisions:
 *    \li   10-01-10 CKC initialized
 *    \li   06-21-11 CKC Revised to full commented the code
 *
 *    License:
 *    This file released under the Lesser GNU Public License. The program is intended for
 *    educational use only, but its use is not restricted thereto.
 */
//=====

#include <stdlib.h>           // Standard C library
#include <avr/io.h>           // Needed it to use pgmspace
#include <avr/pgmspace.h>     // Needed in order to use program memory for variables

#include "lib/rs232int.h"     // Serial interface
#include "lib/global_debug.h" // Header for serial debug
#include "lib/stl_timer.h"    // Just for testing about time
#include "fft_int.h"          // Integer Fast Fourier Transform functions
#include "sensor.h"           // Sensor functions
#include "lib/queue.h"        // For Radio
#include "nRF24L01_text.h"    // Text in Radio
#include "nRF24L01_secure.h"  // Basic data packing for radio
#include "avrsleep.h"         // Sleep functions for low power draw
#include "CBM_v1_0.h"         // Contains alterable parameters for sensing system

/////*****All the following are all included inside CBM_v1_0.h, if this file is used
// seperately, please change the following value accordingly *****
#define NUM_SAMPLES 256
#define SAMPLE_FREQUENCY 30
#define BIN_WIDTH_MHZ SAMPLE_FREQUENCY*1000L/NUM_SAMPLES
#define SAMPLE_CYCLE_TIME NUM_SAMPLES*1000L/SAMPLE_FREQUENCY

// Atmega128
#ifdef __AVR_ATmega128__
// #define CE_MASK      0x10
// #define CSN_MASK     0x20

// Atmega324p, should also be good for Atmega644,
//please add other definitions if another mcu chip is used
#else
// #define CE_MASK      0x08
// #define CSN_MASK     0x10
#endif

#define FFT_PER_CYCLE 3
//Normally about 150 is usually what the max value can get
#define NOISE_THRESHOLD 50
#define QUEUE_SIZE      200           // The size of the queue

```



```
//#define NUM_INIT_REF      8
//#define MIN_NUM_REF        5
//#define NUM_ERR_REF        6
//#define MIN_ERR_REF        4

// Absolute Tolerance in 0.1% increment, so 1025 means 102.5% or 2.5% above normal and
// 975 means 97.5% or 2.5% below normal
//#define W_REF_MAX_TOL      1025
//#define W_REF_MIN_TOL      975

//#define ERR_MAX_TOL        1025
//#define ERR_MIN_TOL        975

//#define FAILURE_PERCENT    920

//The time between individual fft in x8sec. For example: vale of
// 3    means 24sec
// 15    means 2mins
// 150   means 20mins
// 450   means 1 hrs
//#define FFT_INTERVAL8X     150

//CBM_STATUS
//Initializing
#define CBM_INI              0
//Getting w_ref
#define CBM_GET_W_REF        1
//In good regular operation (what it should be in most of the time)
#define CBM_REG_OP           2
//Need attention because it has mixed natural frequencies and is hard to determine failure
#define CBM_NEED_ATTENTION    3
//Blade failure, it reached below the failure criteria percentage
#define CBM_ERROR            4
//It is currently having 0 Hz frequency of oscillation, the blade could be broken off
#define CBM_ZERO_W           5

//==== Global variables ====

//a variable from Sensor.cpp
extern short data2[NUM_SAMPLES];
//header declaration for a function defined at the end of this file for filtering
//frequencies
bool get_filtered_w (short w[],short* p_avg, char max_num, char min_num, short max_tol,
    short min_tol);

int main ()
{
```

```

//initial status
char cbm_status=CBM_INI;

#if defined __AVR_ATmega128__
    //make sure all ports are default output to save the maximum power during sleep
    DDRA=0b11111111;
    DDRB=(DDRB|0b11000001);
    DDRC=0b11111111;
    DDRD=(DDRD|0b11110010);
    DDRE=(DDRE|0b11111100);
    DDRF=(DDRF|0b00001100);
    DDRG=(DDRG|0b11100111);

#elif defined __AVR_ATmega644__ || defined __AVR_ATmega324P__
    DDRA=0b00001111;
    DDRB=0b10111011;
    DDRC=0b00111111;
    DDRD=0b11111011;

#endif

// make the serial transfer rate 9600 brad and port 0 as the serial output port
rs232 the_serial_port (BRAD_RATE, SERIAL_PORT);
set_glob_debug_port (&the_serial_port);
GLOB_DEBUG (PMS ("\nSerial setup done @Main\r"));
GLOB_DEBUG (PMS ("GLOB_DEBUG: set up done!\n\r"));

GLOB_DEBUG (PMS ("*** Entry point of the program ***\n\r"));
GLOB_DEBUG (PMS ("\n\r"));
GLOB_DEBUG (PMS ("MCU:\t\t\t"));

#if defined __AVR_ATmega128__
    GLOB_DEBUG (PMS ("Atmega128\n\r"));
#elif defined __AVR_ATmega644__
    GLOB_DEBUG (PMS ("Atmega644\n\r"));
#elif defined __AVR_ATmega324P__
    GLOB_DEBUG (PMS ("Atmega324P\n\r"));
#endif

//print a list of debugging informations
GLOB_DEBUG (PMS ("Crytal freq:\t\t") <<F_CPU <<PMS (" Hz\n\r"));
GLOB_DEBUG (PMS ("Number of data:\t\t") <<NUM_SAMPLES <<PMS ("\n\r"));
GLOB_DEBUG (PMS ("Sample Freq:\t\t") <<SAMPLE_FREQUENCY <<PMS (" Hz\n\r"));
GLOB_DEBUG (PMS ("Freq Resolution:\t") <<BIN_WIDTH_HZ <<HZ_STRING<<PMS ("\n\r"));
GLOB_DEBUG (PMS ("Sample Cycle time:\t") <<SAMPLE_CYCLE_TIME <<PMS (" mS per FFT\n\r"));
GLOB_DEBUG (PMS ("Noise Threshold:\t") <<NOISE_THRESHOLD <<PMS ("\n\r"));
GLOB_DEBUG (PMS ("FFT per cycle:\t\t") <<FFT_PER_CYCLE <<PMS (" per cycle\n\r"));
GLOB_DEBUG (PMS ("\n\r"));

```

```

//Create an Integer FFT
fft_int my_fft(&the_serial_port);
//declare an instance of Sensor
sensor my_sensor(&the_serial_port);

//declare an instance of Radio text
nRF24L01_text my_radio (PORTB, DDRB, CE_MASK, PORTB, DDRB, CSN_MASK, &the_serial_port);
    my_radio.enhanced_mode ();
    my_radio.reset ();

// Set the radio to receive signals on channel 0x42 using Pipe 0
my_radio.set_RX_address (0x5A697840L);
my_radio.set_TX_address (0x5A697840L);

// Create a circular buffer to holds integers saved from the ADC
queue<unsigned int, unsigned int, QUEUE_SIZE> my_queue;

// Create a secure radio communication object
nRF24L01_secure s_radio(&my_radio,&the_serial_port,&my_queue);

// Create a sleep object
avrsleep my_sleep(&the_serial_port);

//these variable are to get some result back to the main program from the modules
short peaks[2][10]={0};
short cycle_peaks_freq[FFT_PER_CYCLE][2]={0};
short cycle_peaks_mag[FFT_PER_CYCLE][2][3]={0};

//The reference w for 100% healthy blade
short w_ref=0;
//The standard deviation
unsigned short std=0;

//enable global interrupt
sei ();

//Sent radio to let user know the current status
s_radio.quick_packet_send((int)cbm_status, FFT_PER_CYCLE, NUM_ERR_REF, MIN_ERR_REF,
SAMPLE_FREQUENCY,NUM_SAMPLES,NOISE_THRESHOLD,0);

//turn off radio. Don't want to waste powers
my_radio.power_down();
cbm_status=CBM_GET_W_REF;
do{
    //reset the counter
    char cycle_counter=0;
    short w[NUM_INIT_REF]={0};    //clear up a set of numbers for frequency filtering

    //To gather
    while(cycle_counter<NUM_INIT_REF)
    {
        my_sensor.read_accx();
    }
}

```

```

//done with gathering data
if (my_sensor.sensor_data_state()==2)
{
    //perform IFFT
    my_fft.go(data2);
    //Get the top two frequencies
    my_fft.topfreq (data2,peaks);
    //Store data, -1 in get set sum so that it starts with zero
    my_fft.store_fft_cycle (data2, peaks, my_fft.get_set_num()-1,
    cycle_peaks_freq, cycle_peaks_mag);

    //if collected enough data, then proceed to calculate average
    if (my_fft.get_set_num())>=FFT_PER_CYCLE)
    {
        //calculate average for the cycle
        w[cycle_counter]=my_fft.get_avg_freq (cycle_peaks_freq, cycle_peaks_mag,
        &std);

        //reset the cycle
        my_fft.cycle_reset();
        cycle_counter++;
    }

    //deep sleep code here
    do{
        my_sleep.sleep_power_save();
    } while (my_sleep.get_counter())<=FFT_INTERVAL8X);
    //reset values after wake up to avoid possible problems
    my_sleep.reset_sleep_counter();
    my_sensor.reset_data_state ();

}
//once it goes to sleep, adc will start the process to get the data one by one
my_sleep.sleep_idle();
}
//After the program gathered enough cycle averages

//wake up the radio module
my_radio.power_up();
for (char i=0;i<NUM_INIT_REF;i++)
{
    GLOB_DEBUG(PMS(" ")<<w[i]<<PMS(" "));
}

//create an object so that the user will know the next radio send out time
short next_radio_time=0;

//Try getting a filtered number as reference w, radio time is calculated
if(get_filtered_w(w, &w_ref, NUM_INIT_REF,MIN_NUM_REF, W_REF_MAX_TOL,
W_REF_MIN_TOL))
{
    cbm_status=CBM_REG_OP;
}

```

```
    next_radio_time= FFT_INTERVAL8X*FFT_PER_CYCLE*NUM_ERR_REF*8;
}
else //if not successful to get a filtered reference w
{
    next_radio_time= FFT_INTERVAL8X*FFT_PER_CYCLE*NUM_INIT_REF*8;
}

//send out a radio signal to let user know the current status in get reference w
s_radio.quick_packet_send(CBM_GET_W_REF, (int)w_ref, (int)next_radio_time,
NUM_INIT_REF, MIN_NUM_REF,W_REF_TOL,0,0);
my_radio.power_down();

}while(cbm_status!=CBM_REG_OP);

//Getting into regular operation once reference w is found

//A number to keep track of number of cycles the program ran
unsigned short running_cycle_number=0;
//The actual number to see the frequency of oscillation of blade
short w_err_avg=0;
char w_err_avg_counter=0;
short w_err[NUM_ERR_REF]={0};

while(1)
{
    //Enabling reading the x-axis of accelerometer
    my_sensor.read_accx();

    //check to see whether it has finished gathering data
    if (my_sensor.sensor_data_state()==2)
    {
        //my_sensor.send_serial();
        my_fft.go(data2);

        //getting the top two frequencies
        my_fft.topfreq (data2,peaks);

        //Store data, -1 in get set sum so that it starts with zero
        my_fft.store_fft_cycle (data2, peaks, my_fft.get_set_num()-1, cycle_peaks_freq,
        cycle_peaks_mag);

        //A status printing serial debug
        GLOB_DEBUG(my_fft.get_set_num());

        //To see whether it has finished gathering data, if so get the cycle average
        if (my_fft.get_set_num())>=FFT_PER_CYCLE)
        {
            //Get an average from the cycle averages
            short w_avg=my_fft.get_avg_freq (cycle_peaks_freq, cycle_peaks_mag, &std);

            w_err[w_err_avg_counter]=w_avg;
            w_err_avg_counter++;
        }
    }
}
```

```

GLOB_DEBUG (PMS ("\n\rAVG counter ")<<w_err_avg_counter<<PMS (" \n\r"));

//if all the cycle averages are collected, it will calculate the set average
if(w_err_avg_counter==NUM_ERR_REF)
{
    //radio power up
    my_radio.power_up();
    //the following code needs to run for at least 1.5ms because the radio
    //needs at least 1.5ms to warm up
    GLOB_DEBUG (PMS ("\n\rBegin Inspection\n\r"));

    //Check to see whether it has a set averages
    if(get_filtered_w(w_err, &w_err_avg, NUM_ERR_REF,MIN_ERR_REF,
ERR_MAX_TOL, ERR_MIN_TOL))
    {
        GLOB_DEBUG (PMS ("\n\rThe status is...\n\r"));
        if(w_err_avg> (short)((long)w_ref*FAILURE_PERCENT/1000))
        {
            GLOB_DEBUG (PMS ("\n\rGOOD!!!!\n\r"));
            cbm_status=CBM_REG_OP;
        }
        else if (w_err_avg==0)
        {
            GLOB_DEBUG (PMS ("\n\rBlade Broke off\n\r"));
            cbm_status=CBM_ZERO_W;
        }
        else
        {
            GLOB_DEBUG (PMS ("\n\rAbout to fail\n\r"));
            cbm_status=CBM_ERROR;
        }
    }
    else
    {
        GLOB_DEBUG (PMS ("\n\rI can't find a average frequency"));
        cbm_status=CBM_NEED_ATTENTION;
    }
    w_err_avg_counter=0;

    // Send a radio to tell the user the result everytime
    s_radio.quick_packet_send((int)cbm_status, (int)w_err_avg,
FFT_INTERVAL8X*FFT_PER_CYCLE*NUM_ERR_REF*8, (int)w_ref, FAILURE_PERCENT,
ERR_TOL, (int)running_cycle_number,0);

    //power down the radio
    my_radio.power_down();
    //transmit now and clear set_num data, and the peaks data
    GLOB_DEBUG (PMS ("\n\rRadio SENT\n\r"));
    running_cycle_number++;
    GLOB_DEBUG(PMS("\n\rrunning cycle num: ")<<running_cycle_number<<PMS("
is done"));

```

```

    }

    GLOB_DEBUG (PMS ("The average is ")<<w_avg<<HZ_STRING<<PMS ("\n\r"));

    //resets the cycle
    my_fft.cycle_reset();

}

//Sleep in power save mode (wakes up every 8 second), for a total of 30mins if
//the FFT_INTERVAL8X in the header file is set to 225
do{
    my_sleep.sleep_power_save();
} while (my_sleep.get_counter()<=FFT_INTERVAL8X);

//reset all the sleeping stuff
my_sleep.reset_sleep_counter();
my_sensor.reset_data_state ();

}
//once it goes to sleep, adc will start the process to get the data one by one
my_sleep.sleep_idle();
}

return (0);
}

// A function to get an average and make sure the numbers used are within tolerance
bool get_filtered_w (short w[],short* p_w_avg, char max_num, char min_num, short max_tol,
short min_tol)
{
    char error_counter=0;
    char positive_counter;
    char freq_index_del;
    short w_avg=0;
    char max_count=max_num;
    bool done_flag=false;

    //if all of the values are in tolerance or not enough numbers is within tolerance,
    //then the process is done
    while ((done_flag!=true)&&(error_counter<=(max_num-min_num)))
    {

        char freq_indexh=0;
        char freq_indexl=0;
        long sum=0;
        positive_counter=0;

```

```

//Find an extreme index
for (char i=0;i<max_count;i++)
{
    //Find a sum
    sum+=w[i];

    //To compare all the number to see which one is highest and which is lowest
    if (w[i]>w[freq_indexh])
    {
        freq_indexh=i;
    }
    else if (w[i]<w[freq_indexl])
    {
        freq_indexl=i;
    }
}
GLOB_DEBUG(PMS(" sum: ")<<sum<<PMS(" "));
GLOB_DEBUG(PMS(" max_count: ")<<max_count<<PMS(" "));

//Find an average
w_avg=(short)(sum/max_count);
GLOB_DEBUG(PMS(" Average W: ")<<w_avg<<PMS(" "));

//check to see whether the highest number or lower number is further from the
//average and set that as a index for possiblily to delete
if ((w[freq_indexh]-w_avg)>=(w_avg-w[freq_indexl]))
{
    freq_index_del=freq_indexh;
}
else
{
    freq_index_del=freq_indexl;
}

GLOB_DEBUG(PMS(" Index_high: ")<<freq_indexh<<PMS(" "));
GLOB_DEBUG(PMS(" Index_low: ")<<freq_indexl<<PMS(" "));

//check to see whether a number is within tolerance, if not delete the index
for (char i=0;i<max_count;i++)
{
    //added the >= <= with the equal sign, so that 0 can be included.
    //as long as they remain within 2.3% of the range, then we would consider
    that a good data
    if((w[i]<=(short)(((long)w_avg*max_tol)/1000))&&(w[i]>=(short)(((long)
    w_avg*min_tol)/1000)))
    {
        positive_counter++;
    }
    else
    {
        error_counter++;
        //shift the last data out of the end
    }
}

```



```
        max_count--;
        w[freq_index_del]=w[max_count];
        break;
    }
}

//if there are enough data to qualify as a successful filtering run, then set
//the flag as done
if (positive_counter>=min_num)
{
    done_flag=true;
    *p_w_avg=w_avg;
}

GLOB_DEBUG(PMS(" Positive Counter: ")<<positive_counter<<PMS(" "));
GLOB_DEBUG(PMS(" Error Counter: ")<<error_counter<<PMS(" "));
}
return done_flag;
}

/* Test run on Oct 25th, 2010 8:30pm with v0_6_1
16mhz crystal atmega128
256 data point fft=          26.896ms
    top_freq                2.343ms
    store_data              0.019ms
    get_3peaks              0.272msec
    get_avg_freq            0.408msec
sent_8 set data via radio    41.595msec
*/
```

```
//=====

/** \file CBM_v1_0.h
 *   This is a header file for setting up variable that is being included in multiple files
 *
 *   Revisions:
 *   \li 10-01-10 CKC initialized
 *   \li 06-22-11 CKC Revised and fully commented code
 *
 *   License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 */
//=====

/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _CBM_V1_0_H_
#define _CBM_V1_0_H_

//*****
//*****
//***** General Parameters to change *****
//*****
//*****

//NUN_SAMPLES has to be in powers of 2, and from 4 to 32678(?), but subject to ram
//limitation, num samples can at max be about 1/8 of the max ram for 2k or less ram.
//1/4 of the max ram if it is 4k or more ram
#define NUM_SAMPLES 256
//A good guess to the natural frequency in Hz, it is tested to work up to 500Hz
#define MAX_DETECTION_FREQUENCY 30 //in Hertz
#define BRAD_RATE 9600 //Transmission rate at bits/sec
//either 1 or 0, if 1 is not supported, 0 is automatically selected
#define SELECT_SERIAL_PORT 1
#define QUEUE_SIZE 32 //queue is used by radio buffer in bytes
//Samples< FFT < CYCLE< SET
#define FFT_PER_CYCLE 9

//***** General preprocessor equation to *****
//***** calculate value (please don't modified!) *****

//Sampling frequency, a value of 30 means it samples 30 times a second
#define SAMPLE_FREQUENCY (MAX_DETECTION_FREQUENCY*2)
#ifdef UCSR1A
    #if SELECT_SERIAL_PORT <=1
        #define SERIAL_PORT SELECT_SERIAL_PORT
    #else
        #error "This Serial Port number is not supported in this MCU"
    #endif
#else
```

```

#define SERIAL_PORT          0
#endif

//*****
//*****
//***** CBM_v1_0.cpp Parameters to change *****
//*****
//*****

//for radio set up pin. CE= Chip Enable
#ifdef __AVR_ATmega128__
#define CE_MASK              0x10
#define CSN_MASK             0x20

