Project Number: RAP 4019

An Activity Monitor for Diabetic Individuals

A Major Qualifying Project Report:

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Vinith Chemmalil

Marissa Gray

Jennifer Keating

Rebecca Kieselbach

Sarah Latta

Submitted: May 3, 2009

Approved:

Prof. Robert Peura, Major Advisor

1. diabetes 4.ECG
2. activity monitoring 5. accelerometer
3. energy expenditure

Abstract

Continuous monitoring of physical activity for diabetic individuals provides important information regarding energy expenditure. This device makes use of both heart rate sensing and full body acceleration to measure physical activity. Energy expenditure is calculated using an algorithm which determines the amount of activity occurring. The output of the calculation is displayed in carbohydrates to allow for the patient to make better informed decisions regarding blood glucose level adjustment.

Executive Summary

Diabetes is a disease characterized by the body's inability to produce or properly utilize insulin, the hormone which allows for the cellular absorption and metabolism of glucose [1]. As a result of this inability to maintain stable glucose levels, diabetic individuals must be aware of their glucose levels as related to their daily activities.

The largest dietary concern for an insulin dependent (Type I) diabetic individual is carbohydrates, as they break down into glucose. To ensure an even distribution of carbohydrate ingestion, a Type I diabetic individual makes adjustments to their insulin dosage based on the carbohydrate content of a meal. Similarly, a diabetic individual needs to compensate for carbohydrates burned during exercise through both reduced insulin dosages and increased carbohydrate consumption.

Activity monitoring devices available today provide valuable information, however they are not tailored to calculating energy expenditure in real-time for diabetic individuals. When used in combination, heart rate and accelerometry are two valid parameters for determining physical activity [2, 3]. A device that uses these parameters to calculate the user's carbohydrate usage could be used as a noninvasive approach to blood-glucose management.

This project developed an activity monitor which utilizes an algorithm to determine energy expenditure through the measurements of heart rate and full body acceleration. As opposed to Actiheart, a comparable device which displays caloric usage retroactively, the device outputs energy expenditure in terms of carbohydrates in real-time [3].

The first activity monitor device for diabetic individuals was designed by an Major Qualifying Project (MQP) team in 2007-2008. They were able to confirm an appropriate system for obtaining accurate heart rate and tri-axial full-body acceleration. However, the device had many design issues that made it impractical for daily use. The goal of this MQP was to improve the original design.

Much like the previously designed proof-of-concept device, the newly designed device is composed of two parts – the signal acquisition module, and the display module.

The electrode based ECG system from the 2007-2008 project was replaced with a Polar system in order to make the device more wearable. This system is composed of a Polar strap with a built-in transmitter and a receiver. The transmitter is worn around the chest and electrically detects the heart beat. It transmits a magnetic signal for each heart beat it detects.

The receiver wirelessly detects this signal, and upon arrival of a pulse, outputs a 1 millisecond digital pulse. The difference in time between the two most recent pulses is used to calculate an instantaneous heartrate. Wireless communication between the transmitter and receiver is completed in a low frequency electromagnetic field which must be aligned in parallel in order to obtain optimal performance.

In the current design, the accelerometer and filter circuitry is encased in the Polar package. The accelerometer outputs three voltages for the x, y, and z axes; these signals fluctuate around a zero-g offset (a fixed voltage representing no acceleration in the direction of the axes). When the three axial outputs are summed, the result includes only voltage outputs from the accelerometer due to movement. The zero-g voltage, which results from no movement, and therefore no energy expenditure, must be eliminated from the axial outputs before they are summed.

Digital filtering of the three axial accelerometer outputs is performed using a Quickfilter chip. The output of the digital filter is rectified, summed, and counted by software implemented on a microcontroller. The accelerometer count is then wirelessly transmitted to the display module via a pair of ZigBee devices.

The best method to calculate energy expenditure from heart rate and accelerometer counts was developed by Actiheart [25]. Using a branched equation model, the Actiheart algorithm uses parameters such as the subjects' age and gender and variable parameters such as heart rate and accelerometer counts to determine kilocalories per kilogram minute that are burned.

To tailor the device for individuals with diabetes, a process is needed to convert kilocalories to carbohydrates. The process involves determining the amount of carbohydrates burned during certain activities. The highest amount of carbohydrates is burned during anaerobic exercise, while the most fat is burned during light aerobic exercise. In the absence of oxygen, muscles burn carbohydrates supplied from glycogen, which breaks down into glucose. In the presence of excess oxygen, muscles will use fat for energy. The process works similar to the Actiheart algorithm in that the ratio of carbohydrates to fat being metabolized varies with accelerometer counts. A modification has been made to the Actiheart algorithm which converts energy expenditure in terms of caloric output into a carbohydrate expenditure via ratios based on an individual's weight. This algorithm will henceforth be referred to as the modified Actiheart algorithm.

The display module consists of a microcontroller with an LCD display and four input buttons. The microcontroller is responsible for both signal analysis and user interface, and works in two separate states: parameter input state and analysis state.

The device enters the parameter input state as soon as it is turned on. Here, the user is prompted to enter first their gender (1 for male, 0 for female), then their age, and finally their weight. Numerical values are inputted on a button-to-button basis in which each placeholder button scrolls through the values available for that place for each input. For example, during gender input only the fourth button may change, and it may only be 0 or 1. This is to ensure that it is easy for the user to correct mistakes. The values selected by the user are entered by pressing the first button. The data entered during the parameter input state is stored and sent to the modified Actiheart algorithm for use in the analysis state.