//Atmega324p, should also be good for Atmega644
//please add other definition if another chip is used
#else
#define CE_MASK              0x08
#define CSN_MASK             0x10
#endif

//Number of data points during the initialization to get w_ref for a reference w for failure
#define NUM_INIT_REF         8
//The number of data point to be within the range of tolerance (set by w_ref_max_tol and
//w_ref_min_tol) to be acceptable
#define MIN_NUM_REF          5

//Number of data points for comparision. When the set of data points to have a trend to drop
//below the failure criteria, then failure is called
#define NUM_ERR_REF          6
//The number of data point to be within the range of tolerance set by ERR_MAX_TOL and
//ERR_MIN_TOL
#define MIN_ERR_REF          4

//The tolerance accepted to determine initial w reference(in 0.1% increment, so 25 means
//2.5% difference)
#define W_REF_TOL             25

//The tolerance accepted to determine the failure of the blade(in 0.1% increment, so 25
//means 2.5% difference)
#define ERR_TOL               25

//The failure percentage, so 920 means a drop in 8% of the damped natural frequency to
//classified as failure
#define FAILURE_PERCENT       920

//The time between individual fft in x8sec.For example: vale of
// 3    means 24sec
// 15    means 2mins
// 150   means 20mins
// 450   means 1 hrs

```

```

#define FFT_INTERVAL8X          225

//***** CBM_v1_0.cpp preprocesse equation to *****
//***** calculate value (please don't modified!) *****

//The tolerance accepted to determine initial w reference (in 0.1% increment, so 1025 means
//102.5% or 2.5% above normal and 975 means 97.5% or 2.5% below normal)
#define W_REF_MAX_TOL          1000+W_REF_TOL
#define W_REF_MIN_TOL          1000-W_REF_TOL

//The tolerance accepted to determine the failure of the blade(in 0.1% increment, so 1025
//means 102.5% or 2.5% above normal and 975 means 97.5% or 2.5% below normal)
#define ERR_MAX_TOL            1000+ERR_TOL
#define ERR_MIN_TOL            1000-ERR_TOL

//Total of time it takes to finish all the sampling for one single FFT
#define SAMPLE_CYCLE_TIME      NUM_SAMPLES*1000L/SAMPLE_FREQUENCY

//*****
//*****
//***** sensor.cpp Parameters to change *****
//*****
//*****

//***** sensor.cpp preprocesse equation to *****
//***** calculate value (please don't modified!) *****

//The amount of clock cycles between each sampling. 256 is the clock divider
#define CLK_PER_SAMPLE         F_CPU/SAMPLE_FREQUENCY/256

//*****
//*****
//***** fft_int.cc Parameters to change *****
//*****
//*****

//Normally about 150 is usually what the max value can get
#define NOISE_THRESHOLD         40

//***** fft_int.cc preprocesse equation to *****
//***** calculate value (please don't modified!) *****

//Some macro to make sure the mcu display correct items
#if APPROXIMATE_NATURAL_FREQUENCY<= 15
    #define BIN_WIDTH_FACTOR 1000L
    #define HZ_STRING PMS("x10^(-3) Hz")
#elif APPROXIMATE_NATURAL_FREQUENCY<= 150
    #define BIN_WIDTH_FACTOR 10L

```

```
#define HZ_STRING PMS("x10(-1) Hz")
#else
#define BIN_WIDTH_FACTOR 1L
#define HZ_STRING PMS("Hz")
#endif

//i have to make the BIN_WIDTH_MHZ to be a long number
//individual bin resolution is Sample_Frequency/total_number_sample
#define BIN_WIDTH_HZ SAMPLE_FREQUENCY*BIN_WIDTH_FACTOR/NUM_SAMPLES
#define SAMPLES_OVER128 (NUM_SAMPLES/128) //A Constant used to fix bin width

//*****
//*****
//***** avrsleep.cc Parameters to change *****
//*****
//*****

//***** avrsleep.cc preprocesse equation to *****
//***** calculate value (please don't modified!) *****

#endif // _CBM_V1_0_H_
```

```
//=====
/** \file  fft_int.h
 *   This function provide a way to do a integer fast fourier transform
 *
 * Revisions:
 *   \li 04-16-10 CKC Reused the code from fft5_0_4.cc
 *   \li 06-22-11 CKC Revised and fully commented code
 *
 * License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 */
//=====

/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _FFT_INT_H_
#define _FFT_INT_H_

#include "CBM_v1_0.h"

class fft_int
{
protected:
    unsigned char peak_set_num;

public:
    //Enable fft to communicate or print stuff
    fft_int(base_text_serial*);
    //destructor
    ~fft_int(void);

    //This runs the Integer FFT
    void go (signed short data[]);
    //Get sine table
    signed short get_sintab (short theta_int);
    //Find Top 2 frequencies
    void topfreq (signed short data2[],short peaks[][10]);
    //Store frequencies for cycle averaging
    void store_fft_cycle (signed short data[], short peaks[][10], unsigned char set_num,
short cycle_peaks_freq[FFT_PER_CYCLE][2], short cycle_peaks_mag[FFT_PER_CYCLE][2][3]);
    //Cycle averaging
    short get_avg_freq (short cycle_peaks_freq[FFT_PER_CYCLE][2], short cycle_peaks_mag
[FFT_PER_CYCLE][2][3], unsigned short*);
    //Refining the frequency by using adjacent frequency points
    void get_wo_3p(short data[], short peaks[][10], short peaks_adj[], long w_o_mHz[]);
    //Get the current number of FFT
    unsigned char get_set_num(void){return (peak_set_num);};
    //Resetting the cycle, clearing the number for FFT
    void cycle_reset(void);
    //Integer form of Squire Root
    unsigned short isqrt(unsigned long);

```

};

#endif

```

//=====
/** \file  fft_int.c
 *    This function provide a way to do Integer Fast Fourier Transform
 *
 * Revisions:
 * \li 04-16-10 CKC Reused the code from fft5_0_4.cc
 * \li 06-22-11 CKC Revised and fully commented code
 *
 * License:
 * This file released under the Lesser GNU Public License. The program is intended
 * for educational use only, but its use is not restricted thereto.
 */
//=====

#include <stdlib.h> // Include standard library header files
#include <avr/io.h> // need to use it with pgmspace
#include <avr/pgmspace.h> // Use the program memory for variables

#include "lib/global_debug.h" // Include header for serial port class
#include "lib/rs232int.h" // Include header for serial port class
#include "CBM_v1_0.h"

//*****All the following are all included inside CBM_v1_0.h, if this were to used
//seperately, please change the following value

#define NOISE_THRESHOLD 50
//To find natural frequency of
#define BIN_WIDTH_HZ 117
#define NUM_SAMPLES 256
#define SAMPLES_OVER128 (NUM_SAMPLES/128)

#include "fft_int.h" // Include the header

//A macro to swap two values
#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp
//Due to the integer reason, i have to use 2048 as 0.5pi
#define numPi0_5 2048
//Due to the integer reason, i have to use 4096 as pi instead of 3.1415
#define pi_int (2*numPi0_5)
//2pi
#define numPi2_0 (4*numPi0_5)
//The maximum value for sine
#define maxsinval 32768
//Since a signed version of positive 32768 means -32768, some some values uses positive
//32767 instead of 32768
#define maxsinval_1 (maxsinval-1)
//A Coefficient of Sine, not really used in this case
#define sinal 1

//Sine Table, it is stored in program memory
short sintab[] PROGMEM= {0, 25, 50, 75, 101, 126, 151, 176, 201, 226, 251, 276, 302, 327,

```


352, 377, 402, 427, 452, 478, 503, 528, 553, 578, 603, 628, 653, 679, 704, 729, 754, 779, 804, 829, 854, 880, 905, 930, 955, 980, 1005, 1030, 1055, 1081, 1106, 1131, 1156, 1181, 1206, 1231, 1256, 1281, 1307, 1332, 1357, 1382, 1407, 1432, 1457, 1482, 1507, 1533, 1558, 1583, 1608, 1633, 1658, 1683, 1708, 1733, 1758, 1784, 1809, 1834, 1859, 1884, 1909, 1934, 1959, 1984, 2009, 2034, 2060, 2085, 2110, 2135, 2160, 2185, 2210, 2235, 2260, 2285, 2310, 2335, 2360, 2385, 2411, 2436, 2461, 2486, 2511, 2536, 2561, 2586, 2611, 2636, 2661, 2686, 2711, 2736, 2761, 2786, 2811, 2836, 2861, 2887, 2912, 2937, 2962, 2987, 3012, 3037, 3062, 3087, 3112, 3137, 3162, 3187, 3212, 3237, 3262, 3287, 3312, 3337, 3362, 3387, 3412, 3437, 3462, 3487, 3512, 3537, 3562, 3587, 3612, 3637, 3662, 3687, 3712, 3737, 3762, 3787, 3812, 3836, 3861, 3886, 3911, 3936, 3961, 3986, 4011, 4036, 4061, 4086, 4111, 4136, 4161, 4186, 4211, 4236, 4260, 4285, 4310, 4335, 4360, 4385, 4410, 4435, 4460, 4485, 4510, 4534, 4559, 4584, 4609, 4634, 4659, 4684, 4709, 4733, 4758, 4783, 4808, 4833, 4858, 4883, 4907, 4932, 4957, 4982, 5007, 5032, 5057, 5081, 5106, 5131, 5156, 5181, 5205, 5230, 5255, 5280, 5305, 5329, 5354, 5379, 5404, 5429, 5453, 5478, 5503, 5528, 5553, 5577, 5602, 5627, 5652, 5676, 5701, 5726, 5751, 5775, 5800, 5825, 5850, 5874, 5899, 5924, 5948, 5973, 5998, 6023, 6047, 6072, 6097, 6121, 6146, 6171, 6195, 6220, 6245, 6269, 6294, 6319, 6343, 6368, 6393, 6417, 6442, 6467, 6491, 6516, 6541, 6565, 6590, 6614, 6639, 6664, 6688, 6713, 6737, 6762, 6787, 6811, 6836, 6860, 6885, 6910, 6934, 6959, 6983, 7008, 7032, 7057, 7081, 7106, 7130, 7155, 7180, 7204, 7229, 7253, 7278, 7302, 7327, 7351, 7376, 7400, 7425, 7449, 7473, 7498, 7522, 7547, 7571, 7596, 7620, 7645, 7669, 7694, 7718, 7742, 7767, 7791, 7816, 7840, 7864, 7889, 7913, 7938, 7962, 7986, 8011, 8035, 8059, 8084, 8108, 8133, 8157, 8181, 8206, 8230, 8254, 8279, 8303, 8327, 8351, 8376, 8400, 8424, 8449, 8473, 8497, 8521, 8546, 8570, 8594, 8618, 8643, 8667, 8691, 8715, 8740, 8764, 8788, 8812, 8836, 8861, 8885, 8909, 8933, 8957, 8982, 9006, 9030, 9054, 9078, 9102, 9127, 9151, 9175, 9199, 9223, 9247, 9271, 9295, 9319, 9344, 9368, 9392, 9416, 9440, 9464, 9488, 9512, 9536, 9560, 9584, 9608, 9632, 9656, 9680, 9704, 9728, 9752, 9776, 9800, 9824, 9848, 9872, 9896, 9920, 9944, 9968, 9992, 10016, 10040, 10064, 10088, 10112, 10135, 10159, 10183, 10207, 10231, 10255, 10279, 10303, 10326, 10350, 10374, 10398, 10422, 10446, 10469, 10493, 10517, 10541, 10565, 10588, 10612, 10636, 10660, 10684, 10707, 10731, 10755, 10779, 10802, 10826, 10850, 10873, 10897, 10921, 10945, 10968, 10992, 11016, 11039, 11063, 11087, 11110, 11134, 11157, 11181, 11205, 11228, 11252, 11276, 11299, 11323, 11346, 11370, 11393, 11417, 11441, 11464, 11488, 11511, 11535, 11558, 11582, 11605, 11629, 11652, 11676, 11699, 11723, 11746, 11770, 11793, 11816, 11840, 11863, 11887, 11910, 11934, 11957, 11980, 12004, 12027, 12051, 12074, 12097, 12121, 12144, 12167, 12191, 12214, 12237, 12261, 12284, 12307, 12330, 12354, 12377, 12400, 12424, 12447, 12470, 12493, 12517, 12540, 12563, 12586, 12609, 12633, 12656, 12679, 12702, 12725, 12748, 12772, 12795, 12818, 12841, 12864, 12887, 12910, 12933, 12957, 12980, 13003, 13026, 13049, 13072, 13095, 13118, 13141, 13164, 13187, 13210, 13233, 13256, 13279, 13302, 13325, 13348, 13371, 13394, 13417, 13440, 13463, 13485, 13508, 13531, 13554, 13577, 13600, 13623, 13646, 13668, 13691, 13714, 13737, 13760, 13783, 13805, 13828, 13851, 13874, 13896, 13919, 13942, 13965, 13987, 14010, 14033, 14056, 14078, 14101, 14124, 14146, 14169, 14192, 14214, 14237, 14260, 14282, 14305, 14327, 14350, 14373, 14395, 14418, 14440, 14463, 14485, 14508, 14530, 14553, 14576, 14598, 14621, 14643, 14665, 14688, 14710, 14733, 14755, 14778, 14800, 14823, 14845, 14867, 14890, 14912, 14935, 14957, 14979, 15002, 15024, 15046, 15069, 15091, 15113, 15136, 15158, 15180, 15202, 15225, 15247, 15269, 15291, 15314, 15336, 15358, 15380, 15402, 15425, 15447, 15469, 15491, 15513, 15535, 15557, 15580, 15602, 15624, 15646, 15668, 15690, 15712, 15734, 15756, 15778, 15800, 15822, 15844, 15866, 15888, 15910, 15932, 15954, 15976, 15998, 16020, 16042, 16064, 16086, 16108, 16129, 16151, 16173, 16195, 16217, 16239, 16261, 16282, 16304, 16326, 16348, 16369, 16391, 16413, 16435, 16456, 16478, 16500, 16522, 16543, 16565, 16587, 16608, 16630, 16652, 16673, 16695, 16717, 16738, 16760, 16781, 16803, 16825, 16846, 16868, 16889, 16911, 16932, 16954, 16975, 16997, 17018, 17040, 17061, 17083, 17104, 17126, 17147, 17168, 17190, 17211, 17233, 17254, 17275, 17297, 17318, 17339, 17361, 17382, 17403, 17425, 17446, 17467, 17488,

17510, 17531, 17552, 17573, 17594, 17616, 17637, 17658, 17679, 17700, 17721, 17743, 17764, [↗](#)
17785, 17806, 17827, 17848, 17869, 17890, 17911, 17932, 17953, 17974, 17995, 18016, 18037, [↗](#)
18058, 18079, 18100, 18121, 18142, 18163, 18184, 18205, 18226, 18247, 18268, 18288, 18309, [↗](#)
18330, 18351, 18372, 18393, 18413, 18434, 18455, 18476, 18496, 18517, 18538, 18559, 18579, [↗](#)
18600, 18621, 18641, 18662, 18683, 18703, 18724, 18745, 18765, 18786, 18806, 18827, 18848, [↗](#)
18868, 18889, 18909, 18930, 18950, 18971, 18991, 19012, 19032, 19053, 19073, 19093, 19114, [↗](#)
19134, 19155, 19175, 19195, 19216, 19236, 19256, 19277, 19297, 19317, 19338, 19358, 19378, [↗](#)
19399, 19419, 19439, 19459, 19479, 19500, 19520, 19540, 19560, 19580, 19601, 19621, 19641, [↗](#)
19661, 19681, 19701, 19721, 19741, 19761, 19781, 19801, 19821, 19841, 19861, 19881, 19901, [↗](#)
19921, 19941, 19961, 19981, 20001, 20021, 20041, 20061, 20081, 20100, 20120, 20140, 20160, [↗](#)
20180, 20200, 20219, 20239, 20259, 20279, 20298, 20318, 20338, 20357, 20377, 20397, 20416, [↗](#)
20436, 20456, 20475, 20495,
20515, 20534, 20554, 20573, 20593, 20612, 20632, 20652, 20671, 20691, 20710, 20729, 20749, [↗](#)
20768, 20788, 20807, 20827, 20846, 20865, 20885, 20904, 20923, 20943, 20962, 20981, 21001, [↗](#)
21020, 21039, 21059, 21078, 21097, 21116, 21136, 21155, 21174, 21193, 21212, 21231, 21251, [↗](#)
21270, 21289, 21308, 21327, 21346, 21365, 21384, 21403, 21422, 21441, 21460, 21479, 21498, [↗](#)
21517, 21536, 21555, 21574, 21593, 21612, 21631, 21649, 21668, 21687, 21706, 21725, 21744, [↗](#)
21762, 21781, 21800, 21819, 21838, 21856, 21875, 21894, 21912, 21931, 21950, 21968, 21987, [↗](#)
22006, 22024, 22043, 22061, 22080, 22099, 22117, 22136, 22154, 22173, 22191, 22210, 22228, [↗](#)
22247, 22265, 22284, 22302, 22320, 22339, 22357, 22375, 22394, 22412, 22431, 22449, 22467, [↗](#)
22485, 22504, 22522, 22540, 22558, 22577, 22595, 22613, 22631, 22649, 22668, 22686, 22704, [↗](#)
22722, 22740, 22758, 22776, 22794, 22812, 22830, 22848, 22866, 22884, 22902, 22920, 22938, [↗](#)
22956, 22974, 22992, 23010, 23028, 23046, 23064, 23081, 23099, 23117, 23135, 23153, 23170, [↗](#)
23188, 23206, 23224, 23241, 23259, 23277, 23295, 23312, 23330, 23348, 23365, 23383, 23400, [↗](#)
23418, 23436, 23453, 23471, 23488, 23506, 23523, 23541, 23558, 23576, 23593, 23610, 23628, [↗](#)
23645, 23663, 23680, 23697, 23715, 23732, 23749, 23767, 23784, 23801, 23819, 23836, 23853, [↗](#)
23870, 23888, 23905, 23922, 23939, 23956, 23973, 23991, 24008, 24025, 24042, 24059, 24076, [↗](#)
24093, 24110, 24127, 24144, 24161, 24178, 24195, 24212, 24229, 24246, 24263, 24279, 24296, [↗](#)
24313, 24330, 24347, 24364, 24380, 24397, 24414, 24431, 24448, 24464, 24481, 24498, 24514, [↗](#)
24531, 24548, 24564, 24581, 24598, 24614, 24631, 24647, 24664, 24680, 24697, 24713, 24730, [↗](#)
24746, 24763, 24779, 24796, 24812, 24829, 24845, 24861, 24878, 24894, 24910, 24927, 24943, [↗](#)
24959, 24976, 24992, 25008, 25024, 25041, 25057, 25073, 25089, 25105, 25121, 25138, 25154, [↗](#)
25170, 25186, 25202, 25218, 25234, 25250, 25266, 25282, 25298, 25314, 25330, 25346, 25362, [↗](#)
25378, 25394, 25410, 25425, 25441, 25457, 25473, 25489, 25504, 25520, 25536, 25552, 25567, [↗](#)
25583, 25599, 25615, 25630, 25646, 25662,
25677, 25693, 25708, 25724, 25739, 25755, 25771, 25786, 25802, 25817, 25833, 25848, 25863, [↗](#)
25879, 25894, 25910, 25925, 25940, 25956, 25971, 25986, 26002, 26017, 26032, 26048, 26063, [↗](#)
26078, 26093, 26108, 26124, 26139, 26154, 26169, 26184, 26199, 26214, 26229, 26244, 26259, [↗](#)
26275, 26290, 26305, 26320, 26334, 26349, 26364, 26379, 26394, 26409, 26424, 26439, 26454, [↗](#)
26468, 26483, 26498, 26513, 26528, 26542, 26557, 26572, 26586, 26601, 26616, 26630, 26645, [↗](#)
26660, 26674, 26689, 26704, 26718, 26733, 26747, 26762, 26776, 26791, 26805, 26820, 26834, [↗](#)
26848, 26863, 26877, 26892, 26906, 26920, 26935, 26949, 26963, 26977, 26992, 27006, 27020, [↗](#)
27034, 27049, 27063, 27077, 27091, 27105, 27119, 27133, 27147, 27162, 27176, 27190, 27204, [↗](#)
27218, 27232, 27246, 27260, 27273, 27287, 27301, 27315, 27329, 27343, 27357, 27371, 27384, [↗](#)
27398, 27412, 27426, 27440, 27453, 27467, 27481, 27494, 27508, 27522, 27535, 27549, 27562, [↗](#)
27576, 27590, 27603, 27617, 27630, 27644, 27657, 27671, 27684, 27698, 27711, 27724, 27738, [↗](#)
27751, 27765, 27778, 27791, 27805, 27818, 27831, 27844, 27858, 27871, 27884, 27897, 27910, [↗](#)
27924, 27937, 27950, 27963, 27976, 27989, 28002, 28015, 28028, 28041, 28054, 28067, 28080, [↗](#)
28093, 28106, 28119, 28132, 28145, 28158, 28170, 28183, 28196, 28209, 28222, 28234, 28247, [↗](#)
28260, 28273, 28285, 28298, 28311, 28323, 28336, 28349, 28361, 28374, 28386, 28399, 28411, [↗](#)
28424, 28436, 28449, 28461, 28474, 28486, 28499, 28511, 28523, 28536, 28548, 28560, 28573, [↗](#)
28585, 28597, 28610, 28622, 28634, 28646, 28658, 28671, 28683, 28695, 28707, 28719, 28731, [↗](#)

```
28743, 28755, 28767, 28779, 28791, 28803, 28815, 28827, 28839, 28851, 28863, 28875, 28887,
28899, 28911, 28922, 28934, 28946, 28958, 28970, 28981, 28993, 29005, 29016, 29028, 29040,
29051, 29063, 29075, 29086, 29098, 29109, 29121, 29132,
29144, 29155, 29167, 29178, 29190, 29201, 29212, 29224, 29235, 29247, 29258, 29269, 29280,
29292, 29303, 29314, 29325, 29337, 29348, 29359, 29370, 29381, 29392, 29404, 29415, 29426,
29437, 29448, 29459, 29470, 29481, 29492, 29503, 29514, 29525, 29535, 29546, 29557, 29568,
29579, 29590, 29600, 29611, 29622, 29633, 29643, 29654, 29665, 29675, 29686, 29697, 29707,
29718, 29729, 29739, 29750, 29760, 29771, 29781, 29792, 29802, 29813, 29823, 29833, 29844,
29854, 29864, 29875, 29885, 29895, 29906, 29916, 29926, 29936, 29947, 29957, 29967, 29977,
29987, 29997, 30008, 30018, 30028, 30038, 30048, 30058, 30068, 30078, 30088, 30098, 30108,
30118, 30127, 30137, 30147, 30157, 30167, 30177, 30186, 30196, 30206, 30216, 30225, 30235,
30245, 30254, 30264, 30274, 30283, 30293, 30302, 30312, 30322, 30331, 30341, 30350, 30360,
30369, 30378, 30388, 30397, 30407, 30416, 30425, 30435, 30444, 30453, 30462, 30472, 30481,
30490, 30499, 30509, 30518, 30527, 30536, 30545, 30554, 30563, 30572, 30581, 30590, 30599,
30608, 30617, 30626, 30635, 30644, 30653, 30662, 30671, 30680, 30688, 30697, 30706, 30715,
30723, 30732, 30741, 30750, 30758, 30767, 30776, 30784, 30793, 30801, 30810, 30819, 30827,
30836, 30844, 30853, 30861, 30869, 30878, 30886, 30895, 30903, 30911, 30920, 30928, 30936,
30945, 30953, 30961, 30969, 30977, 30986, 30994, 31002, 31010, 31018, 31026, 31034, 31042,
31050, 31059, 31067, 31074, 31082, 31090, 31098, 31106, 31114, 31122, 31130, 31138, 31146,
31153, 31161, 31169, 31177, 31184, 31192, 31200, 31207, 31215, 31223, 31230, 31238, 31246,
31253, 31261, 31268, 31276, 31283, 31291, 31298, 31305, 31313, 31320, 31328, 31335, 31342,
31350, 31357, 31364, 31372,
31379, 31386, 31393, 31400, 31408, 31415, 31422, 31429, 31436, 31443, 31450, 31457, 31464,
31471, 31478, 31485, 31492, 31499, 31506, 31513, 31520, 31527, 31534, 31540, 31547, 31554,
31561, 31568, 31574, 31581, 31588, 31594, 31601, 31608, 31614, 31621, 31627, 31634, 31641,
31647, 31654, 31660, 31667, 31673, 31679, 31686, 31692, 31699, 31705, 31711, 31718, 31724,
31730, 31737, 31743, 31749, 31755, 31761, 31768, 31774, 31780, 31786, 31792, 31798, 31804,
31810, 31816, 31822, 31828, 31834, 31840, 31846, 31852, 31858, 31864, 31870, 31875, 31881,
31887, 31893, 31899, 31904, 31910, 31916, 31921, 31927, 31933, 31938, 31944, 31950, 31955,
31961, 31966, 31972, 31977, 31983, 31988, 31994, 31999, 32005, 32010, 32015, 32021, 32026,
32031, 32037, 32042, 32047, 32052, 32058, 32063, 32068, 32073, 32078, 32083, 32088, 32093,
32099, 32104, 32109, 32114, 32119, 32124, 32129, 32133, 32138, 32143, 32148, 32153, 32158,
32163, 32167, 32172, 32177, 32182, 32186, 32191, 32196, 32201, 32205, 32210, 32214, 32219,
32224, 32228, 32233, 32237, 32242, 32246, 32251, 32255, 32259, 32264, 32268, 32273, 32277,
32281, 32286, 32290, 32294, 32298, 32303, 32307, 32311, 32315, 32319, 32323, 32328, 32332,
32336, 32340, 32344, 32348, 32352, 32356, 32360, 32364, 32368, 32372, 32376, 32379, 32383,
32387, 32391, 32395, 32398, 32402, 32406, 32410, 32413, 32417, 32421, 32424, 32428, 32432,
32435, 32439, 32442, 32446, 32449, 32453, 32456, 32460, 32463, 32467, 32470, 32473, 32477,
32480, 32483, 32487, 32490, 32493, 32496, 32500, 32503, 32506, 32509, 32512, 32515, 32518,
32522, 32525, 32528, 32531, 32534, 32537, 32540, 32543, 32546, 32548, 32551, 32554, 32557,
32560, 32563, 32566, 32568, 32571, 32574, 32577, 32579, 32582, 32585, 32587, 32590,
32592, 32595, 32598, 32600, 32603, 32605, 32608, 32610, 32613, 32615, 32618, 32620, 32622,
32625, 32627, 32629, 32632, 32634, 32636, 32638, 32641, 32643, 32645, 32647, 32649, 32651,
32654, 32656, 32658, 32660, 32662, 32664, 32666, 32668, 32670, 32672, 32674, 32675, 32677,
32679, 32681, 32683, 32685, 32686, 32688, 32690, 32692, 32693, 32695, 32697, 32698, 32700,
32702, 32703, 32705, 32706, 32708, 32709, 32711, 32712, 32714, 32715, 32717, 32718, 32719,
32721, 32722, 32723, 32725, 32726, 32727, 32729, 32730, 32731, 32732, 32733, 32734, 32736,
32737, 32738, 32739, 32740, 32741, 32742, 32743, 32744, 32745, 32746, 32747, 32748, 32748,
32749, 32750, 32751, 32752, 32753, 32753, 32754, 32755, 32756, 32756, 32757, 32758, 32758,
32759, 32759, 32760, 32760, 32761, 32761, 32762, 32762, 32763, 32763, 32764, 32764, 32765,
32765, 32765, 32766, 32766, 32766, 32766, 32767, 32767, 32767, 32767, 32767, 32767,
32767, 32767, 32767, 32767};
```

```

//A constructor in Integer FFT
fft_int::fft_int(base_text_serial* p_serial_port)
{
    //clearing the peak_set_num before first use
    peak_set_num=0;
    GLOB_DEBUG (PMS ("fft setup done \n\r"));
}

//Destructor
fft_int::~fft_int(void)
{
}

//The Integer FFT function
void fft_int::go (signed short data[])
{
    //variables for trigonometric recurrences
    short mmax,m,j,istep,i,count,temp;
    long wtemp,wr,wpr,wpi,wi,theta,tempr,tempi,temprp, tempiip,temprin, tempiin,
    data_max=0, data_min=0;
    wtemp=wr=wpr=wpi=wi=theta=tempr=tempi=temp=temprp=tempiip=temprin=tempiin=0;

    //binary inversion (note that the indexes start from 0 witch means that the real part
    //of the complex is on the even-indexes and the complex part is on the odd-indexes
    j=0;
    for (i=0;i<NUM_SAMPLES/2;i+=2)
    {
        if (j > i) {
            //swap the real part
            SWAP(data[j],data[i]);
            //swap the complex part
            SWAP(data[j+1],data[i+1]);
            // checks if the changes occurs in the first half and use the mirrored effect
            // on the second half
            if((j/2)<(NUM_SAMPLES/4)){
                //swap the real part
                SWAP(data[(NUM_SAMPLES-(i+2))],data[(NUM_SAMPLES-(j+2))]);
                //swap the complex part
                SWAP(data[(NUM_SAMPLES-(i+2))+1],data[(NUM_SAMPLES-(j+2))+1]);
            }
        }

        //The amount of complex pairs
        m=NUM_SAMPLES/2;
        while (m >= 2 && j >= m) {
            j -= m;
            m = m/2;
        }
        j += m;
    }
}

```

```

}

//Danielson-Lanzcos routine
mmax=2;

//To make the number smaller, so it won't be a extremely huge number
for (count=0;count<NUM_SAMPLES;count++)
    data[count]=(data[count])>>1;

//The idea here is that because I need 4-5 sig fig from the calculation, yet i'm
//working with whole integer number, so i need to treat 32768 as 1, but since 1*1
//still equals to 1 but 32768*32768 != 32768, so i have to multiply and divide
//variables to make this 32768 as 1 works.
while (NUM_SAMPLES > mmax)
{
    istep = mmax<< 1;

    //wtemp is a factor 32768
    wtemp=get_sintab(theta/2);
    //wpr=> 2 factor 32768,so need to divide it by maxsinval, which is 32768
    wpr = (-2*wtemp*wtemp)>>15;

    wpi=get_sintab(sinal*(numPi2_0/mmax));

    //They by 0 factor of 32768, so need to multiply maxsinval, which is 32768
    wr=((1UL)<<15); //(long)(1<<15);
    wi=(0<<15);

    //internal loops
    for (m=1;m<mmax;m+=2) {
        for (i= m;i<=NUM_SAMPLES;i+=istep) {
            j=i+mmax;
            tempr=(short)((wr*data[j-1]-wi*data[j])>>15);
            tempi=(short)((wr*data[j]+wi*data[j-1])>>15);
            if (tempr>temprip)
                temprip=tempr;
            if (tempi>tempiip)
                tempiip=tempi;
            if (tempr<temprin)
                temprin=tempr;
            if (tempi<tempiin)
                tempiin=tempi;
            data[j-1]=data[i-1]-tempr;
            data[j]=data[i]-tempi;
            data[i-1] += tempr;
            data[i] += tempi;

            if (data[j-1]>data_max)
                data_max=data[j-1];
            if (data[j]>data_max)
                data_max=data[j];
            if (data[i-1]>data_max)

```

```

        data_max=data[i-1];
        if (data[i]>data_max)
            data_max=data[i];
        if (data[j-1]<data_min)
            data_min=data[j-1];
        if (data[j]<data_min)
            data_min=data[j];
        if (data[i-1]<data_min)
            data_min=data[i-1];
        if (data[i]<data_min)
            data_min=data[i];

    }
    //divide 2 part of the expression by maxsinval
    wr=((wtemp=wr)*wpr)>>15)-((wi*wpi)>>15)+wr;
    wi=((wi*wpr)>>15)+((wtemp*wpi)>>15)+wi;
}
mmax=istep;

}

// Since i didn't zero the data, so there is a zero bias, so i'm going to clear it out
data[0]=0;
GLOB_DEBUG(PMS("\n\r ")<<temprp<<PMS(" ")<<tempiip);
GLOB_DEBUG(PMS(" ")<<temprin<<PMS(" ")<<tempiin);
GLOB_DEBUG(PMS(" ")<<data_max<<PMS(" ")<<data_min<<PMS("\n\r"));
return;
}

//A function to get a value from the sine table
signed short fft_int::get_sintab(short theta_int)
{
    //This starts the process of using a quarter of a sine to simulate the full sine wave
    //To make sure that it is within the positive 0<2pi range
    if (theta_int>=numPi2_0)
    {
        theta_int= theta_int%numPi2_0;
    }
    //To make sure it is in one of the 4 quarter of the wave.
    if (theta_int>6144)
    {
        return (-pgm_read_word(&sintab[8192-theta_int]));
    }
    else if (theta_int==6144)
    {
        return (-maxsinval);
    }
    else if (theta_int>=4096)
    {
        return (-pgm_read_word(&sintab[theta_int-4096]));
    }
    else if (theta_int>2048)

```

```

{
    return (pgm_read_word(&sintab[4096-theta_int]));
}
else if (theta_int==2048)
{
    //In order to save some calculation time, i have to use 16bit signed values,
    //which doesn't have 32768,so i have to use 32767 as max, this may introduce
    //a little more error, but it saves time.
    return maxsinval_1;
}

else if (theta_int>=0)
{
    return pgm_read_word(&sintab[theta_int]);
}

//if the number is negative, make it positive first
else if (theta_int<0)
    theta_int=-theta_int;
//make sure it's in the 0<2pi range
if (theta_int>=numPi2_0)
    theta_int= theta_int%numPi2_0;
//start the sorting process again.
if (theta_int>6144)
    return (pgm_read_word(&sintab[8192-theta_int]));
//can't use max sin value again.
else if (theta_int==6144)
    return (maxsinval_1);
else if (theta_int>=4096)
    return (pgm_read_word(&sintab[theta_int-4096]));
else if (theta_int>2048)
    return (-pgm_read_word(&sintab[4096-theta_int]));
else if (theta_int==2048)
    return -maxsinval;
else if (theta_int>=0)
    return (-pgm_read_word(&sintab[theta_int]));

return 0;
}

//Get the top two peaks from FFT
void fft_int::topfreq (signed short data2[],short peaks[][10] )
{
    //16 bit value
    short fft_peaks[2][10]={0};

    //Copy each value to a new value and make it possitive so we can process it later
    for (int num=0; num<NUM_SAMPLES;num++)
    {
        if (data2[num]>=0)
        {

```

```

    else
    {
        //alter the original set of numbers
        data2[num]=-data2[num];
    }
}

//make sure i zero the number first
for(char i=0; i<10; i++)
{
    peaks[0][i]=0;
    peaks[1][i]=0;
}

//=== Getting the 10 peak ===
int num,max_flag=0;
for(char i=0; i<10; i++)
{
    //by saying NUM_SAMPLES/2, we ignore the higher frequency
    for (num=0;num<(NUM_SAMPLES/2);num++)
    {
        if (data2[num]>fft_peaks[1][i])
        {
            fft_peaks[1][i]=data2[num];
            fft_peaks[0][i]=num;
            //make sure we get the proper item from the list
            max_flag=num;
        }
    }
    data2[max_flag]=0;
}

//restore the 10 items back to the fft data2
for(char i=0; i<10; i++)
{
    data2[fft_peaks[0][i]]=fft_peaks[1][i];
}

//=== Isolate individual peaks ===
//A routine that grows to detect frequency that is close to the peak frequency.
for (char i=0; i<9; i++)
{
    for (char j=i+1; j<10;j++)
    {
        //compare the frequency
        if ((fft_peaks[0][j]<=(fft_peaks[0][i]+1))&&(fft_peaks[0][j]>=(fft_peaks[0][i]-1)))
        {
            //zero the magnitude if the frequency is in the neighborhood.
            fft_peaks[1][j]=0;
            //if multiple frequency appears in the list, means skew distribution
        }
    }
}

```



```

    }
}

//=== Storing the data into the class and filter out low number(noise)===
//since a few of the value is being zero out, so it require a seperate sort routine
char index=0; char stupid;
for (char j=0;j<10;j++)
{
    if (fft_peaks[1][j]>=NOISE_THRESHOLD)
    {
        //frequency
        peaks[0][index]=fft_peaks[0][j];
        //value
        peaks[1][index]=data2[(fft_peaks[0][j])];
        index++;
    }
}

//=== Storing the data into the class and filter out low number(noise)===
for (char i=index;i<10;i++)
{
    peaks[0][index]=0;
    //value
    peaks[1][index]=0;
}

//Update the peak set number to tell the user how many times fft has been run in a given
cycle
peak_set_num++;
return;
}

//Store the fft peaks frequencies and their neighboring frequencies' intensity
void fft_int::store_fft_cycle (short data[], short peaks[][10], unsigned char set_num, short
cycle_peaks_freq[FFT_PER_CYCLE][2], short cycle_peaks_mag[FFT_PER_CYCLE][2][3])
{
    for (char i=0;i<2;i++)
    {
        cycle_peaks_freq[set_num][i]=peaks[0][i];

        if (peaks[0][i]!=0)
        {
            cycle_peaks_mag[set_num][i][0]=data[(peaks[0][i]-1)];
        }
        else
        {
            cycle_peaks_mag[set_num][i][0]=0;
        }
        cycle_peaks_mag[set_num][i][1]=data[(peaks[0][i])];
    }
}

```

```

        cycle_peaks_mag[set_num][i][2]=data[(peaks[0][i]+1)];
    }
}

//Get an average frequency per cycle
short fft_int::get_avg_freq (short cycle_peaks_freq[FFT_PER_CYCLE][2], short cycle_peaks_mag
[FFT_PER_CYCLE][2][3], unsigned short* p_std)
{
short w_bar_final=0;
short w_bar=0;
short index=0;
short adjust=0;
short w_counter=0;
unsigned char trouble_peaks_counter=0;
unsigned short sum_mag1=0;
unsigned short sum_mag2=0;
unsigned long diff_sum=0;

//If samples are 32 or less
#if NUM_SAMPLES<=32
    short bin[NUM_SAMPLES]={0};
    short w_bar2=0;
    for (char i=0;i<FFT_PER_CYCLE;i++)
    {
        for (char j=0;j<2;j++)
        {
            bin[cycle_peaks_freq[i][j]]++;
        }
    }
    for (short i=1;i<NUM_SAMPLES-1;i++)
    {
        if(bin[i]>index)
        {
            index=bin[i];
            //not enough resolution to be adjusting
            w_bar= i;
        }
    }
    for (short i=(w_bar+1);i<(NUM_SAMPLES-1);i++)
    {
        if(bin[i]==index)
        {
            trouble_peaks_counter++;
            w_bar2=i;
        }
    }
    if(trouble_peaks_counter!=0)
    {
        GLOB_DEBUG(PMS("Deciding...")<<w_bar<<PMS(" or ")<<w_bar2<<PMS(" "));
        if (trouble_peaks_counter>=2)
        {
            GLOB_DEBUG(PMS("Too many peaks:")<<trouble_peaks_counter<<PMS("\n\n"));

```