The device then enters the analysis state. During the analysis state the chip continuously receives signals from the accelerometer and heart rate monitor, converts them into counts per unit time, and displays the result. Results are displayed in a five-second cyclic manner. In this state, the cumulative carbohydrate value calculated using the parameter values entered is available to the user on call via the second button. To reset the carbohydrate count to zero the user may press the first button.

Approval for human subject testing was received through the WPI Institutional Review Board (IRB). This process included creating a study protocol, a case report form, and a consent form. Before obtaining final approval the group participated in online training through the National Institutes of Health (NIH). Once approved the group was able to test the device on students attending WPI on a volunteer basis.

In order to ensure that our device performs as well, if not better than, the previous device, we will be calibrating our device using the algorithms and tests from the first study period. These tests include simulating everyday activities such as washing dishes and climbing stairs, in addition to walking/running on a treadmill at three speeds. Each activity will be performed for five minutes to allow for heart rate and accelerometer stabilization and readings will be taken for the last minute. These results are forthcoming.

Table of Contents

Table of Figures

Table of Tables

Authorship

Abstract	Vinith Chemmalil, Marissa Gray, Jennifer Keating, Rebecca Kieselbach, Sarah Latta
Executive Summary	Jennifer Keating
1 Introduction	Sarah Latta
2 Literature Review	Vinith Chemmalil, Marissa Gray, Jennifer Keating, Rebecca Kieselbach, Sarah Latta
3 Design Methods	Vinith Chemmalil, Marissa Gray, Jennifer Keating, Rebecca Kieselbach, Sarah Latta
4 Detailed Design	Jennifer Keating, Rebecca Kieselbach, Vinith Chemmalil
5 Testing	Sarah Latta, Vinith Chemmalil
6 Conclusions	Marissa Gray
Appendices	
Weighted Objectives	Jennifer Keating
IRB Application	Sarah Latta
Signal Acquisition Code	Rebecca Kieselbach
Display Module Code	Jennifer Keating

1 Introduction

Insulin is the hormone responsible for the cellular uptake of carbohydrates and sugar within the body. It is produced within the pancreas in response to the presence of glucose in the blood [5]. When blood glucose levels increase, the levels of insulin produced increase respectively. The American Diabetes Association (ADA) defines diabetes as "a disease in which the body does not produce or properly use insulin," [4]. In a diabetic individual, the body does not produce enough insulin, or does not recognize its presence and thus does not properly use it.

Currently there are 23.6 million Americans with diabetes, which composes approximately 7.8% of the population [4]. There are three types of diabetes: gestational, Type 1, and Type 2. Gestational diabetes occurs in pregnant women when the hormones required to create the placenta also block insulin reception. This is called insulin resistance and occurs in 4% of pregnant women annually and normally disappears after the baby is born. Thus, this project is not focused on cases of gestational diabetes, but rather Type 1 and Type 2 diabetes. In Type 1 diabetes, formerly known as juvenile diabetes, the body hosts an immune response against its own, insulin producing islet cells, resulting in the inability to produce insulin. Clinically, diabetes is diagnosed through a fasting plasma glucose test (FPG). To perform an FPG test the individual fasts for at least 12 hours, then blood is drawn and glucose levels are analyzed. If FPG test levels are higher than 126 mg/dl, the patient is diagnosed with diabetes. Levels between 100 and 125 mg/dl indicate pre-diabetes, a condition which is normally considered a precursor to Type 2 diabetes.

In order to control blood sugar levels, an individual with Type 1 diabetes must be injected with insulin several times per day. In Type 2 diabetes, the body either does not produce enough insulin or the cells do not recognize the insulin when it is present [4]. Type 2 diabetes is the most common form of diabetes in the United States and is often associated with obesity and poor nutritional habits.

When blood glucose levels are not properly maintained, both Type 1 and Type 2 diabetes pose serious health risks. Hypoglycemia (low blood sugar) can cause a person to lose consciousness. Hyperglycemia (high blood sugar) can lead to ketoacidosis. Ketoacidosis occurs when the body no longer has insulin to break down glucose and begins to metabolize fats for energy. This releases molecules called ketones that in high doses will poison the body. Under normal circumstances the kidneys can filter out excess ketones, however, in this case the concentration of ketones is too high, and the body begins to shut down and enters a stage called diabetic coma. Higher than normal blood glucose levels, in addition to high ketone concentrations, cause the kidneys to work harder and can cause kidney disease or eventually failure.

In addition to these complications, diabetic individuals are 40% more likely to develop glaucoma and 60% more likely to develop cataracts [4]. Diabetic retinopathy is another common problem caused by the swelling of capillaries in the retina which may result in eye damage and in some cases total loss of vision. Nerve damage known as diabetic neuropathy can lead to double vision, paralysis on one side of the face, bladder control problems, nausea, and dizziness. Diabetes can also lead to blood vessels hardening and narrowing, especially in the foot and leg where it causes several severe complications, often requiring amputation. All of these conditions can be prevented or their symptoms reduced by properly maintaining blood glucose levels.

Several methods are recommended by the American Diabetes Association (ADA) to control blood glucose levels. The gold standard of diabetic monitoring is frequent blood testing to determine blood glucose levels. Also suggested is a method known as carbohydrate counting. Carbohydrates are complex organic molecules that the body breaks down into its subcomponents – glucose. The higher the carbohydrate content in the food, the more the blood glucose level will be affected during digestion. Carbohydrates are used by the body during exercise, especially anaerobic exercise. This causes a dramatic drop in blood sugar after a long period of time. To correct this, an individual may eat something sweet, such as fruit or a candy bar, causing blood sugar to rise rapidly.