```

        *p_std=0;
        return 0; //this set of data should be disregarded due to too many peaks
    }
    else
    {
        GLOB_DEBUG(PMS("which of 2..."));
        for (char i=0; i<FFT_PER_CYCLE; i++)
        {
            for (char j=0; j<2; j++)
            {
                if(cycle_peaks_freq[i][j]==w_bar)
                {
                    sum_mag1+=cycle_peaks_mag[i][j][1];
                }
                else if (cycle_peaks_freq[i][j]==w_bar2)
                {
                    sum_mag2+=cycle_peaks_mag[i][j][1];
                }
            }
        }
        GLOB_DEBUG(PMS("\nrmag1: ")<<sum_mag1<<PMS(" mag2: ")<<sum_mag2<<PMS("\n
        \r"));
        if(sum_mag2>sum_mag1)
        {
            w_bar=w_bar2;
        }
        else if (sum_mag2==sum_mag1)
        {
            GLOB_DEBUG(PMS("is hard..."));
            if((bin[w_bar2+1]+bin[w_bar2]+bin[w_bar2-1])>(bin[w_bar+1]+bin[w_bar]
            +bin[w_bar-1]))
            {
                w_bar=w_bar2;
            }
            else if ((bin[w_bar2+1]+bin[w_bar2]+bin[w_bar2-1])==(bin[w_bar+1]+bin
            [w_bar]+bin[w_bar-1]))
            {
                GLOB_DEBUG(PMS("Give up!!!"));
                *p_std=0;
                return 0;
            }
        }
    }
}

//since the resolution is so low, including neighboring bins doesn't make sense, so i
assume everything that is close to the frequency is within the bin(not too good of
assumption).
*p_std=0;
w_bar_final= w_bar*BIN_WIDTH_HZ; //slafjsdkfactor

//if samples are 128 or less
#elif NUM_SAMPLES<=128

```

```

short bin[NUM_SAMPLES]={0};
short w_bar_raw=0;
short w_bar_raw2=0;
//the resolution of the bin just matches the resolution of the frequency spectrum
for (char i=0;i<FFT_PER_CYCLE;i++)
{
    for (char j=0;j<2;j++)
    {
        bin[cycle_peaks_freq[i][j]]++;
    }
}
bin[0]=0;
//2 extreme ends can't be the most amount, and this should grab the first item with the highest repetition
for (short i=1;i<(NUM_SAMPLES-1);i++)
{
    if(bin[i]>index)
    {
        index=bin[i];
        w_bar_raw= i;
    }
}

for (short i=(w_bar_raw+1);i<(NUM_SAMPLES-1);i++)
{
    if(bin[i]==index)
    {
        trouble_peaks_counter++;
        w_bar_raw2=i;
    }
}

// GLOB_DEBUG(PMS("\n\rwbarraw: ")<<w_bar_raw<<PMS("\n\r"));
// GLOB_DEBUG(PMS("\n\rwbarraw2: ")<<w_bar_raw2<<PMS("\n\r"));
// GLOB_DEBUG(PMS("\n\rtrouble counter: ")<<trouble_peaks_counter<<PMS("\n\r"));

//there are more than 1 peaks inside the same bin(frequency), this is a routine to decide with peak to pick
if(trouble_peaks_counter!=0)
{
    GLOB_DEBUG(PMS("Deciding...")<<w_bar_raw<<PMS(" or ")<<w_bar_raw2<<PMS(" "));
    if (trouble_peaks_counter>=2)
    {
        GLOB_DEBUG(PMS("Too many peaks:")<<trouble_peaks_counter<<PMS("\n\r"));
        *p_std=0;
        return 0;//this set of data should be disregarded due to too many peaks
    }
    else
    {
        GLOB_DEBUG(PMS("which of 2..."));
        for (char i=0;i<FFT_PER_CYCLE;i++)

```

```

    {
        for (char j=0;j<2;j++)
        {
            if(cycle_peaks_freq[i][j]==w_bar_raw)
            {
                sum_mag1+=cycle_peaks_mag[i][j][1];
            }
            else if (cycle_peaks_freq[i][j]==w_bar_raw2)
            {
                sum_mag2+=cycle_peaks_mag[i][j][1];
            }
        }
    }
    GLOB_DEBUG(PMS("\nrmag1: ")<<sum_mag1<<PMS(" mag2: ")<<sum_mag2<<PMS("\n  ↗
    \r"));
    if(sum_mag2>sum_mag1)
    {
        w_bar_raw=w_bar_raw2;
    }
    else if (sum_mag2==sum_mag1)
    {
        GLOB_DEBUG(PMS("is hard..."));
        if((bin[w_bar_raw2+1]+bin[w_bar_raw2]+bin[w_bar_raw2-1])>(bin[w_bar_raw  ↗
        +1]+bin[w_bar_raw]+bin[w_bar_raw-1]))
        {
            w_bar_raw=w_bar_raw2;
        }
        else if ((bin[w_bar_raw2+1]+bin[w_bar_raw2]+bin[w_bar_raw2-1])==(bin  ↗
        [w_bar_raw+1]+bin[w_bar_raw]+bin[w_bar_raw-1]))
        {
            GLOB_DEBUG(PMS("Give up!!!"));
            *p_std=0;
            return 0;
        }
    }
}

//GLOB_DEBUG(PMS("bin[w_bar_raw-1]: ")<<bin[w_bar_raw-1]<<PMS(" "));
//GLOB_DEBUG(PMS("w_bar_raw: ")<<w_bar_raw<<PMS(" "));

//GLOB_DEBUG(PMS("bin[w_bar_raw]: ")<<bin[w_bar_raw]<<PMS(" "));
//GLOB_DEBUG(PMS("bin[w_bar_raw+1]: ")<<bin[w_bar_raw+1]<<PMS(" "));
//GLOB_DEBUG(PMS("n: ")<<n<<PMS(" "));

for (char i=0;i<FFT_PER_CYCLE;i++)
{
    for (char j=0;j<2;j++)
    {
        if((cycle_peaks_freq[i][j]<=(w_bar_raw+1))&&(cycle_peaks_freq[i][j]>=  ↗
        (w_bar_raw-1)))
        {

```

```

        adjust+=(short)((long)(cycle_peaks_mag[i][j][2]-cycle_peaks_mag[i][j]
[0])*100/(cycle_peaks_mag[i][j][0]+cycle_peaks_mag[i][j][1]
+cycle_peaks_mag[i][j][2]));
        w_counter++;

        //GLOB_DEBUG(PMS("avg w: ")<<w_bar<<PMS(" "));
        //GLOB_DEBUG(PMS("freq: ")<<cycle_peaks_freq[i][j]<<PMS(" "));
        //GLOB_DEBUG(PMS("n: ")<<n<<PMS(" "));
    }
}

w_bar= (short)((long)(bin[w_bar_raw-1]*(w_bar_raw-1)+bin[w_bar_raw]*(w_bar_raw)+bin
[w_bar_raw+1]*(w_bar_raw+1))*BIN_WIDTH_HZ/
        w_counter);

for (char i=0;i<FFT_PER_CYCLE;i++)
{
    for (char j=0;j<2;j++)
    {
        if((cycle_peaks_freq[i][j]<=(w_bar_raw+1))&&(cycle_peaks_freq[i][j]>=
(w_bar_raw-1)))
        {
            short n=(cycle_peaks_freq[i][j]*BIN_WIDTH_HZ)-w_bar;
            diff_sum+= (unsigned long)n*n;
            //GLOB_DEBUG(PMS("avg w: ")<<w_bar<<PMS(" "));
            //GLOB_DEBUG(PMS("freq: ")<<cycle_peaks_freq[i][j]<<PMS(" "));
            //GLOB_DEBUG(PMS("n: ")<<n<<PMS(" "));
        }
    }
}

//if(w_bar_raw==0)
//{
//    w_bar_raw=1;
//}

if(w_counter<2)
{
    w_counter=2;
}
GLOB_DEBUG(PMS("w_counter: ")<<w_counter<<PMS(" "));
GLOB_DEBUG(PMS(" ")<<diff_sum<<PMS(" "));
*p_std=sqrt((diff_sum/(w_counter-1)));

//GLOB_DEBUG(PMS("\n\rwbarraw: ")<<w_bar_raw<<PMS("\n\r"));
//GLOB_DEBUG(PMS("\n\rwbarraw2: ")<<w_bar_raw2<<PMS("\n\r"));
//    GLOB_DEBUG(PMS("\n\rw_counter: ")<<w_counter<<PMS("\n\r"));
//    GLOB_DEBUG(PMS("\n\rtrouble counter: ")<<trouble_peaks_counter<<PMS("\n\r"));
//    GLOB_DEBUG(PMS("\n\r")<<(bin[w_bar_raw-1]+bin[w_bar_raw]+bin[w_bar_raw+1])<<PMS("
compare to ")<<w_counter<<PMS("\n\r"));

```

```

#else //NUM_SAMPLES>=256 if higher than 1024 is needed, please change freq_sum to
unsigned long.
    short bin[128]={0};
    short w_bar_raw=0;
    short w_bar_raw2=0;
    short freq_sum=0;

    //tabulate all the data points into 128bins
    for (char i=0;i<FFT_PER_CYCLE;i++)
    {
        for (char j=0;j<2;j++)
        {
            //unsigned short freq_bin=(cycle_peaks_freq[i][j])/SAMPLES_OVER128;
            bin[(cycle_peaks_freq[i][j])/SAMPLES_OVER128]++;
            //GLOB_DEBUG(cycle_peaks_freq[i][j]<<PMS(" "));

        }
    }

    //for (short i=0;i<128;i++)
    // GLOB_DEBUG(bin[i]<<PMS(" "));
    bin[0]=0;
    //find the bin with the most data exist to examine whethr it is natural frequency
    for (short i=1;i<127;i++)
    {
        if(bin[i]>index)
        {
            index=bin[i];
            w_bar_raw= i;
        }
    }
    //if there are more bins having the same amount of data points, then store them(at
    the end only store the one with highest frequency
    for (short i=(w_bar_raw+1);i<127;i++)
    {
        if(bin[i]==index)
        {
            trouble_peaks_counter++;
            w_bar_raw2=i;
        }
    }

    //GLOB_DEBUG(PMS("\nrwbarraw: ")<<w_bar_raw<<PMS("\nr"));
    //GLOB_DEBUG(PMS("\nrwbarraw2: ")<<w_bar_raw2<<PMS("\nr"));
    //GLOB_DEBUG(PMS("\nrtrouble counter: ")<<trouble_peaks_counter<<PMS("\nr"));

    //determine what happen if more than 1 bin have the same amount of data points.
    if(trouble_peaks_counter!=0)
    {
        GLOB_DEBUG(PMS("Deciding...")<<w_bar_raw<<PMS(" or ")<<w_bar_raw2<<PMS(" "));
    }

```

```

//if there are more than 2 frequency having the same amount of data points,
then, this is too hard to determine, and very possibly this is not useful
if (trouble_peaks_counter>=2)
{
    GLOB_DEBUG(PMS("Too many peaks:")<<trouble_peaks_counter<<PMS("\n\n"));
    *p_std=0;
    return 0;//this set of data should be disregarded due to too many peaks
}
else
{
    GLOB_DEBUG(PMS("which peak of 2..."));
    for (char i=0;i<FFT_PER_CYCLE;i++)
    {
        for (char j=0;j<2;j++)
        {
            GLOB_DEBUG(cycle_peaks_freq[i][j]);
            if((cycle_peaks_freq[i][j]/SAMPLES_OVER128)==w_bar_raw)
            {
                sum_mag1+=cycle_peaks_mag[i][j][1];
            }
            else if ((cycle_peaks_freq[i][j]/SAMPLES_OVER128)==w_bar_raw2)
            {
                sum_mag2+=cycle_peaks_mag[i][j][1];
            }
        }
    }
    GLOB_DEBUG(PMS("\n\nrmag1: ")<<sum_mag1<<PMS(" mag2: ")<<sum_mag2<<PMS("\n\n\n"));
    if(sum_mag2>sum_mag1)
    {
        w_bar_raw=w_bar_raw2;
    }
    else if (sum_mag2==sum_mag1)
    {
        GLOB_DEBUG(PMS("is hard..."));
        //if there are only two frequency that has same amount of data points,
        then pick the one with more data points in the neighboring bins.
        if((bin[w_bar_raw2+1]+bin[w_bar_raw2]+bin[w_bar_raw2-1])>(bin[w_bar_raw
+1]+bin[w_bar_raw]+bin[w_bar_raw-1]))
        {
            w_bar_raw=w_bar_raw2;
        }
        //if the neighboring bins have the same data points too, too bad, it's
        too hard to determine, or probably it not useful anyways.
        else if ((bin[w_bar_raw2+1]+bin[w_bar_raw2]+bin[w_bar_raw2-1])==(bin
[w_bar_raw+1]+bin[w_bar_raw]+bin[w_bar_raw-1]))
        {
            GLOB_DEBUG(PMS("Give up!!!"));
            *p_std=0;
            return 0;
        }
    }
}

```



```

    }
}
for (char i=0;i<FFT_PER_CYCLE;i++)
{
    for (char j=0;j<2;j++)
    {
        if ((w_bar_raw+1)>=(cycle_peaks_freq[i][j]/SAMPLES_OVER128)&&(w_bar_raw-1)<=
(cycle_peaks_freq[i][j]/SAMPLES_OVER128))
        {
            freq_sum+=cycle_peaks_freq[i][j];
            w_counter++;
            adjust+=(short)((long)(cycle_peaks_mag[i][j][2]-cycle_peaks_mag[i][j]
[0])*100/(cycle_peaks_mag[i][j][0]+cycle_peaks_mag[i][j][1]
+cycle_peaks_mag[i][j][2]));
        }
    }
}

w_bar= (short)((long)(freq_sum)*BIN_WIDTH_HZ/w_counter);

for (char i=0;i<FFT_PER_CYCLE;i++)
{
    for (char j=0;j<2;j++)
    {
        if((w_bar_raw+1)>=(cycle_peaks_freq[i][j]/SAMPLES_OVER128)&&(w_bar_raw-1)<=
(cycle_peaks_freq[i][j]/SAMPLES_OVER128))
        {
            short n=(cycle_peaks_freq[i][j]*BIN_WIDTH_HZ)-w_bar;
            diff_sum+= (unsigned long)n*n;
        }
    }
}
if(w_counter<2)
{
    w_counter=2;
}
//GLOB_DEBUG(PMS(" ")<<diff_sum<<PMS(" "));
*p_std=isqrt((diff_sum/(w_counter-1)));

//GLOB_DEBUG(PMS("\n\r")<<(bin[w_bar_raw-1]+bin[w_bar_raw]+bin[w_bar_raw+1])<<PMS("
compare to ")<<w_counter<<PMS("\n\r"));
w_bar_final=w_bar+(short)((((long)adjust*BIN_WIDTH_HZ)/w_counter)/100);

#endif

GLOB_DEBUG(PMS("\n\rThe Standard deviation is: ")<<*p_std<<PMS("\n\r"));
return w_bar_final;
}

//Getting a compensated average frequency
void fft_int::get_wo_3p(short data[], short peaks[][10], short peaks_adj[], long w_o_mHz[])

```

```

//peaks[x][y] =>x=0 means frequency, x=1 means value
for (char i=0;i<3;i++)
{
    //the i amount of datat (10 here) that has the adjust value of the peak (in mHz)
    peaks_adj[i]=0;
    w_o_mHz[i]=0;
    if (peaks[1][i]!=0)
    {
        //if it is not at the end point and it is truely the peak of 3 points
        if ((peaks[0][i]>1) && (peaks[0][i]<(NUM_SAMPLES-1)) && (data[(peaks[0][i])] >
            (data[(peaks[0][i]+1)]) && (data[(peaks[1][i])] > (data[(peaks[1][i]-1)])))
        {
            peaks_adj[i]=(short)((((long)((data[(peaks[0][i]+1)])-(data[(peaks[0]
                [i]-1)])))*100)/((data[(peaks[0][i]+1)]+(data[(peaks[0][i]-1)]+(data[peaks
                [0][i]]))));
        }
    }
    w_o_mHz[i]=(peaks[0][i]*BIN_WIDTH_HZ+peaks_adj[i]*BIN_WIDTH_HZ/100);
}

return;
}

//reset the set number flag
void fft_int::cycle_reset(void)
{
    peak_set_num=0;
}

/* Modified from http://www.dattalo.com/technical/theory/sqrt.html to incorporate isqrt of
long*/
unsigned short fft_int::isqrt(unsigned long N)
{
    unsigned short x=0;
    for(unsigned short j=(1<<15); j!=0; j>>=1)
    {
        x = x + j;
        if( (unsigned long)x*x > N)
        {
            x = x - j;
        }
    }
    //GLOB_DEBUG(PMS("Square root of ")<<N<<PMS(" is")<<x<<PMS("\n\r"));
    return(x);
}

```

```
//=====
/** \file sensor.h
 *   This function provide a way to read sensor
 *
 * Revisions:
 *   \li 05-27-10 CKC First revision
 *   \li 06-22-11 CKC Revised and fully commented code
 *
 * License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 */
//=====

/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _SENSOR_H_
#define _SENSOR_H_

#include "avr_adc.h"

class sensor//: public avr_adc
{
protected:
    // The fft class needs a pointer to the serial port used to say hello. This
    // base class pointer could also be used for a wireless radio

public:
    sensor(base_text_serial*);
    ~sensor(void);
    // This could be a function to read one channel once, returning the result as
    // an unsigned integer. The parameter is the channel number
    // unsigned short read_once (unsigned char channel);

    //set up to read x-axis of accelerometer
    void read_accx (void);
    //set up to read y-axis of accelerometer
    void read_accy (void);
    //set up to read z-axis of accelerometer
    void read_accz (void);
    //Let the user know the state of data
    unsigned char sensor_data_state (void);
    //Reset data state
    void reset_data_state (void);
    //Let the user know whether it is time to go into a idle mode
    bool is_nap_time (void);
    //Send out the whole spectrum of data out
    void send_serial (void);

};

#endif // _SENSOR_H_

```

```
//=====
/** \file sensor.cpp
 *   This function provide a way to read sensor
 *
 * Revisions:
 *   \li 05-27-10 CKC First revision
 *   \li 10-04-10 CKC Works on Atmega128 and need to work on codes for atmega324P, which
 *           is the platform for final product at the thesis defense.
 *   \li 06-22-11 CKC Revised and fully commented code
 *
 * License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 *
 * Notes: In order to be pin compatible to Atmega324P, The timer has to change and all
 *         the register associate with it. The ADC registers can still remain the same. By
 *         default, this file uses 16 bit Timer 3, but if the microcontroller doesn't have
 *         Timer 3, it will uses 16 bit Timer 1 instead. This file also require configuration
 *         of the pins to the accelerometer for the board that is use.
 */
//=====

#include <avr/interrupt.h>           //To use interrupt
#include <avr/io.h>
#include "lib/global_debug.h"       // Include header for serial port class
#include "stdint.h"
#include "lib/rs232int.h"
#include "sensor.h"                 // Include header file
#include "CBM_v1_0.h"               // Alterable constants located here

////*****All the following are all included inside CBM_v1_0.h, if this were to used
////seperately, please change the following value
#define CLK_PER_SAMPLE 2083
#define NUM_SAMPLES 256L
#define WAITING 0
#define GET_DATA 1
#define PROCESSING 2

//about it prints 1.05ms per character via serial
/* task_timer my_timer;

//make time stamp objects
time_stamp time0=0;
time_stamp time1=0;
time_stamp time2=0;*/

#define ADC_PRESCALE 0x04           //< Default prescaler setting

//Setup the channel of ADC in various mcu
#ifndef ETT_ATMEGA128
```

```

    //We mainly use X-axis
    #define X_AXIS 0
    #define Y_AXIS 1
    #define Z_AXIS 2

#elif defined SWOOP_BOARD
    //We mainly use X-axis
    #define X_AXIS 7
    #define Y_AXIS 6
    #define Z_AXIS 5

#else
    #error "The Accelerometer axis pins are not yet configured in file sensor.cpp"
#endif

// In order to get 16-bit data
typedef union
{
    unsigned int whole;           //< The data as a 16-bit int
    unsigned char half[2];       //< The data as an array of 8-bit chars
} u16union;

//All these used for timer interrupt
u16union oc_time={0};
u16union adc_read={0};
unsigned char data_state=0;
unsigned char adc_channel=0;
unsigned short num_int=0;
unsigned short dummy_counter=0;
bool sleep_flag=false;

short data2[NUM_SAMPLES]={0};

sensor::sensor(base_text_serial* p_serial_port)
{
    ADMUX = 0;                      // Set reference to AVCC pin voltage
    //This make the reading left justified
    ADMUX |= (1<<ADLAR);
    ADCSRA = (1 << ADEN) | ADC_PRESCALE;    // Set clock prescaler and turn A/D on

    //CTC mode only works with Register A, not B nor C
    #ifdef TCCR3A
    //CTC mode prescaler clk/256
    TCCR3A  &=~((1<<WGM30)|(1<<WGM31));
    TCCR3B  &=~((1<<WGM33)|(1<<CS31)|(1<<CS30));

    TCCR3B  |= ((1<<WGM32)|(1<<CS32));
    //0.025008s per interrupt for 1563.
    oc_time.whole=CLK_PER_SAMPLE;
    OCR3AH= oc_time.half[1];
    
```

```

OCR3AL= oc_time.half[0]; //391;

//conversion enable
ETIMSK |= ((1<<OCIE3A));

#else
//CTC mode prescaler clk/256
TCCR1A &=~((1<<WGM10)|(1<<WGM11));
TCCR1B &=~((1<<WGM13)|(1<<CS11)|(1<<CS10));
TCCR1B |= ((1<<WGM12)|(1<<CS12));
// TIMSK1 |= ((1<<OCIE1B));
oc_time.whole=CLK_PER_SAMPLE;
OCR1AH= oc_time.half[1];
OCR1AL= oc_time.half[0]; //391;

//conversion enable
TIMSK1 |= ((1<<OCIE1A));

#endif
GLOB_DEBUG (PMS ("Sensor setup done\n\r"));
}
sensor::~sensor(void)
{
}

//x-axis
void sensor::read_accx (void)
{
    if (data_state==0)
    {
        data_state=1;
        num_int=0;
        adc_channel=X_AXIS;
        // Set the channel
        ADMUX &= 0b11111000; // Clear the channel bits
        ADMUX |= (adc_channel & 0x07); // Put channel bits in the register
    }

    return;
}

//Y-axis
void sensor::read_accy (void)
{
    if (data_state==0)
    {
        data_state=1;
        num_int=0;
        adc_channel=Y_AXIS;
        // Set the channel

```

```
    ADMUX &= 0b11111000;           // Clear the channel bits
    ADMUX |= (adc_channel & 0x07);  // Put channel bits in the register

}
return;
}

//Z-axis
void sensor::read_accz (void)
{
    if (data_state==0)
    {
        data_state=1;
        num_int=0;
        adc_channel=Z_AXIS;
        // Set the channel
        ADMUX &= 0b11111000;           // Clear the channel bits
        ADMUX |= (adc_channel & 0x07);  // Put channel bits in the register

    }
    return;
}

//Send out the whole spectrum of data out
void sensor::send_serial (void)
{
    GLOB_DEBUG (PMS ("data2 size:")<<(sizeof(data2))<<PMS("\n\n"));
    for(int num=0;num<(sizeof(data2))/2;num++)
    {
        GLOB_DEBUG (num<<PMS (": ")<<data2[num]<<PMS(" "));
    }
    return;
}

//Let the user know the state of data
unsigned char sensor::sensor_data_state (void)
{
    return data_state;
}

//Reset data state
void sensor::reset_data_state (void)
{
    data_state=0;
}

//Let the user know whether it is time to go into a idle mode
bool sensor::is_nap_time (void)
{
    return sleep_flag;
}
```

```
//Begining of interrupt code
#ifdef TCCR3A
ISR(TIMER3_COMPA_vect)

#else
ISR(TIMER1_COMPA_vect)
#endif
{
    sleep_flag=false;

    //Keep sampling as long as needed
    if(data_state==1)
    {
        sleep_flag=true;
        if (num_int!=0)
        {
            adc_read.half[0] = ADCH;
            adc_read.half[1] = 0;

            data2[num_int-2]=adc_read.whole;
            data2[num_int-1]=0;

        }
        //if it reach the last sample, do not sample anymore
        if (num_int>=NUM_SAMPLES)
        {
            //back to waiting
            data_state=2;
            sleep_flag=false;
        }
        num_int+=2;
    }
}
```



```
//=====
/** \file avrsleep.h
 *   This is a header file for setting up an easy to use sleep function with Timer
 *   interrupt (0)(could be use for RTC)
 *
 * Revisions:
 *   \li 10-12-10 CKC initialized
 *   \li 06-23-11 CKC Revised and fully commented code
 *
 * License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 */
//=====
/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _AVRSLEEP_H_
#define _AVRSLEEP_H_

#include <avr/sleep.h>

class avrsleep
{
protected:

public:
    avrsleep(base_text_serial*);
    ~avrsleep(void);

    void sleep_idle(void);
    void sleep_adc(void);
    void sleep_power_down(void);
    void sleep_power_save(void);
    void sleep_standby(void);
    void sleep_ext_standby(void);
    unsigned short get_counter (void);
    void reset_sleep_counter (void);

};

#endif // _AVRSLEEP_H_
```

```
//=====
/** \file avr_sleep.cpp
 *   This is a file for setting up an easy to use sleep function with Timer interrupt
 *   (0)(could be use for RTC)
 *
 * Revisions:
 *   \li 10-12-10 CKC initialized
 *   \li 06-23-11 CKC Revised and fully commented code
 *
 * License:
 *   This file released under the Lesser GNU Public License. The program is intended
 *   for educational use only, but its use is not restricted thereto.
 *
 * Save Power:
 *   This file is trying to minimize power usage of the MCU. Here are a few suggested
 *   by the datasheet (according to atmega128)
 *   1) Turn off adc ~300uA
 *   2) Turn off Analog Converter (in idle/adc), other modes automatically turn it
 *      off by themselves ~200uA
 *   3) Disable Brown-out Detector
 *   4) Diabile Internal Voltage Reference
 *   5) Disable Watch Dog Timer by 2 operation
 *   6) Pull port pin output ~200-500uA
 *   7) Disable either Jtag, or on chip debug
 *
 */
//=====
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <avr/wdt.h>
#include "lib/global_debug.h"
#include "lib/rs232int.h"
#include "avrsleep.h"
#include "CBM_v1_0.h"

unsigned short sleep_counter=0;
bool sleep_done=true;

avrsleep::avrsleep(base_text_serial* p_serial_port)
{
    //the Atmega 128 is the weird one that uses Timer0 for asynchronus operation because all
    //others use Timer2
    #if defined __AVR_ATmega128__
        //disable interrupt
        TIMSK&= ~((1<<OCIE0)|(1<<TOIE0));
        //set up for enable external crystal interrupts
        ASSR|=(1<<AS0);
    #endif
}
```

```

    TCNT0=0;
    //Reset TCCR0 bits and set clk/128 (1sec per interrupt)
    //TCCR0=0x05;
    //Reset TCCR0 bits and set clk/1024 (8sec per interrupt)
    TCCR0=0x04;
    //To make sure all the register are ready
    while (ASSR&(TCN0UB| OCR0UB| TCR0UB))
    {}
    //clear timer0 flag
    // TIFR=(1<< OCF0);
    //reenable output compare interrupt
    TIMSK|= (1<<TOIE0);

    #else

        //disable interrupts
        TIMSK2 &= ~((1<<OCIE2B)|(1<<TOIE2));
        //set up for enable external crystal interrupts
        ASSR|=(1<<AS2);
        TCNT2=0;
        //Reset TCCR0 bits and set clk/1024 (8sec per interrupt)
        TCCR2B=0x04;
        //Reset TCCR0 bits and set clk/128 (1sec per interrupt)
        //TCCR2B=0x05;

        //To make sure all the register are ready
        while (ASSR&(TCN2UB| OCR2AUB| OCR2BUB| TCR2AUB| TCR2BUB))
        {}
        //clear timer2 flag
    // TIFR2=(1<< OCF2B);
    //reenable output compare interrupt
    TIMSK2|= (1<<TOIE2);

    #endif

    wdt_disable();
    MCUCR|=(1<<JTD);
    MCUCR|=(1<<JTD);
    //cli();
    //need to enable Watch Dog Timer Change first, and within 4 clk, it need to perform
    operation
    //within 4 clk of change enable, it disable the Watch Dog Timer

    //default to 16sec of sleep

    GLOB_DEBUG (PMS ("Sleep setup done\n\r "));
}

avrsleep::~avrsleep(void)
{
}

```

//Sleep Idle- The sleeping mode that is the fastest to wake up, but draws the most power

void avrsleep::sleep_idle(void)

```
{  
  
    unsigned char acr=ACSR;  
    //turn off Analog Comparator  
    ACSR |= (1<<ACD);  
  
    unsigned char pr=PRR;  
    PRR=0b10110110;  
  
    set_sleep_mode(SLEEP_MODE_IDLE);  
    cli();  
  
    PRR=pr;  
    ACSR=acr;  
}
```

//Draws a lot less power than Idle

void avrsleep::sleep_adc(void)

```
{  
    //turn off Analog Comparator  
    ACSR |= (1<<ACD);  
    set_sleep_mode(SLEEP_MODE_ADC);  
    cli();  
    sleep_enable();  
    sei();  
    sleep_cpu();  
    sleep_disable();  
}
```

//draws the lowest amount of current, 0.1uA

void avrsleep::sleep_power_down(void)

```
{  
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);  
    cli();  
    sleep_enable();  
    sei();  
    sleep_cpu();  
    sleep_disable();  
}
```

//Second lowest amount of current draw, but since the code needs a timer, so this is the
//best sleeping mode for the project

void avrsleep::sleep_power_save(void)

```
{  
  
    //check to see whether serial port buffer cleared or not, if not, set cpu to sleep in  
    //idle til it completely cleared  
    #if defined UCSR1A
```

```
    if (!(UCSR1A & TXC1))
    {
        //enable the transmit complete interrupt, so that it can wake up in idle and
        //sleep in power save
        UCSR1B|=(1<<TXCIE1);
        sleep_idle();
    }
    if (!(UCSR0A & TXC0))
    {
        //enable the transmit complete interrupt, so that it can wake up in idle and
        //sleep in power save
        UCSR0B|=(1<<TXCIE0);
        sleep_idle();
    }
}
#else
    if (!(UCSR0A & TXC0))
    {
        //enable the transmit complete interrupt, so that it can wake up in idle and
        //sleep in power save
        UCSR0B|=(1<<TXCIE0);
        sleep_idle();
    }
}

#endif

//analog comparator
#ifdef DIDR1
    unsigned char di1=DIDR1;
    DIDR1= 0b00000011;
#endif

#ifdef DIDR0
    unsigned char di0=DIDR0;
    DIDR0= 0b11111111;
#endif

unsigned char spr=SPCR;
unsigned char uc1b=UCSR1B;
unsigned char adcr=ADCSRA;

//SPI disabled
SPCR &= ~(1<<SPE);
//Disable UART Transmit and Receive
UCSR1B &= 0b11100111;

//disable the analog to digital converter before going to sleep
ADCSRA &= ~(1 << ADEN);
//PRR=0b11111111;

#ifdef __AVR_ATmega128__
    //make sure all ports are default output to save the maximum power during sleep
```

```
    DDRA=0b11111111;
    DDRB=(DDRB|0b11000001);
    DDRC=0b11111111;
    DDRD=(DDRD|0b11110010);
    DDRE=(DDRE|0b11111100);
    DDRF=(DDRF|0b00001100);
    DDRG=(DDRG|0b11100111);

    #elif defined __AVR_ATmega644__ || defined __AVR_ATmega324P__
        DDRA=0b00001111;
        DDRB=0b10111011;
        DDRC=0b00111111;
        DDRD=0b11111011;

    #endif

    set_sleep_mode(SLEEP_MODE_PWR_SAVE);

    cli();
    sleep_enable();
    sleep_bod_disable();
    sei();
    sleep_cpu();
    sleep_disable();

    //restored adc
    ADCSRA=adcr;
    //restored SPI
    SPCR=spr;
    //restored UART
    UCSR1B=uc1b;

    //Restores the pins
    #ifdef DIDR1
        DIDR1= di1;
    #endif

    #ifdef DIDR0
        DIDR0= di0;
    #endif

}

//Third lowest current draw sleep mode
void avrsleep::sleep_standby(void)
{
    set_sleep_mode(SLEEP_MODE_STANDBY);
    cli();
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
}
```

```
}

//Draws slightly less current than sleep_adc
void avrsleep::sleep_ext_standby(void)
{
    set_sleep_mode(SLEEP_MODE_EXT_STANDBY);
    cli();
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
}

//Let the user to know how many sleep has gone through
unsigned short avrsleep::get_counter (void)
{
    return sleep_counter;
}

//Reset the sleep counter
void avrsleep::reset_sleep_counter (void)
{
    sleep_counter=0;
}

//ATmega 128 is the odd one that uses timer0 for RTC
#ifdef __AVR_ATmega128__
    ISR(TIMER0_OVF_vect)
#else
    ISR(TIMER2_OVF_vect)
#endif
{
    sleep_counter++;
}

//if serial transmission is complete, then shut it off
#ifdef UCSR1A

    ISR(USART1_TX_vect)
    {
        //disable transmit complete interrupt
        UCSR1B|=(1<<TXCIE1);
    }
    ISR(USART0_TX_vect)
    {
        //disable transmit complete interrupt
        UCSR0B|=(1<<TXCIE0);
    }
#else
    ISR(USART0_TX_vect)
```

```
{  
    //disable transmit complete interrupt  
    UCSR0B|=(1<<TXCIE0);  
}  
#endif
```



```

//*****
/** \file nRF24L01_base.h
 *   This file contains a class full of utility functions to work with a Nordic
 *   nRF24L01 based radio module.
 *
 * Revised:
 *   \li 04-22-07 MNL Original file
 *   \li 04-26-07 MNL Added a bunch of functions. Transfer verified working.
 *   \li 05-09-07 MNL Changed so that setup() doesn't have to be called by user
 *   \li 11-15-07 SCH Overhauled to properly control radios and increase speed
 *   \li 01-26-07 SCH Fixed more major flaws
 *   \li 02-15-08 JRR Text based version, written for use on ME405 boards
 *   \li 02-17-08 JRR nRF24L01_base split off from nRF24L01_text
 *   \li 10-11-08 JRR Added control for enhanced/broadcast mode, long addresses
 *   \li 10-11-08 JRR Changed SPI port from bit-banged to SPI hardware
 */
//*****

/// These defines prevent this file from being included more than once in a *.cc file
#ifndef _NRF24L01_BASE_H_
#define _NRF24L01_BASE_H_

#include "lib/base_text_serial.h"           // Header for base serial devices

#define NRF24_MAX_PKT_SZ    32              ///< Maximum packet size for the radio
#define NRF24_SPI_TIMEOUT    1000           ///< Retries until timeout for SPI port

#define NRF24_TRANSMIT       0b00000000    ///< Status bit puts radio in transmit mode
#define NRF24_RECEIVE        0b00000001    ///< Status bit puts radio in receive mode

// Define interrupt types for use in virtual interrupt handler
#define NRF24_INT_RX         0b01000000    ///< Mask (block) data received interrupt
#define NRF24_INT_TX         0b00100000    ///< Mask data successfully send interrupt
#define NRF24_INT_M_RT       0b00010000    ///< Mask send-has-failed interrupt

// Define CRC ON/OFF
#define NRF24_CRC_ON         0b00001000    ///< Activates automatic CRC checking
#define NRF24_CRC_2_BYTE     0b00000100    ///< Sets CRC to two bytes rather than one

// Define address width.
#define NRF24_AW_3           0b00000001    ///< Specifies address width of 3 bytes
#define NRF24_AW_4           0b00000010    ///< Specifies address width of 4 bytes
#define NRF24_AW_5           0b00000011    ///< Specifies address width of 5 bytes

// Define data rate and power output
#define NRF24_DR_1M          0b00000001    ///< Put radio at 1 MB/s data rate
#define NRF24_DR_2M          0b00001001    ///< Put radio at 2 MB/s data rate
#define NRF24_RF_POW_0       0b00000111    ///< Set power level to 0 dB (highest)
#define NRF24_RF_POW_N6     0b00000101    ///< Set power level to -6 dB
#define NRF24_RF_POW_N12    0b00000011    ///< Set power level to -12 dB
#define NRF24_RF_POW_N18    0b00000001    ///< Set power level to -18 dB (lowest)

```

```

// These are commands to be sent to the radio
#define nRF24_RD_REG      0b00000000    ///< Command to read a register
#define nRF24_WR_REG      0b00100000    ///< Command to write a register
#define nRF24_RD_PLD      0b01100001    ///< Command to read payload buffer
#define nRF24_WR_PLD      0b10100000    ///< Command to write payload buffer
#define nRF24_FLUSH_TX    0b11100001    ///< Command to flush transmitter buffer
#define nRF24_FLUSH_RX    0b11100010    ///< Command to flush receiver buffer
#define nRF24_REUSE_TX_PLD 0b11100011    ///< Command to re-send transmitted data
#define nRF24_NOP          0b11111111    ///< Command does nothing but read status

// REG_ADDR - Address of registers to send commands to.
#define nRF24_REG_CONF      0x00    ///< Address of configuration register
#define nRF24_REG_EN_AA     0x01    ///< Register enables Auto-Ack for pipes
#define nRF24_REG_EN_RXADDR 0x02    ///< Register enables RX addresses (pipes)
#define nRF24_REG_SETUP_AW  0x03    ///< Address width (3 - 5 bytes) register
#define nRF24_REG_SETUP_RETR 0x04    ///< Setup retransmit register
#define nRF24_REG_RF_CH     0x05    ///< RF Channel register
#define nRF24_REG_RF_SETUP  0x06    ///< RF Setup register
#define nRF24_REG_STATUS    0x07    ///< Status register always returned by SPI
#define nRF24_REG_OBS_TX    0x08    ///< Observe transmission register
#define nRF24_REG_CD        0x09    ///< Carrier Detect (1 bit) status register
#define nRF24_REG_RX_ADDR_P0 0x0A    ///< Address for Pipe 0 (5 bytes)
#define nRF24_REG_RX_ADDR_P1 0x0B    ///< Address for Pipe 1 (5 bytes)
#define nRF24_REG_RX_ADDR_P2 0x0C    ///< Address for Pipe 2 (1 byte)
#define nRF24_REG_RX_ADDR_P3 0x0D    ///< Address for Pipe 3 (1 byte)
#define nRF24_REG_RX_ADDR_P4 0x0E    ///< Address for Pipe 4 (1 byte)
#define nRF24_REG_RX_ADDR_P5 0x0F    ///< Address for Pipe 5 (1 byte)
#define nRF24_REG_TX_ADDR    0x10    ///< Address to which data is transmitted
#define nRF24_REG_PW_P0     0x11    ///< Data size in FIFO register for Pipe 0
#define nRF24_REG_PW_P1     0x12    ///< Data size in FIFO register for Pipe 1
#define nRF24_REG_PW_P2     0x13    ///< Data size in FIFO register for Pipe 2
#define nRF24_REG_PW_P3     0x14    ///< Data size in FIFO register for Pipe 3
#define nRF24_REG_PW_P4     0x15    ///< Data size in FIFO register for Pipe 4
#define nRF24_REG_PW_P5     0x16    ///< Data size in FIFO register for Pipe 5
#define nRF24_REG_FIFO_STATUS 0x17    ///< Register holds status of FIFO's