This see-saw effect can be reduced by using a method developed in 1995 by Dr. Lois Jovanovic-Peterson called Extra Carbs for Exercise (ExCarbs) [6]. This method predicts the number of carbohydrates that will be utilized during exercise, which can be used to adjust nutritional intake and/or insulin dosages. Dr. Jovanovic-Peterson provides a table that lists these values, however the values are not tailored toward individuals. Values are only given for three weights, and not all exercises are included. For example, if an individual has to walk up seven flights of stairs because the elevators are not working, it would certainly cause a drop in blood sugar; however, such exercises are not included in the table, nor are they predictable. For these reasons and others, it is obvious that better methods of determining how many carbohydrates are used in a given time period are needed.

Developing an improved method of estimated carbohydrate expenditure was the goal of this project. Strides toward this goal were made during the 2007 – 2008 academic year, in which an activity monitor utilizing ECG electrodes and an accelerometer was used to detect heart rate and accelerometry values to estimate energy expenditure. However, placing ECG electrodes correctly is not an easy task for non-healthcare professionals. The device created was heavy and bulky, and did not display how many carbohydrates were used at any point in time. The current project intended to address and resolve these issues. First, the ECG electrodes to determine heart rate were replaced with a Polar transmitter strap. This strap is easily placed around the rib cage by anyone without the need of any special training. Attached to it is the accelerometry unit, which calculates the full body acceleration from the x, y, and z axes. To reduce size and weight the analog circuitry was replaced with digital circuitry. Data from the Polar strap and the accelerometer was wirelessly transmitted to the receiver. Since wires were not necessary to connect the electrodes to a processing unit, the receiver was made to be hand held and portable enough to clip on a belt, or place in a pocket. The receiver displays the individual's carbohydrate expenditure in real-time by utilizing a branched equation algorithm that modeled expenditure rate according to activity level.

The research, development, and testing of this device is detailed in the pages below. Though there are many activity monitors on the market, it is the hope of the design team that this one, as it is specifically tailored to pertain to diabetes, will improve quality of life for those individuals with diabetes.

2 Literature Review

Diabetes is a disease in which the body either does not produce or properly utilize insulin – the hormone which allows the body to convert sugars, starches, and other foods into energy [1]. This results in an inability to inability to monitor and maintain stable blood glucose levels. As a result diabetic individuals must be constantly aware of their blood glucose levels in order to avoid health complications. There are currently 23.6 million people living with diabetes in the United States, thus the design and implementation of a diabetic-specific monitoring device may greatly benefit society as a whole [4]. The following sections discuss blood sugar and the body's response to blood sugar level changes, the special dietary and exercise needs of diabetic persons, the effects of poor glucose management, and how an activity monitor customized to meet the needs of a diabetic person would enable better control of blood glucose levels.

2.1 Understanding Blood Sugar Levels

Blood sugar levels are a measurement of the glucose volume found in blood [12]. Maintaining an appropriate glucose volume is an important factor in homeostatic balance. Detailed below are the role of carbohydrates in blood sugar levels, the glucose response, hormonal response to glucose, and the definition of diabetes as related to these subjects.

2.1.1 The Role of Carbohydrates

Carbohydrates are an essential part of every diet [7]. They provide direct energy for the brain, central nervous system, and muscle cells in the form of glucose [8]. Depending on their origin, they are broken down into simple or complex (fast or slow) carbohydrates during digestion. Simple carbohydrates include sugars such as fruit sugar, corn or grape sugar, and table sugar, whereas complex carbohydrates include foods made of three or more linked sugars, like wholegrain breads, oats, muesli, and brown rice.

Simple carbohydrates, or monosaccharides/disaccharides, are organic compounds and sugars which are easily broken down by digestion [8]. These include galactose, fructose, maltose, sucrose, lactose, and glucose. Complex carbohydrates, or polysaccharides, must be broken down into monosaccharides before they can be released into the blood stream. All sugars are absorbed into the blood stream, with glucose playing a major role in what is called the "blood sugar level".

2.1.2 The Glucose Response

Glucose is a major source of energy for the human body as it is the primary fuel for most cells [9, 10]. Glucose is absorbed into the bloodstream from the gastrointestinal tract after digestion, where it is transported to cells that require energy. Immediate use of glucose following the breakdown of carbohydrates involves the burning of molecules within mitochondria for the release of carbon dioxide, water, and energy into the body [11]. If glucose is not immediately needed, it is converted by the liver or muscles into glycogen in order to supply muscles and other parts of the body with energy. Any remaining glucose after glycogen saturation has occurred may be converted into fat by the liver and stored in adipose tissue around the body. In a healthy individual, these conversion processes allow for blood glucose levels to remain at a relatively homeostatic balance.

2.1.3 Hormonal Responses to Glucose

The body's regulatory system maintains control over the level of glucose found within the bloodstream through hormone regulation systems [12]. There are two types of mutually antagonistic metabolic hormones which affect blood glucose levels. These are catabolic hormones (most commonly glucagon) which increase blood glucose, and an anabolic hormone (insulin) which decreases blood glucose.