// Bits in the status register
#define nRF24_RX_DR         0b01000000    ///< Bitmask for data received
#define nRF24_TX_DS         0b00100000    ///< Bitmask for transmission complete
#define nRF24_MAX_RT        0b00010000    ///< Bitmask for retries maxed out
#define nRF24_RX_P_NO       0b00001110    ///< Bitmask for pipe number (3 bits)
#define nRF24_TX_FULL       0b00000001    ///< Bitmask for transmitter full

// Bits in the configuration register (use RX_DR, TX_DS, MAX_RT above for masking)
#define nRF24_EN_CRC        0b00001000    ///< Bitmask to enable CRC
#define nRF24_CRCO          0b00000100    ///< Bitmask for CRC encoding, 1 or 2 bit
#define nRF24_PWR_UP        0b00000010    ///< Bitmask for power-up bit
#define nRF24_PRIM_RX       0b00000001    ///< Bitmask for receive or transmit

// Standard configurations for transmit, receive, and owl stretching time
/// This configuration puts the radio in receive mode with only RX interrupts enabled

```

```

#define nRF24_RECV_MODE      (nRF24_EN_CRC | nRF24_PWR_UP | nRF24_PRIM_RX \
                              | nRF24_INT_TX | nRF24_INT_M_RT)

/// This configuration puts the radio in transmit mode with only RX interrupts enabled
#define nRF24_XMIT_MODE      (nRF24_EN_CRC | nRF24_PWR_UP \
                              | nRF24_INT_TX | nRF24_INT_M_RT)

//-----
/** This class operates a radio module based on a Nordic nRF24L01 chip in text mode.
 * As a base class, this class is intended to be extended; its descendents will
 * implement text-based connections and packet-based connections.
 *
 * The current version uses the SPI port hardware rather than a bit-banged serial
 * port. This is more efficient, though it precludes monitoring SPI bits.
 *
 * Lines which are connected to the radio on the ME405 board, in addition to the
 * usual Vcc and ground (radio pin first, CPU pin second; board V0.5A/V0.60/Swoop):
 * \li MISO - MISO      - SPI data, AVR to radio
 * \li MOSI - MISO      - SPI data, radio to AVR
 * \li SCK  - SCK       - SPI clock line
 * \li CSN  - SS        - SPI slave select
 * \li CE   - E7/D1/B3  - Chip enable selects transmit or receive mode
 * \li IRQ  - E6/E0/B2  - Interrupt (radio interrupts CPU when data is received)
 */

class nRF24L01_base
{
    // Protected data and methods are accessible from this class and its descendents
protected:
    /// This is the I/O port to which the Chip Enable (CE) line is connected
    volatile unsigned char* port_CE;

    /// This bitmask has a 1 for the Chip Enable (CE) line
    unsigned char mask_CE;

    /// This is the I/O port for the Slave Select line (SS on CPU, CSN on radio)
    volatile unsigned char* port_SS;

    /// This is a bitmask for the Slave Select (SS on CPU, CSN on the radio) bit
    unsigned char mask_SS;

    /// This is a pointer to a serial port object which is used for debugging the
    /// radio modem code. Left blank, it defaults to NULL and no debugging info
    base_text_serial* dbg_port;

    /// This switch is true if the radio is to be used in enhanced shockburst mode
    /// with checksums, automatic retransmission, etc.
    bool in_enhanced_mode;

    // Public methods can be called from anywhere in the program where there is a
    // pointer or reference to an object of this class

```

```
public:
    // The constructor sets up the radio interface
    nRF24L01_base (volatile unsigned char&, volatile unsigned char&, unsigned char,
        volatile unsigned char&, volatile unsigned char&, unsigned char,
        base_text_serial* = NULL);

    bool ready_to_send (void);          // Check if the port is ready to transmit
    void reset (void);                 // Reset radio module to starting state

    // Method which sets the address for the transmitter
    void set_TX_address (unsigned long);

    // Method which sets the address for one pipe in the receiver
    void set_RX_address (unsigned long, unsigned char = 0);

    // Method to display the contents of the radio chip's registers
    void dump_regs (base_text_serial*);

    // Set the width of the payload which the receiver will expect
    void set_payload_width (unsigned char, unsigned char = 0);

    // Put the radio into a standard transmission mode
    void set_transmit_mode (void);

    // Put the radio into a standard receiving mode
    void set_receive_mode (void);

    // Method to transmit a packet of data via the radio
    bool transmit (char*);

    // This method transfers an array of bytes to the radio and saves the reply
    bool spi_transfer (unsigned char*, unsigned char);

    // This method puts the radio in "enhanced shockburst" mode with checksums,
    // acknowledgements, automatic retransmission, etc.
    void enhanced_mode (void);

    // This method puts the radio in broadcast mode without acknowledgements
    void broadcast_mode (void);

    // This method checks if there is a packet of data available
    bool data_ready (void);

    // This method just power up the radio module
    void power_up (void);

    // This method shut down the radio module for the lowest power draw
    void power_down (void);
};

#endif // _NRF24L01_BASE_H_
```

```
//*****
/** \file nRF24L01_base.cc
 *   This file contains a class full of utility functions to work with a Nordic
 *   nRF24L01 based radio module.
 *
 *   Revised:
 *   \li 04-22-07 MNL Original file
 *   \li 04-26-07 MNL Added a bunch of functions. Transfer verified working.
 *   \li 05-09-07 MNL Changed so that setup() doesn't have to be called by user
 *   \li 11-15-07 SCH Overhauled to properly control radios and increase speed
 *   \li 01-26-07 SCH Fixed more major flaws
 *   \li 02-15-08 JRR Text based version, written for use on ME405 boards
 *   \li 02-17-08 JRR nRF24L01_base split off from nRF24L01_text
 *   \li 10-11-08 JRR Added control for enhanced/broadcast mode, long addresses
 *   \li 10-11-08 JRR Changed SPI port from bit-banged to SPI hardware
 *   \li 11-06-10 CKC Modified to use in the CBMv1_0 project
 */
//*****

#include <stdint.h>
#include <stdlib.h>
#include <avr/io.h>

#include "global_debug.h"           // Class for serial port debugging
#include "nRF24L01_base.h"

//-----
/** This constructor sets up the nRF24L01 radio for communications. It configures bits
 * in I/O ports which are used for communication with the radio module, and it saves
 * ports and bitmasks so that other functions can talk to the radio later.
 * @param CE_port Port for the Chip Enable (output) line
 * @param CE_ddr Data direction register for the Chip Enable line
 * @param CE_mask Bitmask for the Chip Enable line
 * @param SS_port Port for the SPI Slave Select line
 * @param SS_ddr Data direction register for the SPI Slave Select line
 * @param SS_mask A bitmask for the SPI Slave Select (called CSN by radio) bit
 * @param debug_port A serial port (usually RS232) for debugging text (default NULL)
 */

nRF24L01_base::nRF24L01_base (volatile unsigned char& CE_port, volatile unsigned char&
CE_ddr, unsigned char CE_mask, volatile unsigned char& SS_port, volatile unsigned
char& SS_ddr, unsigned char SS_mask, base_text_serial* debug_port)
{
    port_CE = &CE_port;           // Save CE port
    mask_CE = CE_mask;            // Save CE bitmask
    CE_ddr |= CE_mask;            // Set CE pin as an output
    port_SS = &SS_port;           // Save SS port
    mask_SS = SS_mask;            // Save SS bitmask
    SS_ddr |= SS_mask;            // Set SS pin as an output
    dbg_port = debug_port;        // Save pointer to serial port
}
```

```

in_enhanced_mode = false; // Not in enhanced mode by default

// Enable the correct port pins for the SPI port
#ifdef __AVR_AT90S2313__ // For the AT90S2313 processor
    #error "SPI pins not yet configured for this processor in nRF24L01_base.cc"
#elif defined __AVR_ATmega8__ // For the ATmega8 processor
    #error "SPI pins not yet configured for this processor in nRF24L01_base.cc"
#elif defined __AVR_ATmega8535__ // For the old ATmega8535 processor
    #error "SPI pins not yet configured for this processor in nRF24L01_base.cc"
#elif (defined __AVR_ATmega644__ || defined __AVR_ATmega324P__ \
      || defined __AVR_ATmega32__)
    DDRB &= ~0x40; // Inputs: MISO
    PORTB |= 0x40 | 0x10; // Turn on a pullup and keep SS high
    DDRB |= 0x80 | 0x20 | 0x10; // Outputs: SCK, MOSI, SS
    SPCR = (1 << MSTR) | (1 << SPE); // Enable SPI interface as Master
    SPCR |= (1 << SPR0); // Set clock to f_osc/16, 8 in double-speed
    SPSR = 0x01; // Set SPI interface to double speed
    //why all the above miss a 0 at the back?

#elif defined __AVR_ATmega128__ // For the big ATmega128 processor
    DDRB &= ~0x08;
    PORTB |= 0x08 | 0x01; // Turn on a pullup and keep SS high
    DDRB |= 0x02 | 0x04 | 0x01; // Outputs: SCK, MOSI, SS //
    Inputs: MISO
//     PORTB |= 0x08 | 0x20; // Turn on a pullup and keep SS high
//     DDRB |= 0x02 | 0x04 | 0x20; // Outputs: SCK, MOSI, SS
    SPCR = (1 << MSTR) | (1 << SPE); // Enable SPI interface as Master
    SPCR |= (1 << SPR0); // Set clock to f_osc/16, 8 in double-speed
    SPSR = SPI2X; // Set SPI interface to double speed
#else
    #error "SPI pins not yet configured for this unknown processor in nRF24L01_base.cc"
#endif

reset (); // Set radio module to starting state
}

//-----
/** This method sends a string of bytes to the radio through the SPI port. As the bytes
 *  are transferred to the radio in series, the radio sends back bytes which are stored
 *  in the same string.
 *  @param p_bytes A pointer to the array of bytes to be sent to the radio
 *  @param num_bytes The number of bytes to be sent and received
 *  @return False if the transfer was successful and true if there was a timeout
 */

bool nRF24L01_base::spi_transfer (unsigned char* p_bytes, unsigned char num_bytes)
{
    unsigned char timeout_count;

    // Set the slave select pin for the radio low to get its attention
    *port_SS &= ~mask_SS;

```

```

    // Send each byte from the character array and replace it with what's recieved
    for (unsigned char index = 0; index < num_bytes; index++)
    {
        SPDR = p_bytes[index];
        for (timeout_count = 0; !(SPSR & (1 << SPIF)); timeout_count++)
        {
            if (timeout_count == 255)          // This means a timeout
            {
                *port_SS |= mask_SS;
                return (true);
            }
        }
        p_bytes[index] = SPDR;                // Save what was sent back by the radio
    }

    *port_SS |= mask_SS;                      // De-select the radio's slave select line
    return (false);
}

//-----
/** This method sets the number of bytes of payload the receiver will be expecting.
 * For first versions, we'll be setting this at 32 bytes for all uses.
 * @param bytes The number of bytes to be expected in the payload
 * @param pipe The number of the pipe whose payload width is to be set (default 0)
 */

void nRF24L01_base::set_payload_width (unsigned char bytes, unsigned char pipe)
{
    unsigned char cmd[2];                    // Temporary storage for commands & data

    if (pipe > 6) return;                    // Pipe numbers over 6 aren't legal
    if (bytes > 32) return;                  // Maximum packet size is 32 bytes

    cmd[0] = nRF24_WR_REG | (nRF24_REG_PW_P0 + pipe);
    cmd[1] = bytes;
    spi_transfer (cmd, 2);
}

//-----
/** This method puts the radio in "enhanced shockburst" mode so that checksums and
 * automatic retransmission will be used in an attempt to ensure that data gets to
 * the receiver at the correct address. The other available mode is broadcast mode,
 * in which data is sent out willy-nilly without checking if it was received.
 */

void nRF24L01_base::enhanced_mode (void)
{
    unsigned char cmd[2];                    // Temporary storage for commands/data

```

```
// Enable auto-acknowledgement for data pipes 0 through 6
cmd[0] = nRF24_WR_REG | nRF24_REG_EN_AA;
cmd[1] = 0x3F;
spi_transfer (cmd, 2);

// Turn on automatic retries, waiting 1000 units (about 2ms?) before retrying and
// allowing 5 retries
cmd[0] = nRF24_WR_REG | nRF24_REG_SETUP_RETR;
cmd[1] = 0x85;
spi_transfer (cmd, 2);

// Set the enhanced mode flag so we'll stay in this mode after resetting radio
in_enhanced_mode = true;
}

//-----
/** This method puts the radio in broadcast mode, in which it sends data out to anyone
 * who will listen and doesn't check for acknowledgements and checksums which would
 * indicate that the data was received correctly. The other available mode is
 * "enhanced shockburst" mode, which uses checksums, acknowledgements and automatic
 * retries to try really hard to ensure that the data was correctly received.
 */

void nRF24L01_base::broadcast_mode (void)
{
    unsigned char cmd[2];          // Temporary storage for commands/data

    // Disable auto-acknowledgement for text mode
    cmd[0] = nRF24_WR_REG | nRF24_REG_EN_AA;
    cmd[1] = 0x00;
    spi_transfer (cmd, 2);

    // Turn off automatic retries
    cmd[0] = nRF24_WR_REG | nRF24_REG_SETUP_RETR;
    cmd[1] = 0x00;
    spi_transfer (cmd, 2);

    // Clear the enhanced mode flag so we'll stay in broadcast mode after radio reset
    in_enhanced_mode = false;
}

//-----
/** This method puts the radio into a standard transmission mode. Generally this will
 * be done just long enough to send a packet, then things go back to receive mode, or
 * sometimes it's nap time.
 */

void nRF24L01_base::set_transmit_mode (void)
{
    unsigned char cmd[2];          // Temporary storage for commands & data
```



```

    cmd[0] = nRF24_WR_REG | nRF24_REG_CONF;
    cmd[1] = nRF24_XMIT_MODE;
    spi_transfer (cmd, 2);

    *port_CE &= ~mask_CE;                // Turn the receiver off by dropping CE
}

//-----
/** This method puts the radio into a standard receiving mode. It will wait for
 * packets to arrive and alert the microcontroller using the IRQ line when one has
 * arrived. Other interrupt sources (transmitter empty and too many retries) are
 * masked so they won't cause interrupts.
 */

void nRF24L01_base::set_receive_mode (void)
{
    unsigned char cmd[2];                // Temporary storage for commands & data

    cmd[0] = nRF24_WR_REG | nRF24_REG_CONF;
    cmd[1] = nRF24_RECV_MODE;
    spi_transfer (cmd, 2);

    *port_CE |= mask_CE;                // Receiver needs CE high to be on
}

//-----
/** This method checks if there's data available to be read.
 * @return True if there's data available in the buffer and false if not
 */

bool nRF24L01_base::data_ready (void)
{
    unsigned char cmd[2];                // Temporary space for commands & data

    cmd[0] = nRF24_RD_REG | nRF24_REG_STATUS;
    spi_transfer (cmd, 2);

    if (cmd[0] & nRF24_RX_DR)
        return (true);

    return (false);
}

//-----
/** This method transmits a packet of data using the radio. Data sent to this method
 * should be in an array of 33 unsigned characters; the first element in the array
 * will be discarded (it will be used to hold the Write Payload command for the radio)
 * and the other 32 elements contain data to be sent over the radio.
 */

```

```

* @param buffer A pointer to an array of 33 unsigned chars, the first one expendable
* @return True if the data was successfully sent, false if not
*/

```

```

bool nRF24L01_base::transmit (char* buffer)
{
    unsigned char cmd[2];                // Smaller buffer for commands & configs
    bool success = true;                 // Success flag to be returned by this fn.
    unsigned int timeout = 0;            // Counter for timing out and giving up

    set_transmit_mode ();                // Turn off the receiver for a moment

    // Flush transmitter buffer
    cmd[0] = nRF24_FLUSH_TX;
    cmd[1] = 0x00;
    spi_transfer (cmd, 2);

    // Make sure the previous transmission has finished: send no command, check status
    do {
        cmd[0] = nRF24_NOP;
        spi_transfer (cmd, 1);
    }
    while ((cmd[0] & nRF24_TX_DS) && ++timeout < 1000);

    // Clear the transmission-complete bit
    cmd[0] = nRF24_WR_REG | nRF24_REG_STATUS;
    cmd[1] = nRF24_TX_DS;
    spi_transfer (cmd, 2);

    // Put the data to send in the buffer
    buffer[0] = (char)nRF24_WR_PLD;
    spi_transfer ((unsigned char*)buffer, 33);

    // Pulse the CE line to begin the transmission; the pulse must be >10 us long
    *port_CE |= mask_CE;
    for (volatile unsigned int blah = 0; blah < 30; blah++);
    *port_CE &= ~mask_CE;

    // Check for success -- what happened? Look for acknowledgement of transmission
    for (timeout = 0; ; timeout++)
    {
        cmd[0] = nRF24_RD_REG | nRF24_REG_STATUS;
        cmd[1] = 0x00;
        spi_transfer (cmd, 2);

        if (cmd[1] & nRF24_TX_DS)        // It worked OK! Success remains true
            break;
        if (cmd[1] & nRF24_MAX_RT)        // Used up all the retries (if any), phooey
        {
            success = false;
            break;
        }
    }
}

```

```

        if (timeout > 1000)                // Waited too long for acknowledgement
        {
            success = false;
            break;
        }
    }

    set_receive_mode ();                    // Turn the receiver back on again

    return (success);
}

//-----
/** This method resets the radio. It can be used if the radio gets into an unknown
 * state. It sets the following configuration:
 * \li Only the data received interrupt is used; others are masked off
 * \li Auto-acknowledgement and re-transmission are enabled for all pipes
 * \li Send and receive payload widths are set to 32 bytes
 */

void nRF24L01_base::reset (void)
{
    unsigned char cmd[2];                  // Temporary storage for commands/data

    // Mask off unneeded interrupts and put radio in receive mode
    cmd[0] = nRF24_WR_REG | nRF24_REG_CONF;
    cmd[1] = nRF24_RECV_MODE;
    spi_transfer (cmd, 2);

    // Put the radio in "enhanced shockburst mode" or broadcast mode as needed
    if (in_enhanced_mode)
        enhanced_mode ();
    else
        broadcast_mode ();

    // Set address width to 4 bytes
    cmd[0] = nRF24_WR_REG | nRF24_REG_SETUP_AW;
    cmd[1] = 0x02;
    spi_transfer (cmd, 2);

    // Set RF setup register: Max power, 1MB/s data rate, LNA (Low Noise Amp) gain ??
    cmd[0] = nRF24_WR_REG | nRF24_REG_RF_SETUP;
    cmd[1] = 0x07;
    spi_transfer (cmd, 2);

    // Set receiver payload width for Pipe 0 to 32 bytes
    set_payload_width (32, 0);

    // Activate receiver pipe 0 only
    cmd[0] = nRF24_WR_REG | nRF24_REG_EN_RXADDR;
    cmd[1] = 0x01;

```