When the level of glucose, or blood sugar, in the bloodstream is too high, insulin is released by the pancreas, which notifies fat and muscle cells to absorb glucose and thus lower the blood sugar level [12]. When blood sugar levels are too low, the hormone glucagon is released from the pancreas, which signals the liver to break down glycogen and release more glucose into the blood stream, thus raising blood glucose levels back to normal. In this way, insulin secretion operates on a negative feedback mechanism. Insulin levels rise when glucose is absorbed from the gastrointestinal tract, and drop to normal levels during the presence of glucagon. Insulin levels in a healthy person therefore fluctuate all day long in order to maintain a stable blood sugar level.

2.1.4 Defining Diabetes

Diabetes is a carbohydrate metabolism disorder in which the body is unable to properly maintain blood glucose levels. The normal range for blood glucose is defined as 70-125 mg/dL of blood (less than 7.0 mmol/L). Levels above or below this value result in hyperglycemia (persistently high levels), or hypoglycemia (persistently low levels), respectively [13, 14].

Diabetic individuals must closely monitor their blood glucose levels in order to avoid these conditions.

There are two main types of diabetes – insulin dependent/juvenile diabetes, now termed "Type 1", and non-insulin dependent/adult-onset diabetes, now termed "Type 2" [7]. In cases of Type 1 diabetes the body does not produce enough insulin for cells to absorb glucose at the appropriate rate. This form of diabetes is caused by the destruction of the beta cells in the pancreas by immune mechanisms, which results in little to no insulin secretion by the pancreas [15, 16].

With Type 2 diabetes a much different problem arises. The cells in the body exhibit insulin resistance, which causes blood sugar and insulin levels to stay high after eating for much longer than they should. In these situations insulin-making cells are weakened over time, and eventually the production of insulin stops. Insulin resistance has been linked with issues such as high blood pressure, high levels of triglycerides, low HDL cholesterol, and excess weight. Several things can promote insulin resistance such as a sedentary lifestyle, being overweight, and a diet rich in processed carbohydrates.

While Type 2 diabetes can be treated by other things such as a proper diet and exercise, all Type 1 diabetic individuals must be treated with daily insulin and must monitor their diets and exercise levels consistently [7]. Patients with Type 1 diabetes administer insulin shots two to four times per day and ensure an even distribution of ingested carbohydrates throughout the day. They also measure their blood sugar levels to ensure that they remain within acceptable range throughout the day.

2.2 Nutrition

As diabetes is a metabolic disease, it is nearly impossible to control it with medicine alone – an entire lifestyle change is needed [4]. Diabetic individuals need to be concerned with what they are eating, how often they are eating, their weight, and how often they exercise. The American Diabetes Association (ADA) has put together a comprehensive website outlining these topics and providing tools for diabetic individuals to more easily make the lifestyle changes necessary for their health. When a diabetic individual takes proper care of him or herself, they are able to lead a normal, long life. Unfortunately, many diabetic individuals do not monitor their lifestyle enough and often fatal complications arise.

A large part of a diabetic individual's dietary concerns are carbohydrates, as carbohydrates have the greatest effect on glucose levels [4]. However, there are different kinds of carbohydrates and each has different effects on glucose levels. The three types of carbohydrates are sugar, starch, and dietary fiber. Of the three, dietary fiber effects glucose levels the least. The effects of sugars and starches have been carefully researched and organized for reference by diabetic individuals in something called the glycemic index.

2.2.1 The Glycemic Index

The glycemic index (GI) ranks how a food will affect blood glucose [4]. There are a few factors involved in determining a food's GI, these include: ripeness, processing, cooking, and variety of the food. For example, a banana that is still slightly green will have a lower GI than a spotted banana. Converted long-grain rice has a lower GI than brown rice, but short-grain white rice has a higher GI. Juice has a higher GI than the fruit itself. The lower the GI, the less of an effect on blood glucose the item will have. In general, diabetic individuals should eat more unprocessed foods and less high GI items such as white bread, pineapples, and instant oatmeal [4]. The list below categorizes several foods as low, medium, or high GI foods.

Low GI

- Stone Ground Wheat Bread, Pumpernickel Bread
- Pasta
- Sweet Potatoes, corn, peas, carrots
- Most fruits

Medium GI

- Whole Wheat Bread, Rye Bread
- Quick Oats
- Brown Rice, Couscous

High GI

- White Bread
- Corn Flakes, Instant Oatmeal
- Popcorn
- Pineapple, melons

Figure 1: Hierarchy of some foods in their respective glycemic index categories. Foods with a lower glycemic index will impact blood sugar levels less than foods with a high glycemic index [4].

2.2.2 American Diabetes Association Food Pyramid

While focusing on carbohydrates, it is easy to forget other portions of a meal, but both fat and fiber are needed to slow the rise of blood glucose levels [4]. To account for this, the ADA suggests eating peanut butter with apples and including low-fat protein sources, such as fish or chicken. The key to diabetes management is keeping a well-balanced diet, full of variety.

The ADA has created a food pyramid to further help diabetic individuals plan their meals [4]. This pyramid is different than the standard United States Department of Agriculture (USDA) pyramid in that the ADA pyramid is based on carbohydrate count, rather than just food classifications. For example, in the USDA pyramid, potatoes are counted as a vegetable, whereas in the ADA pyramid potatoes are counted as starches because of their high carbohydrate content. Other than some items that are in different categories, like potatoes, the pyramid is essentially the same. It is divided into six categories – Starches, Fruits, Vegetables, Proteins, Milk, and Oils and Sweets – and contains recommended servings for each category.

Starches include bread, bagels, dry cereal, cooked cereal, potatoes, yams, peas, corn, cooked beans, winter squash, rice, and pasta [4]. The pyramid suggests 6-11 servings of starch a day. Most people do not actually eat eleven servings. Most grain and starches contain 15g of carbohydrates per ½ cup. The ADA suggests eating certain foods over others. For example, diabetic individuals should eat whole grains rather than processed white flour. To this extent the ADA has put out a list of ‘best’ foods for the starch category.

Non-starchy vegetables include spinach, Swiss chard, broccoli, cabbage, brussel sprouts, cauliflower, kale, carrots, tomatoes, cucumbers, and lettuce [4]. Three to five servings are recommended daily, however, a serving varies if the vegetable is cooked. One cup of raw tomatoes is a serving, but only ½ cup of stewed tomatoes is a serving. When shopping for vegetables diabetic individuals should look for fresh, frozen, or canned veggies without any added salt or sauce.

Fruits have approximately the same number of carbohydrates as starches, but generally contain more vitamins and lower GI values [4]. High fructose and fiber contents account for the lower GI value of fruits. Approximately ½ cup of canned fruit or juice has 15g of carbohydrates, the same as ½ cup of starches. For berries, a serving is about 1 cup. Diabetic individuals should watch portion sizes with dried fruit since a serving is only 2 tablespoons.

Best Grains

- Bulgur
- Whole Wheat flour
- Whole oats
- Whole grain corn
- Popcorn
- Brown rice
- Whole rye
- Wild rice
- Buckwheat
- Millet
- Quinoa
- Sorghum

Best Starchy Vegetables

- Parsnip
- Plantain
- Potato
- Pumpkin
- Acorn Squash
- Butternut Squash
- Green Peas
- Corn

Best Dried Beans

- Black, Lima, and Pinto beans
- Lentils
- Black-eyed and split peas
- Fat-free refried beans
- Vegetarian baked beans

Figure 2: This is a listing of some of the best starchy foods for diabetic individuals. This list has been designed based on ADA food classifications to make it easier for a diabetic individual to choose the bulk of meal wisely.

According to the pyramid, two to three servings of dairy products is needed a day. Fat-free or low fat milk, non-fat light yogurt, and unflavored soy milk are the best choices. The traditional USDA pyramid includes cheese and eggs in the dairy category, however, due to their high protein content, they are included with meat. In addition to eggs, the meat category includes all high-protein foods such as beans and all meat. Four to six ounces of meat spread out throughout the day is recommended. The best choices for protein include dried beans, fish, skinless chicken, eggs, and beef with all visible fat removed. It is important to keep in mind that though meat and fish do not contain carbohydrates, dried beans do [4]. Fats, sweets, and alcohol do not contain much nutrition and should be reserved for special occasions.

Utilizing this pyramid, the recommended servings and serving sizes in addition to the glycemic index to guide choices, a diabetic patient can easily take control of their diet and take care of themselves properly.

2.3 Lifestyle of a Diabetic Individual

Type 1 diabetic individuals should test themselves four or more times a day, after every meal and just before going to sleep [4]. Although this is the general recommendation, patients who are at greater risk of hyperglycemia may require increased testing, to allow for tighter blood glucose management. Type 2 diabetic individuals often do not need such a strict control of their blood glucose. However, those who require multiple daily insulin injections or injections coupled with glucose reduction medication would require a testing regimen comparable to that of a Type 1 diabetic individual.

The rate at which blood glucose levels are measured is often determined after an individual is diagnosed with diabetes, and is the result of the combination of both the physician's recommendation and the patient's ability to maintain the regimen [4]. Individuals who are too young, too old, and others who are incapable of regular blood glucose testing often have more laidback testing plans, but those patients often lose the full efficacy of the treatment plan. In an investigational study on how often diabetic patients regularly checked their blood glucose levels, only 20% of Type 1 diabetic individuals and 16% of Type 2 diabetic individuals tested their blood glucose levels according to daily recommendations [17]. It is these inconsistencies in blood glucose monitoring that cause a decreased efficacy in prevention of diabetic symptoms.

2.3.1 Diet and Diabetes

Maintaining proper dietary habits is crucial aspect of minimizing diabetic symptoms. Type 2 diabetes is often attributed to a poor diet due to the fact that overweight individuals are shown to be at greater risk of attaining Type 2 diabetes [4]. One of the main recommendations to overweight diabetic individuals is weight reduction coupled with an improved dietary lifestyle. During this dietary change, individuals are encouraged to test their blood glucose after every meal. With this knowledge the individual should better understand how different foods affect their blood glucose levels, and give them an insight on consuming healthier foods.

Since it is the regulation of glucose, a carbohydrate, that is important to diabetics, it is more useful for an individual to understand the amount of carbohydrates that they are consuming than the number of calories or other nutritional characteristics in a food. Carbohydrate counting has been a useful tool in the regulation of blood glucose levels, and can be appropriately implemented by individuals with the proper guidance of a doctor, dietician, or a certified diabetes educator.

Individuals with Type 1 and Type 2 diabetes who are dependent on supplemental insulin injections must inject themselves with an appropriate amount of insulin prior to eating to compensate for the imminent glycemic increase produced by the carbohydrates in a meal [29]. Since insulin absorption varies between individuals, as well as the varying rates of efficacy amongst different types of insulin, most insulin plans are tailored to the individual.