```

    spi_transfer (cmd, 2);

    // Clear all the interrupt sources
    cmd[0] = nRF24_WR_REG | nRF24_REG_STATUS;
    cmd[1] = nRF24_TX_DS | nRF24_RX_DR | nRF24_MAX_RT | nRF24_TX_FULL;
    spi_transfer (cmd, 2);

    // Flush the transmit and receive buffers
    cmd[0] = nRF24_FLUSH_TX;
    cmd[1] = 0x00;
    spi_transfer (cmd, 2);

    cmd[0] = nRF24_FLUSH_RX;
    cmd[1] = 0x00;
    spi_transfer (cmd, 2);

    // Leave the CE line high to keep the receiver on
    set_receive_mode ();
}

//-----
/** This method sets the transmitter's address. Four-byte addresses are used because
 * they're more convenient than three or five byte addresses to work with.
 * @param addr The address to be set, as a long integer with 4 bytes in it
 */

void nRF24L01_base::set_TX_address (unsigned long addr)
{
    unsigned char bytes[6];           // Array to be sent via SPI to radio

    // Fill the array with write-register command and the address data
    bytes[0] = nRF24_WR_REG | nRF24_REG_TX_ADDR;
    char* p_addr = (char*)&addr;
    for (unsigned char count = 4; count > 0; count--)
        bytes[count] = *p_addr++;
    bytes[5] = 0x00;

    spi_transfer (bytes, 6);
}

//-----
/** This method sets the transmitter's address. Four-byte addresses are used because
 * they're more convenient than three or five byte addresses to work with. Currently
 * only Pipe 0 is supported.
 * @param addr The address to be set, in a five-byte character array
 * @param pipe The number of the pipe whose address will be set (default 0)
 */

void nRF24L01_base::set_RX_address (unsigned long addr, unsigned char pipe)
{

```

```

    unsigned char bytes[6]; // Array to be sent via SPI to radio

    // Fill the array with write-register command and the address data
    bytes[0] = nRF24_WR_REG | nRF24_REG_RX_ADDR_P0;
    char* p_addr = (char*)&addr;
    for (unsigned char count = 4; count > 0; count--)
        bytes[count] = *p_addr++;
    bytes[5] = 0x00;

    // Send the information over the SPI connection. If sending an address for pipe 0
    // or 1, send a 5-byte number; for pipes 2 - 6, send only one byte
    if (pipe == 0 || pipe == 1)
        spi_transfer (bytes, 6);
    else
        spi_transfer (bytes, 2);
}

//-----
/** This function checks if the radio transmitter is ready to send data. It does so by
 *  sending a NOP (no operation) command to the radio and checking the status byte
 *  which the radio generously sends back in return.
 *  FIXME: This hasn't been well tested
 *  @return True if the radio is ready to send, and false if not
 */

bool nRF24L01_base::ready_to_send (void)
{
    unsigned char cmd = nRF24_NOP;
    spi_transfer (&cmd, 1);
    if (cmd & nRF24_TX_DS)
        return (false);
    return (true);
}

//-----
/** This method supposedly will disable the radio to save power.
 */
void nRF24L01_base::power_down (void)
{
    unsigned char cmd[2];
    // power down, it set all other bits 1 except the pwr_up bit
    cmd[0] = nRF24_WR_REG | nRF24_REG_CONF;
    cmd[1] = ~nRF24_PWR_UP;
    spi_transfer (cmd, 2);
    *port_CE &= ~mask_CE;
}

//-----
/** This method supposedly will enable the radio to warm up the radiosave power.
 */
void nRF24L01_base::power_up (void)

```

```

{
    unsigned char cmd[2];
    // power up, it set all other bits 0 except the pwr_up bit
    cmd[0] = nRF24_WR_REG | nRF24_REG_CONF;
    cmd[1] = nRF24_PWR_UP;
    spi_transfer (cmd, 2);

}

//-----
/** This method displays the contents of the nRF24L01's registers on the given serial
 * device in a convenient, easy to read (for nerds) format.
 * @param p_serial A pointer to the serial port device on which to display registers
 * @param base Base in which results are displayed (bin, oct, dec, hex; default hex)
 */

void nRF24L01_base::dump_regs (base_text_serial* p_serial)
{
    unsigned char reg_data[6];

    GLOB_DEBUG(PMS("Registers in nRF24L01:") << bin << endl);
    /*p_serial << "Registers in nRF24L01:" << bin << endl;

    reg_data[0] = nRF24_REG_CONF;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("00 Config: ") << reg_data[1] << endl);
    /*p_serial << "00 Config: " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_EN_AA;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("01 Auto Ack: ") << reg_data[1] << endl);
    /*p_serial << "01 Auto Ack: " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_EN_RXADDR;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("02 Pipes En: ") << reg_data[1] << endl);
    /*p_serial << "02 Pipes En: " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_SETUP_AW;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("03 Addr Wid: ") << reg_data[1] << endl);
    /*p_serial << "03 Addr Wid: " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_SETUP_RETR;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);

```

```

    GLOB_DEBUG(PMS("04 Retry: ") << reg_data[1] << endl);
    /*p_serial << "04 Retry:  " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_RF_CH;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("05 RF Chan: ") << reg_data[1] << endl);
    /*p_serial << "05 RF Chan:  " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_RF_SETUP;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("06 RF Setup: ") << reg_data[1] << endl);
    /*p_serial << "06 RF Setup: " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_STATUS;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("07 Status: ") << reg_data[1] << endl);
    /*p_serial << "07 Status:  " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_OBS_TX;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("08 TX Errs: ") << reg_data[1] << endl);
    /*p_serial << "08 TX Errs:  " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_CD;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 2);
    GLOB_DEBUG(PMS("09 Carrier: ") << reg_data[1] << endl);
    /*p_serial << "09 Carrier:  " << reg_data[1] << endl;

    reg_data[0] = nRF24_REG_RX_ADDR_P0;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 6);
    GLOB_DEBUG(PMS("0A P0 Addr: ") << hex << reg_data[1] << PMS(".") << reg_data[2] << PMS(
    ".")
        << reg_data[3] << PMS(".") << reg_data[4] << PMS(".") << reg_data[5] << endl <<
        bin);
    /*p_serial << "0A P0 Addr:  " << hex << reg_data[1] << "." << reg_data[2] << "."
    //    << reg_data[3] << "." << reg_data[4] << "." << reg_data[5] << endl << bin;

    reg_data[0] = nRF24_REG_TX_ADDR;
    reg_data[1] = 0x00;
    spi_transfer (reg_data, 6);
    GLOB_DEBUG(PMS("10 TX Addr: ") << hex << reg_data[1] << PMS(".") << reg_data[2] << PMS(
    ".")
        << reg_data[3] << PMS(".") << reg_data[4] << PMS(".") << reg_data[5] << endl <<
        bin);
    /*p_serial << "10 TX Addr:  " << hex << reg_data[1] << "." << reg_data[2] << "."
    //    << reg_data[3] << "." << reg_data[4] << "." << reg_data[5] << endl << bin;

```

```
reg_data[0] = nRF24_REG_PW_P0;
reg_data[1] = 0x00;
spi_transfer (reg_data, 6);
GLOB_DEBUG(PMS("11 P0 Width: ") << reg_data[1] << PMS(" (") << dec << reg_data[1] << PMS(")"))
    << endl << bin);
/*p_serial << "11 P0 Width: " << reg_data[1] << " (" << dec << reg_data[1] << ")"
//    << endl << bin;

reg_data[0] = nRF24_REG_FIFO_STATUS;
reg_data[1] = 0x00;
spi_transfer (reg_data, 6);
GLOB_DEBUG(PMS("17 FIFO St: ") << reg_data[1] << endl);
/*p_serial << "17 FIFO St: " << reg_data[1] << endl;

GLOB_DEBUG(dec << endl);
/*p_serial << dec << endl;
}
```



```
class nRF24L01_text: public nRF24L01_base, public base_text_serial
{
    // Private data and methods are accessible only from within this class and
    // cannot be accessed from outside -- even from descendents of this class
private:

    // Protected data and methods are accessible from this class and its descendents
protected:
    char transmit_buffer[33];          ///< Buffer holds characters to send out
    unsigned char buf_index;           ///< Index of next char. space in buffer

    // Public methods can be called from anywhere in the program where there is a
    // pointer or reference to an object of this class
public:
    // The constructor sets up the radio interface
    nRF24L01_text (volatile unsigned char&, volatile unsigned char&, unsigned char,
        volatile unsigned char&, volatile unsigned char&, unsigned char,
        base_text_serial* = NULL);

    bool putchar (char);               // Write one character to serial port
    void puts (char const*);           // Write a string to serial port
    bool check_for_char (void);        // Check if a character is in the buffer
    char getchar (void);               // Get a character; wait if none is ready
    void transmit_now (void);          // Immediately transmit any buffered data
};

#endif // _NRF24L01_TEXT_H_
```

```

//*****
/** \file nRF24L01_text.cc
 *   This file contains a class which operates a Nordic nRF24L01 based radio module
 *   attached to an AVR processor through the AVR's SPI port in text mode. In this
 *   context, "text mode" means that the radio will send characters and character
 *   strings as one would send strings through a serial line to a terminal, as
 *   opposed to sending packets of binary data with error checking.
 *
 * Revised:
 *   \li 04-22-07 MNL Original file
 *   \li 04-26-07 MNL Added a bunch of functions. Transfer verified working.
 *   \li 05-09-07 MNL Changed so that setup() doesn't have to be called by user
 *   \li 11-15-07 SCH Overhauled to properly control radios and increase speed
 *   \li 01-26-07 SCH Fixed more major flaws
 *   \li 02-15-08 JRR Changed to use hardware SPI port on ME405 boards (not kept)
 *   \li 05-21-08 JRR Added code for second revision of ME405 boards (V0.60)
 *   \li 10-11-08 JRR Added code for the Swoop sensor board with ATmega324PV
 *   \li 11-06-10 CKC Modified to use in the CBMv1_0 project
 */
//*****

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#include "lib/global_debug.h"
#include "nRF24L01_text.h"

/// This define chooses a version of the ME405 board to use. The older V0.5A boards
/// with one motor driver need ME405_BOARD_V05, and the newer V0.60 boards with dual
/// motor drivers are signified by ME405_BOARD_V06. Choose only one of these defines.
#if (!defined SWOOP_BOARD && !defined ME405_BOARD_V05 && !defined ME405_BOARD_V06 && !
defined ETT_ATMEGA128)
    #error You must specify a board (-DSWOOP_BOARD, -DME405_BOARD_V05, ...) in Makefile
#endif

#ifdef SWOOP_BOARD
    #define nRF24_IRQ_vect INT2_vect        ///< The radio RX interrupt vector
    #define nRF24_IRQ_done INTF2           ///< Bit for ack-ing radio interrupts
    #define nRF24_IRQ_cbit ISC21           ///< Bit sets IRQ in falling edge mode
    #define nRF24_IRQ_mask INT2           ///< Mask to enable radio interrupts
    #define nRF24_EICR EICRA              ///< Interrupt mode control register
    #define nRF24_EIMSK EIMSK             ///< External interrupt mask name

#elif defined ME405_BOARD_V05
    #define nRF24_IRQ_vect INT7_vect        ///< The radio RX interrupt vector
    #define nRF24_IRQ_done INTF7           ///< Bit for ack-ing radio interrupts
    #define nRF24_IRQ_cbit ISC71           ///< Bit sets IRQ in falling edge mode
    #define nRF24_IRQ_mask INT7           ///< Mask to enable radio interrupts
    #define nRF24_EICR EICRB              ///< Interrupt mode control register

```

```

#define nRF24_EIMSK      EIMSK          ///< External interrupt mask name

#elif defined ME405_BOARD_V06
    #define nRF24_IRQ_vect INT0_vect    ///< The radio RX interrupt vector
    #define nRF24_IRQ_done INTF0       ///< Bit for ack-ing radio interrupts
    #define nRF24_IRQ_cbit ISC01       ///< Bit sets IRQ in falling edge mode
    #define nRF24_IRQ_mask INT0       ///< Mask to enable radio interrupts
    #define nRF24_EICR      EICRA       ///< Interrupt mode control register
    #define nRF24_EIMSK      EIMSK       ///< External interrupt mask name

#elif defined ETT_ATMEGA128
    #define nRF24_IRQ_vect INT0_vect    ///< The radio RX interrupt vector
    #define nRF24_IRQ_done INTF0       ///< Bit for ack-ing radio interrupts
    #define nRF24_IRQ_cbit ISC01       ///< Bit sets IRQ in falling edge mode
    #define nRF24_IRQ_mask INT0       ///< Mask to enable radio interrupts
    #define nRF24_EICR      EICRA       ///< Interrupt mode control register
    #define nRF24_EIMSK      EIMSK       ///< External interrupt mask name
#endif

//-----
/** This is a file-scope copy of the address of the one radio object which should be
 *  instantiated. It is used by the interrupt service routine to communicate with the
 *  radio object. */

nRF24L01_text* g_p_radio_object;

//-----
/** This circular buffer holds characters received from the radio. The characters can
 *  be read by calls to getchar(). */

queue<unsigned char, unsigned char, 254> g_RX_queue;

//-----
/** This constructor sets up the nRF24L01 radio for communications. It doesn't have to
 *  do much work itself, because the constructors of its ancestors (the base nRF24L01
 *  and base text serial port) do most of the work.
 *
 *  One radio: This system is designed to work with just one radio module in a system,
 *  connected to the hardware SPI port and receiving characters under the control of
 *  one interrupt service routine.
 *  @param CE_port Port for the Chip Enable (output) line
 *  @param CE_ddr Data direction register for the Chip Enable line
 *  @param CE_mask Bitmask for the Chip Enable line
 *  @param SS_port Port for the slave select (called CSN by radio) line
 *  @param SS_ddr Data direction register for slave select (called CSN by radio) line
 *  @param SS_mask Bitmask for the slave select (called CSN by radio) bit
 *  @param debug_port A serial port (usually RS232) for debugging text (default NULL)
 */

```

```

nRF24L01_text::nRF24L01_text (volatile unsigned char& CE_port, volatile unsigned char&
    CE_ddr, unsigned char CE_mask, volatile unsigned char& SS_port, volatile unsigned
    char& SS_ddr, unsigned char SS_mask, base_text_serial* debug_port)
    : nRF24L01_base (CE_port, CE_ddr, CE_mask, SS_port, SS_ddr, SS_mask, debug_port),
      base_text_serial ()
{
    // Enable External Interrupt, connected to the radio's IRQ pin
    nRF24_EICR |= (1 << nRF24_IRQ_cbit);          // Int. on falling edges only
    nRF24_EIMSK |= (1 << nRF24_IRQ_mask);

    // Begin with no characters in the transmission buffer, but leave the zero-th
    // element empty, as a radio command will go there
    buf_index = 1;

    // Save file-scope variable used to access this radio module interface object
    g_p_radio_object = this;
}

//-----
/** This method sends one character to the radio object's transmit buffer. Because
 * characters are usually sent as part of strings and we don't want to send single
 * character packets all the time, the character is just stored in the buffer unless
 * it causes the buffer to reach 31 bytes in length or someone has requested immediate
 * transmission, in which case the buffer is sent.
 * @param chout The character to be sent out
 * @return True if everything was OK and false if there was a timeout
 */

bool nRF24L01_text::putchar (char chout)
{
    // Put the character in the buffer; if it's element 32, it's time to transmit
    transmit_buffer[buf_index++] = chout;

    // If there's a send-it-now code, check if it's time to send it now
    #ifdef nRF_SEND_NOW_CODE
        if (chout == nRF_SEND_NOW_CODE)
            transmit_now ();
    #endif

    // If the buffer is full, send the whole load of stuff
    if (buf_index > 32)
    {
        transmit (transmit_buffer);
        buf_index = 1;
    }
}

//-----
/** This method causes whatever is in the transmission buffer to be put into a packet,
 * padded with zeros if necessary, and sent out immediately.

```

```
*/

void nRF24L01_text::transmit_now (void)
{
    // If there's nothing in the buffer at all, don't bother transmitting
    if (buf_index <= 1) return;

    // Put a null character after the current character; the receiver will ignore
    // the null character and any data following it in the packet
    if (buf_index < 32)
        transmit_buffer[buf_index] = '\\0';

    // Here's where the data is actually transmitted
    transmit (transmit_buffer);
    buf_index = 1;
}

//-----
/** This method writes all the characters in a string until it gets to the '\\0' at
 * the end. If the string is short enough, it's sent in one packet. If it's too long
 * to fit in the current buffer, the string is split up into as many packets as are
 * needed to get the data from here to there.
 * @param str The string to be written
 */

void nRF24L01_text::puts (char const* str)
{
    while (*str) putchar (*str++);
}

//-----
/** This method gets one character from the serial port, if one is there. If not, it
 * waits until there is a character available. This can sometimes take a long time
 * (even forever), so use this function carefully. One should almost always use
 * check_for_char() to ensure that there's data available first.
 * @return The character which was found in the serial port receive buffer
 */

char nRF24L01_text::getchar (void)
{
    if (!(g_RX_queue.is_empty ()))
        return (g_RX_queue.get ());

    return ('\\0');
}

//-----
/** This function checks if there is a character in the serial port's receiver buffer.
 * It returns 1 if there's a character available, and 0 if not.
```

```

* @return True for character available, false for no character available
*/

bool nRF24L01_text::check_for_char (void)
{
    // Check the receiver queue first; if there's something there, we're done
    if (g_RX_queue.is_empty ())
        return (false);

    return (true);
}

//-----
/** This is the interrupt service routine which is called whenever the nRF23L01 radio
 * module drops its interrupt pin low. This event occurs when there's data which has
 * arrived into the receiver. The ISR gets the data from the radio and puts it into
 * the receiving queue. Defines are used to select the appropriate interrupt pin,
 * either INT7 for the older ME405 board (with one motor driver) or INT0 for the newer
 * V0.60 board (with dual motor drivers), or INT2 for the Swoop sensor board.
 */

ISR (nRF24_IRQ_vect)
{
    static unsigned char buffer[33];          // Buffer holds data from radio
    unsigned char index;                     // Index into the buffer
    GLOB_DEBUG (PMS ("R\n\r"));
    // Clear the interrupt flag in the microcontroller -- should be automatically
    // cleared when the ISR is executed
    // EIFR |= (1 << nRF24_IRQ_done);

    // Get data out from the buffer
    buffer[0] = nRF24_RD_PLD;
    for (unsigned char count = 1; count < 33; count++)
        buffer[count] = '\0';
    g_p_radio_object->spi_transfer (buffer, 33);

    // Put data from the buffer into the queue
    for (index = 1; index <= 32; index++)
    {
        g_RX_queue.put (buffer[index]);
        if (buffer[index] == '\0') break;
    }

    // Clear the interrupt source in the nRF24L01 radio
    buffer[0] = nRF24_WR_REG | nRF24_REG_STATUS;
    buffer[1] = nRF24_RX_DR | nRF24_TX_DS | nRF24_MAX_RT; // Shouldn't need DS or RT
    g_p_radio_object->spi_transfer (buffer, 2);

    // Flush the receiver buffer
    buffer[0] = nRF24_FLUSH_RX;
    buffer[1] = 0x00;
}

```