2.3.2 Exercise and Diabetes

Due to the ever growing adaptation of a sedentary lifestyle, in which individuals are deskbound at work and inactive at home, coupled with the lack of an exercise plan, the onset of Type 2 diabetes has only been nurtured in recent years. As with the recommendation of proper eating habits, doctors often recommend that diabetic individuals increase their activity level by following an exercise program that is designed for the severity of their disease and by their overall ability to perform any particular activity. Since exercise helps lower blood sugar levels, Type 2 diabetic individuals can often manage their glucose levels without the use of any medications, through a strict diet and exercise routine [4]. Since the need for insulin during exercise is lowered, Type 2 diabetic individuals need to either consume an appropriate level of carbohydrates or lower their insulin intake before they partake in a vigorous activity. This requires the individual to understand how particular exercise affects their blood glucose level, and know the amount of carbohydrates to ingest to sufficiently accommodate those changes.

The consumption of extra carbohydrates (ExCarbs) prior to partaking in strenuous exercise is a concept that has been effective in supplementing the active lifestyles of Type 1 diabetic individuals [29]. The ExCarb model provides information on the number of grams of carbohydrates one should take prior to participating in a particular activity.

Table 4: Extra carbohydrates that one should consume before participating in vigorous activity. These values can be used as a guide to prevent a severe drop in blood sugar after exercise [26]. The ExCarb solution estimates the number of carbohydrates needed for one hour of activity based on the weight of the individual and type of activity. It exhibits a competence in providing good carbohydrate expenditure for aerobic activities.

Estimation of ExCarbs(g/h) according to type of activity/weight							
Activity	**Weight(kg)**				**Weight(kg)**		
	45kg	***68kg***	***90kg***		***45kg***	***68kg***	***90kg***
Baseball	25	38	50	***Running***			
Basketball				8km/h	45	68	90
moderate	35	48	61	13km/h	96	145	190
vigorous	59	88	117	16km/h	126	189	252
Bicycling				***Shoveling***	31	45	57
10km/h	20	27	34	***Skating***			
16km/h	35	48	61	moderate	25	34	43
22km/h	60	83	105	vigorous	67	92	117
29km/h	95	130	165	***Skiing***			
32km/h	122	168	214	cross-country 8km/h	76	105	133
Dancing				downhill	52	72	92
moderate	17	25	33	water	42	58	74
vigorous	28	43	57	***Soccer***	45	67	89
Digging	45	65	83	***Swimming***			
Eating	6	8	10	slow crawl	41	56	71
Golf (with pull cart)	23	35	46	fast crawl	69	95	121
Handball	59	88	117	***Tennis***			
Jump rope (80/min)	73	109	145	moderate	23	34	45
Mopping	16	23	30	vigorous	59	88	117
Mountain climbing	60	90	120	***Volleyball***			
Outside Painting	21	31	42	moderate	23	34	45
Raking Leaves	19	28	38	vigorous	59	88	117
				Walking			
				5km/h	15	22	29
				7km/h	30	45	59

The ExCarb solution estimates the number carbohydrates needed for one hour of activity based on the weight of the individual and type of activity. This solution exhibits a competence in providing good carbohydrate expenditure for aerobic activities, but during the cases of anaerobic activity such as sprinting, power lifting, or some aspects of basketball, the ExCarb methodology does not adequately estimate the expected expenditure rate.

2.3.3 Poor Blood Glucose Management

As previously discussed, an investigational study has shown that only 20% of Type 1 diabetic individuals and 16% of Type 2 diabetic individuals tested their blood glucose levels according to daily recommendations. This poor blood glucose management results in the occurrences of hyperglycemic and hypoglycemic conditions [17].

When blood glucose levels are consistently higher than 126 mg/dL, an individual is considered to be hyperglycemic [5]. During this hyperglycemic state, the body builds up a greater level of ketones. High levels of ketones in the blood, known as ketoacidosis, are poisonous to body, leading to long term comatose and potentially death. If high blood glucose levels continue to go unmonitored, damage to the cardiovascular, retinal, and nervous systems can occur [4].

When blood glucose levels are consistently lower than 60 mg/dL, on average, an individual is considered to be hypoglycemic [5]. Low glucose levels often make individuals tired and lethargic, causing them to faint, and sometimes causing them to become comatose. Diabetic individuals who have fallen into this state remedy the situation by the consumption of simple carbohydrates, like eating a piece of candy or drinking a glass of juice. If low blood sugar levels continue to go unmonitored, cellular function can no longer be maintained, which can lead to permanent neurological damage [4].

2.4 Activity Monitors

Activity monitoring of physical activity can occur in both laboratory and non laboratory conditions. This project intended to retain the accuracy of physical activity monitoring in laboratory conditions in a portable non-laboratory activity monitor. Activity monitoring in both of these settings is detailed in the sections below.

2.4.1 Monitoring Physical Activity in Laboratory Conditions

During exercise, the action of moving muscles requires energy in the form of adenosine triphosphate (ATP) [5]. ATP is produced in all cells during the oxidation of glucose. When the muscles are active due to exercise, they need more oxygen to oxidize glucose and produce ATP. To gain this extra oxygen, the normal human response is to breathe more, which results in greater oxygen consumption and the production of carbon dioxide [5]. In this case heart rate also increases so that more oxygen rich blood is provided to the working muscles. Thus, energy expenditure has traditionally been found to relate to CO_2 production and heart rate.