```
g_p_radio_object->spi_transfer (buffer, 2);  
}
```



```

//*****
/** \file nRF24L01_secure.h
 *   This file contains a class which uses a Nordic nRF24L01 based radio module
 *   attached to an AVR processor through the AVR's SPI port in text mode. The
 *   program creates packets of data from data stored in a queue with and adds
 *   a checksum; it also sends the packets and listens for acknowledgment of
 *   data received.
 *
 * Revised:
 *   \li 10-22-08 MLG Original File
 *   \li 10-28-08 MLG Successfull sending packets with checksum
 *   \li 10-29-08 MLG, TG Added confirmation of received messages
 *   \li 12-02-08 MLG, Modified to delete loops and prepare it for use in tasks
 *   \li 11-06-10 CKC Modified to used in the CBMv1_0 project
 *   \li 06-22-11 CKC edited the comments
 */
//*****

/// These defines prevent this file from being included more than once in a *.cc file
#ifndef _NRF24L01_FORMATED_TEXT_H_
#define _NRF24L01_FORMATED_TEXT_H_

#include "nRF24L01_text.h"           // Header for nRF24L01 text radio driver
#include "lib/queue.h"               // Header for the queue template
#include "CBM_v1_0.h"               // Header for the alterable parameters

#define DATA_POINTS      7          // Nuber of data point to send in each packet
#define CONFIRM_TIMEOUT   200        // Number of times to check before timeout

/////*****All the following are all included inside CBM_v1_0.h, if this were to used
/////seperately, please change the following value
// #define QUEUE_SIZE      200        // The size of the queue

//-----
/** This class uses a radio module based on a Nordic nRF24L01 chip in text mode.
 * It is designed to send packets of data of 32 bytes. Each packet will contain 7
 * data point and a checksum. Each data point is a 4 digit hexadecimal representation
 * of an integer.
 */

class nRF24L01_secure
{
    // Private data and methods are accessible only from within this class and
    // cannot be accessed from outside -- even from descendents of this class
private:

    // Protected data and methods are accessible from this class and its descendents
protected:
    int packet_buffer[DATA_POINTS+1];    ///< Buffer holds data to send out
    nRF24L01_text* p_radio;              ///< Pointer to a nRF24L01_text object

```

```

    ///< Pointer to the queue that holds the data
    queue<unsigned int,unsigned int,QUEUE_SIZE>* p_queue;
    unsigned int counter;          ///< Counter for confirm method
    int buffer_index;              ///< Index of where to put nex data point in packet
    int sum;                       ///< Variable to hold the checksum
    char n_char;                   ///< Counter number of characters in the message
    bool temp;                     ///< Variable used in the confirm method
    bool confirm_done;             ///< Variable used in the quick send method

// Public methods can be called from anywhere in the program where there is a
// pointer or reference to an object of this class
public:
    // The constructor sets up the secure mode radio interface
    nRF24L01_secure (nRF24L01_text*,rs232*,queue<unsigned int,unsigned
int,QUEUE_SIZE>*);

    bool make_packet (void);        // Make a packet from the data in the queue
    bool send_one (void);           // Send one element of the packet to the radio
    bool data_available (void);     // Return false if queue is empty, true otherwise
    // Sets the variables used to zero in confirming a message
    void reset_variables (void);
    // Put the values into a packet and calls the quick_send method
    void quick_packet_send (int,int,int,int,int,int,int, int);
    // Send the packet out via radio
    void quick_send (void);
    char confirm (void);            // Confirms if the data packet sent was recieved

};

#endif // _NRF24L01_FORMATED_TEXT_H_

```

```
//*****
/** \file nRF24L01_secure.cc
 *   This file contains a class which uses a Nordic nRF24L01 based radio module
 *   attached to an AVR processor through the AVR's SPI port in text mode. The
 *   program creates packets of data from data stored in a queue with and adds
 *   a checksum; it also sends the packets and listens for acknowledgment of
 *   data received.
 *
 * Revised:
 *   \li 10-22-08 MLG Original File
 *   \li 10-28-08 MLG Successful sending packets with checksum
 *   \li 10-29-08 MLG, TG Added confirmation of received messages
 *   \li 12-02-08 MLG, Modified to delete loops and prepare it for use in tasks
 *   \li 11-06-10 CKC Modified to use in the CBMv1_0 project
 */
//*****

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#include "lib/global_debug.h"
#include "nRF24L01_secure.h"

//-----
/** This constructor saves the pointers passes to it and initializes the variables
 *   used in some of the methods of the class.
 *
 * @param Radio The radio object that will be used to send the data
 * @param Serial The serial port object that will be used to send data to the screen
 * @param Queue The queue object in which the data from the ADC is stored
 */

nRF24L01_secure::nRF24L01_secure (nRF24L01_text* radio, rs232* serial,
    queue<unsigned int,unsigned int,QUEUE_SIZE>* queue)
{
    // Save the input to the function for further reference
    p_radio = radio;
    //p_serial = serial;
    p_queue = queue;//pass the queue that holds the data to the constructor

    counter=0;
    buffer_index = 0;
    sum = 0;
    //r_char = 0;
    n_char = 0;
    temp = true;

    GLOB_DEBUG (PMS ("S Radio setup done \n\r"));
}
```

```

    /**p_serial<<"S Radio set up done\n\r";
    }

//-----
/** This method creates a packet using data in the queue. Each time the method is run
 * one data point is added. When 7 data points have been added a check sum of those
 * seven data point is added to the packet. A packet is completed when 7 data points
 * and a checksum have been added to a packet.
 *
 * @return true if a packet has been completed, false otherwise
 */

bool nRF24L01_secure::make_packet (void)
{
    if (buffer_index < DATA_POINTS)
    {
        if(!(p_queue->is_empty ()))
        {
            packet_buffer[buffer_index] = p_queue->get ();
            sum += packet_buffer[buffer_index];
            buffer_index++;
        }
        return(false);
    }
    else if (buffer_index >= DATA_POINTS)
    {
        packet_buffer[buffer_index] = -sum;
        sum = 0;
        buffer_index = 0;
        return(true);
    }
}

//-----
/** This method sends one data point in the packet trough the radio. The data point
 * is an integeres which is sent in hexadecimal format as four ascii characters.
 *
 * @return true if a complete packet has been sent, false otherwise
 */

bool nRF24L01_secure::send_one (void)
{
    if (buffer_index <= DATA_POINTS)
    {
        *p_radio << hex << packet_buffer[buffer_index];
        buffer_index++;
        return(false);
    }
    else
    {

```

```

        buffer_index = 0;
        return(true);
    }
}

//-----
/** This method make and send a quick packet, it fill not needed space with zero. The
 * data point is an integeres which is sent in hexadecimal format as four ascii characters.
 */

void nRF24L01_secure::quick_packet_send (int data,int status, int num_per_cycle, int
time_per_cycle, int sample_frequency, int num_samples, int w_ref, int std)
{
    packet_buffer[0] = data;
    packet_buffer[1] = status;
    packet_buffer[2] = num_per_cycle;
    packet_buffer[3] = time_per_cycle;
    packet_buffer[4] = sample_frequency;
    packet_buffer[5] = num_samples;
    packet_buffer[6] = w_ref;
    packet_buffer[7] = std;

    quick_send();

    return;
}

//-----
/** This method used by quick_packet_send to send a packet through radio
 */
void nRF24L01_secure::quick_send (void)
{
    for (char i=0;i<8;i++)
    {
        *p_radio << hex << packet_buffer[i];
        GLOB_DEBUG(hex<<packet_buffer[i]<<PMS(" ")<<dec);
        confirm_done=0;
    }

    return;
}

//-----
/** This method is used for checking is there is data available in the queue
 *
 * @return true if the is data in the queue, false otherwise
 */

bool nRF24L01_secure::data_available (void)
{
    return(!(p_queue->is_empty ()));
}

```

```

}

//-----
/** This method resets the variables used in the confirm method to their original
 * values. The method is just used for convenience and not having to type the
 * four lines over and over again in the confirm method.
 */

void nRF24L01_secure::reset_variables (void)
{
    //r_char = 0;
    n_char = 0;
    temp = true;
    counter = 0;
}

//-----
/** This method is used to confirm if the data sent through the radio arrived was
 * received by the receiver. It accomplishes this by waiting for a reply to arrive
 * containing the sum of the values in the packet sent. The message expected is a
 * message containing four zeros and a null.
 *
 * @return 1 = checking radio; 2 = wrong checksum; 3 = correct checksum; 4 = timeout
 */

char nRF24L01_secure::confirm(void)
{
    if (counter < CONFIRM_TIMEOUT)
    {
        if (p_radio->check_for_char ())
        {
            char r_char = p_radio->getchar ();
            temp = temp && (r_char == 48 || r_char == 0);
            n_char++;
            if (temp && n_char <= 3)
            {
                return(1);
            }
            else if (temp && n_char > 3)
            {
                //counter=0;
                reset_variables ();
                return(3);
            }
            else
            {
                //counter=0;
                reset_variables ();
                return(2);
            }
        }
    }
}

```

```
    }  
    else  
    {  
        counter++;  
        return(1);  
    }  
}  
else  
{  
    //counter=0;  
    reset_variables ();  
    return(4);  
}  
}
```

```
//=====
/** \file CBM_Base_Receiver.cc
 *   This file contains a program to test the receiver side of the project
 *
 *   Revisions
 *   \li 11-28-08 MLG Original file created to test the receiver task
 *   \li 10-08-10 CKC Modified for use with the CBM project
 *
 *   License:
 *   This file released under the Lesser GNU Public License, version 2.
 */
//=====

#include <stdlib.h>           // Standard C library
#include <avr/io.h>           // Input-output ports, special registers
#include <avr/interrupt.h>    // Interrupt handling functions

#include "lib/rs232int.h"     // Serial port header
#include "lib/global_debug.h" // Header for serial debugging port
#include "lib/stl_timer.h"    // Task timer and time stamp header
#include "queue.h"            // Queue template header
#include "nRF24L01_text.h"    // Nordic nRF24L01 radio module header
#include "lib/stl_task.h"     // Base class for all task classes
#include "task_receiver.h"    // The task that records the ADC data

//Set up the radio modules pin definition
#ifdef __AVR_ATmega128__
    #define CE_MASK      0x10
    #define CSN_MASK     0x20
#else // Atmega324p, please add other definition if another chip is used
    #define CE_MASK      0x08
    #define CSN_MASK     0x10
#endif

#define HOLD_BUTTON while(PINA&0x80){}
#define TEST_LED_BUTTON PORTA|=0x40;while(PINA&0x80){};PORTA&=0xbf
//-----
/** The main function is the "entry point" of every C++ program, the one which runs first
 * (after standard setup code has finished). For mechatronics programs, main() runs an
 * infinite loop and never exits.
 */

int main ()
{

// DDRA=0b01000000;
// Create a serial port object. It should be hooked up to a dumb terminal program
rs232 the_serial_port (9600, 1);

set_glob_debug_port (&the_serial_port);
GLOB_DEBUG (PMS ("GLOB_DEBUG: set up done!\n\r"));
}
```



```

// Print a greeting message. Lets the user know that the program is running

GLOB_DEBUG (PMS ("\r\nCBM Base Reciever V1.0\n\r"));
//the_serial_port << "\r\n\nCBM Base Reciever V1.0" << endl;

GLOB_DEBUG (PMS ("Software MCU:\t\t"));
//the_serial_port<<"Software MCU:\t\t";
#ifdef __AVR_ATmega128__
    GLOB_DEBUG (PMS ("Atmega128\n\r"));
    //the_serial_port<<"Atmega128\n\r";
#endif

#ifdef __AVR_ATmega324P__
    GLOB_DEBUG (PMS ("Atmega324P\n\r"));
    //the_serial_port<<"Atmega324P\n\r";
#endif
// TEST_LED_BUTTON;
GLOB_DEBUG (PMS ("Crytal frequency:\t")<<F_CPU<<PMS(" Hz\n\r"));
//the_serial_port<<"Crytal frequency:\t"<<F_CPU<<" Hz\n\r";

// Create a microsecond-resolution timer
task_timer the_timer;

// Create a nRF24L01 radio text object for the SWOOP board
nRF24L01_text my_radio (PORTB, DDRB, CE_MASK, PORTB, DDRB, CSN_MASK, &the_serial_port);
    my_radio.enhanced_mode ();
    my_radio.reset ();

// Set the radio to receive signals on channel 0x42 using Pipe 0
my_radio.set_RX_address (0x5A697840L);
my_radio.set_TX_address (0x5A697840L);

// Create a time stamp which holds the interval between runs of the receiver task
// The time stamp is initialized with a number of seconds, then microseconds
time_stamp interval_time (0, 1000);

GLOB_DEBUG (PMS ("Receiver Task Interval: ")<< interval_time << PMS("\n\r"));
task_receiver receive_task (the_timer, interval_time, &my_radio, &the_serial_port);

// Turn on interrupt processing so the timer can work
sei ();

// Run the main scheduling loop, in which the tasks are continuously scheduled.
while (true)
{
    receive_task.schedule ();
}

return (0);
}

```

```

//*****
/** \file task_receiver.h
 *   This file contains a task class which controls the receiver unit. It gets
 *   data, checks if its correct, send replies to the transmitter and passes the
 *   data to matlab to update the GUI if correct data was received.
 *
 *   Revisions:
 *   \li 11-29-08 MLG Original file
 *   \li 10-8-10 CKC Modified for CBM project
 *
 *   License:
 *   This file released under the Lesser GNU Public License, version 2.
 */
//*****

```

```

class task_receiver : public stl_task
{
protected:
    nRF24L01_text* p_radio; ///< Pointer to a secure radio communication object
    base_text_serial* p_serial; ///< Pointer to a serial port for messages

    char packet[32];          ///< Array used to store a packet of data
    int values[8];            ///< Array used to store values from a packet
    int sum;                  ///< used to store the sum of the values in a packet
    char index_1;             ///< Used as an index for packet array
    char index_2;             ///< used as an index for values array

public:
    // The constructor creates a new task object
    task_receiver (time_stamp*, nRF24L01_text*, base_text_serial* = NULL);
    task_receiver (task_timer&, const time_stamp&, nRF24L01_text*, base_text_serial* =
NULL);

    // The run method is where the task actually performs its function
    char run (char);

    int to_int (char*); //conver 4 elements hexadecimal to integer
    //int checksum (char*); //computes the sum of the integers in the array

};

```

```

//*****
/** \file task_receiver.cc
 *   This file contains a task class which controls the receiver unit. It gets
 *   data, checks if its correct, send replies to the transmitter and passes the
 *   data to matlab to update the GUI if correct data was received.
 *
 *   Revisions:
 *   \li 11-29-08 MLG Original file
 *   \li 10-8-10 CKC Modified for CBM project
 *
 *   License:
 *   This file released under the Lesser GNU Public License, version 2.
 *
 *   Note:
 *   I have to unplug the programming cable to get correct data from the radio chip
 */
//*****

#include <stdlib.h>
#include <avr/io.h>
#include "lib/rs232int.h"
#include "lib/global_debug.h"
#include "lib/stl_timer.h"
#include "lib/stl_task.h"
#include "nRF24L01_text.h"
#include "task_receiver.h"

const char PACKET_SIZE = 32;    // Number of characters in a packet
const char VALUES_SIZE = 8;    // Number of values in a packet
const char GET_DATA = 0;        // State: check radio for data available and store it
const char CHECK_DATA = 1;      // State: convert data to integers and compute sum
const char UPDATE_GUI = 2;      // State: Send data to matlab to update the GUI

#define TEST_LED_BUTTON PORTA|=0x40;while(PINA&0x80){};while(PINA&0x80){};PORTA&=0xbf
//-----
/** This constructor creates a receiver task object. The communication object needs
 * pointers to a nRF24L01_text object in order to do its thing, and a serial
 * port pointer is handy for debugging.
 * @param t_stamp A timestamp which contains the time between runs of this task
 * @param p_r_text A pointer to the a text based radio communication object
 * @param p_ser A pointer to a serial port for sending messages (default NULL)
 */

//task_receiver::task_receiver (time_stamp* t_stamp, nRF24L01_text* p_r_text,
//    base_text_serial* p_ser) : stl_task (*t_stamp, p_ser)
task_receiver::task_receiver (task_timer& the_time,const time_stamp& interval_time,
nRF24L01_text* p_r_text, base_text_serial* p_ser)
    : stl_task (the_time, interval_time)

//
task_timer&, const time_stamp&

```

```

{
    p_radio = p_r_text;           // Save pointers to other objects
    //p_serial = p_ser;
    index_1 = 0;                  // Initialize variables
    index_2 = 0;
    sum = 0;
    GLOB_DEBUG(PMS("\n\rTask Receiver setup done\n\r"));
}

```

```

//-----
/** This is the function which runs when it is called by the task scheduler. It stores
 * data coming through the radio, convert the characters to integers, sends a reply
 * to the transmitter and updates the GUI if necessary.
 * @param state The state of the task when this run method begins running
 * @return The state to which the task will transition, or STL_NO_TRANSITION if no
 *         transition is called for at this time
 */

```

```

char task_receiver::run (char state)
{
    switch (state)
    {
        // In State 0, check if data has arrived and store it in an array
        case (GET_DATA):
            if (index_1 < PACKET_SIZE)
            {
                if (p_radio->check_for_char ())
                {
                    packet[index_1] = p_radio->getchar ();
                    index_1++;
                }
                return (STL_NO_TRANSITION);
            }
            else
            {
                index_1 = 0;
                return (CHECK_DATA);
            }

            // In State 1, convert data into integers, compute sum and send sum to radio
            case (CHECK_DATA):
                if (index_2 < VALUES_SIZE)
                {
                    values[index_2] = to_int(packet + index_2*4);
                    GLOB_DEBUG(values[index_2]<<PMS(" "));
                    sum += values[index_2];
                    index_2++;
                    return (STL_NO_TRANSITION);
                }
                else

```

```

    {
        *p_radio << hex << sum;
        p_radio->transmit_now ();
        index_2 = 0;
        if (sum == 0)
        {
            GLOB_DEBUG(PMS("\n\rAlright!!!\n\r"));
            return (UPDATE_GUI);
        }
        else
        {
            sum = 0;
            GLOB_DEBUG(PMS("\n\rOh,too bad...\n\r"));
            return (GET_DATA);
        }
    }

    // In State 2, send data through serial port if data arrived uncorrupted
    case (UPDATE_GUI):
        if (index_1 < PACKET_SIZE)
        {
            GLOB_DEBUG(PMS(" ")<<packet[index_1]<<PMS(" "));
            //p_serial->putchar (packet[index_1]);
            index_1++;
            return (STL_NO_TRANSITION);
        }
        else
        {
            GLOB_DEBUG(PMS("END\n\r"));
            index_1 = 0;
            return(GET_DATA);
        }

    // If the state isn't a known state, call Houston; we have a problem
    default:
        GLOB_DEBUG (PMS ("WARNING:task receiver in state ") << state << endl);
        return (GET_DATA);
};

    // We should never get here, but put a return here to keep the compiler happy
    return (STL_NO_TRANSITION);
}

```

```

//-----
/** This method take a pointer to an array of four characters representing an integer
 * in hexadecimal form. It converts the 4 characters into an int in base ten.
 * @param an_array A reference to an array of 4 characters
 * @return the integer representation of the 4 character in the array
 */

```

```

int task_receiver::to_int (char* an_array)

```

```
{
    int result;
    int mult[4] = {4096,256,16,1};
    //static int result;
    //static int mult[4] = {4096,256,16,1};
    result = 0;
    for (char i = 0; i < 4; i++)
    {
        //GLOB_DEBUG(*(an_array+i));
        if (*(an_array+i) < 58)
        {
            result += (int)(*(an_array+i)-48)*mult[i];
            //GLOB_DEBUG(result<<endl);
        }
        else
        {
            result += (int)(*(an_array+i)-55)*mult[i];
            //GLOB_DEBUG(result<<endl);
        }
    }
    return(result);
}
```