In the laboratory, gas analysis is the standard method of assessing the accuracy of activity monitors due to the correlation between CO_2 production and energy expenditure [26]. Gas analysis usually involves a subject breathing into a mouth piece so that volumes of CO_2 can be measured with each breath. Another method used to measure CO_2 production is doubly labeled water (DLW) in which the disappearance rates of two non-radioactive isotopes of water ($H_2{}^{18}O$ and $^{2}H_2O$) are observed. The difference in the disappearance rates of each isotope is used to calculate the amount of CO_2 produced per unit of time. By knowing the CO_2 production rate and the diet of a subject, energy expenditure can be calculated [27]. Although an accurate method, DLW is a relatively expensive and time consuming test to administer, therefore it would not be a suitable method to continuously monitor patient's energy expenditures in a portable device.

2.4.2 Monitoring Physical Activity in Non-laboratory Conditions

Currently, there is no single, universally accepted device for measuring energy expenditure for individuals available on the market. However, there are many devices that use a variety of parameters such as pedometer counts, heart rate, and accelerometry counts to assess the energy expenditure of physical activity in individuals. Still other devices use a combination of these parameters to output energy usage.

Pedometers

A pedometer is a device used to count each step a person takes through motion detection [23]. Many pedometers contain a sensor which moves vertically whenever a person takes a step. Movement of the sensor closes an electrical circuit, and each time the circuit is closed, it counts as a step. A summation of the amount of times the circuit closed within a certain period of time is displayed on the pedometer screen. Pedometers are often small, lightweight, inexpensive, and can be used during every day activities. However, they do not account for activity types and intensity and may lose accuracy during intense activity.

The Digiwalker pedometer shown in Figure 3 has been found to be superior to other devices of its kind and boasts accurate counting due to its state of the art lever arm sensor, a tightly coiled spring for increased accuracy and special features to reduce noise and corrosion [18].

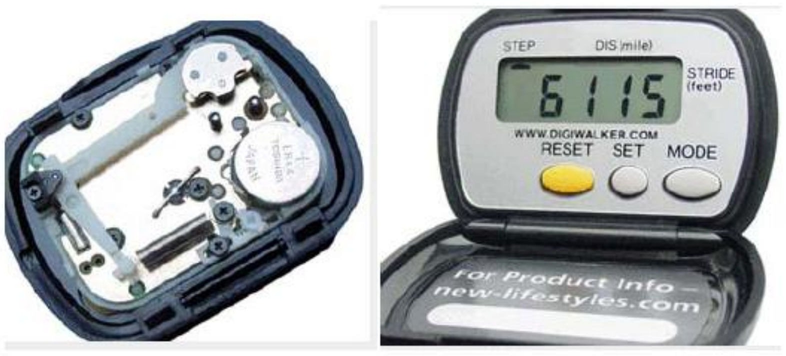


Figure 3: Digiwalker Pedometer boasts accurate counting due to state of the art lever arm sensor, a tightly coiled spring for increased accuracy, and special features to reduce noise and corrosion [18].

Another activity monitor, developed specifically to monitor the activity of those with Type 2 Diabetes is described in a Norwegian study [19]. The device designed in this study is different from Digiwalker in that it contains Bluetooth technology to wirelessly transmit data to a mobile phone as shown in Figure 4 below [20].

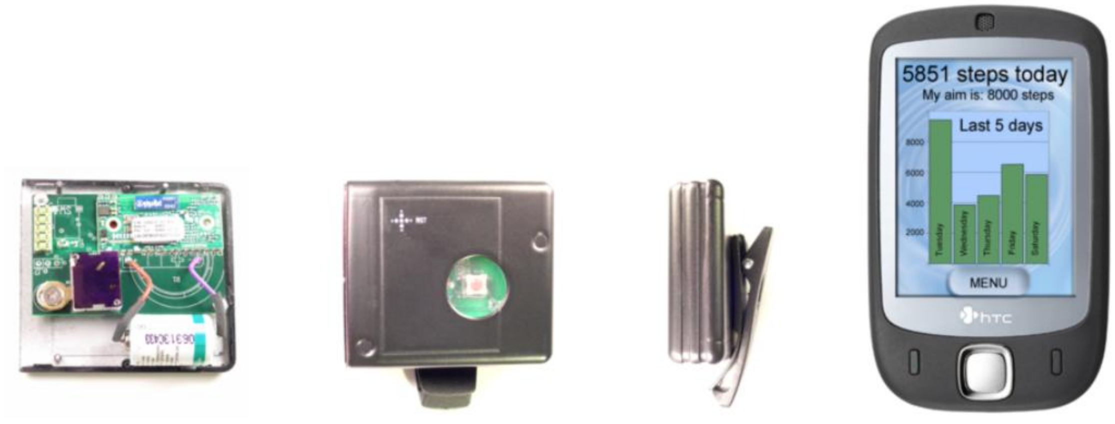


Figure 4: Final prototype of activity monitor for Type 2 diabetic individuals. The device contains Bluetooth technology to wirelessly transmit data to a mobile phone [19].

To develop this monitor, 15 people with Type 2 Diabetes provided the researchers with input on what features a new, mobile device should have. Over a period of four months the subjects gave their opinions on what specific tools the device should have as well as opinions on the overall look and comfort of the device. Based on these opinions, the researchers designed a pedometer activity monitoring device that met the needs of the diabetic patients. This activity monitor contains a specifically designed mechanical sensor chosen to minimize power consumption. Also, the device has capabilities to filter out random movement and any outside noise. To assess the marketability of the device, the researchers asked 1001 people about their use of step pedometers. Only 6.5% used pedometers daily while 20% used them daily, weekly, or monthly. The appeal of this device is that it is small, compact and discrete. However, pedometer activity monitors may not be as accurate as devices that consider multiple parameters.

Heart Rate Monitors

Heart rate has been validated against the doubly labeled water technique for estimating energy expenditure and has a correlation coefficient of 0.94 as compared to DLW [28]. However, it is an indirect method of measuring energy expenditure and thus data often presents discrepancies.

Heart rate monitors on the market commonly boast higher accuracy rates than pedometers for measuring energy usage. Heart rate is measured by first placing electrodes on the chest. These electrodes acquire an ECG signal which is then fed to a data processing unit. This processing unit has the capacity to calculate the heart rate. As discussed previously, heart rate measured in laboratory conditions is also a method used to assess the accuracy of portable activity monitors.

Polar is the leading manufacturer of heart rate monitors [18]. These monitors have a wide range of uses including fitness programs, athletic teams, medical purposes and physical education programs. This specific heart rate monitor consists of a transmitter belt that is worn around the chest and a device that is attached to the wrist. The belt has a fabric strap and two electrodes that are arranged to measure the potential difference across the chest made by the two electrodes. This potential difference is sent to the wrist device which receives and processes the signal, as well as calculates and displays the user's heart rate. Figure 5 shows the Polar heart rate monitor in use.

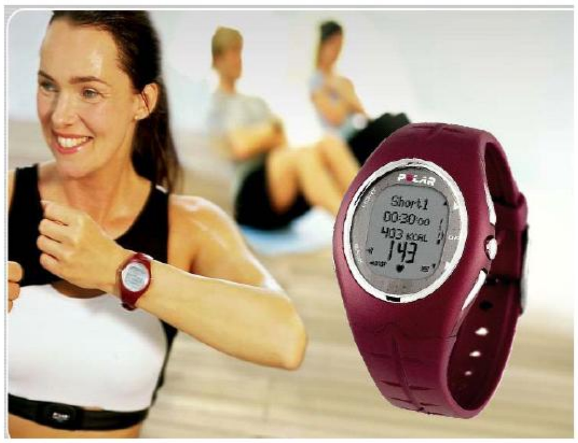


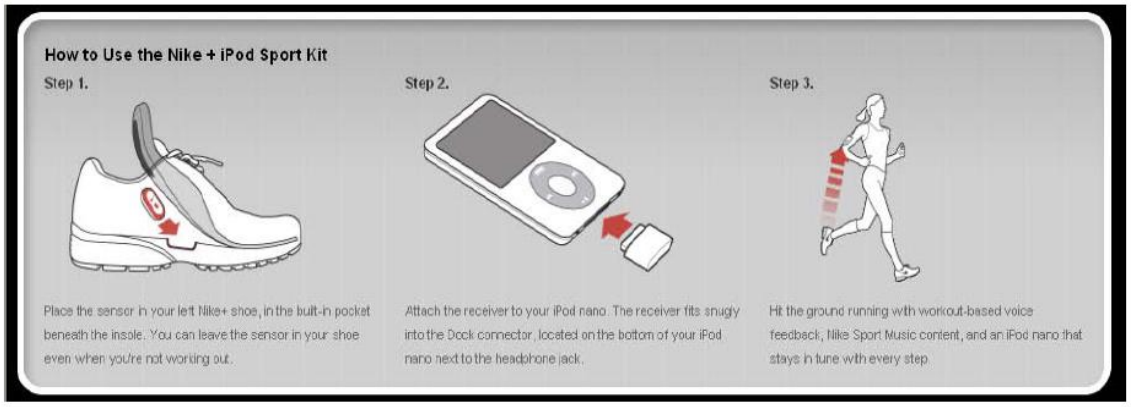
Figure 5: Polar heart rate monitor with chest strap and wrist device. Consists of a transmitter belt worn around the chest and a device that is attached to the wrist which displays energy expenditure parameter output [18].

Other heart rate monitors include additional features, such as the Garmin Forerunner 305. This heart rate monitor contains a GPS feature to measure physical distance travelled and pace of activity [20]. Heart rate monitors are reasonably accurate in obtaining and calculating heart rate, however, they may have several inaccuracies when applied to activity monitoring. For example, if a person is nervous and their heart is beating at a high rate, the heart rate activity monitor will interpret this information as though the person is undergoing strenuous physical activity when in fact they could remain motionless. Therefore, an activity monitor that considers multiple parameters may have a higher accuracy than a single parameter device such as a heart rate monitor.

Accelerometers

Accelerometers are devices which detect when a subject is in motion [22]. Not only can they detect human movement, but they also detect the intensity and frequency of movement. The output of an accelerometer can be processed to isolate the frequency band of human movement which can provide a measurement for quantifying human activity [18]. In this way, using accelerometer outputs is a more accurate method than using heart rate to evaluating energy expenditure.

An example of an accelerometer used to calculate energy expenditure is the Nike+iPod sport kit which utilizes a piezoelectric accelerometer that is placed in the user's shoe [21]. When the user's foot hits the ground, the piezoelectric device senses the movement and wirelessly transmits the signal to a receiver which is attached to an iPod. The receiver counts each step and is able to calculate the distance traveled by the user. This receiver is able to calculate the user's time, distance, pace, and burned calories and the information is displayed on the iPod. Figure 6 displays the technology behind the Nike+iPod sport kit. The manufacturer of Nike+iPod boasts 90% accuracy for first time use and increased accuracy if the device is calibrated. Nike or iPod has not, at this time, released research papers validating their device or information regarding any clinical trials.


How to Use the Nike + iPod Sport Kit

Step 1.

Place the sensor in your left Nike+ shoe, in the built-in pocket beneath the insole. You can leave the sensor in your shoe even when you're not working out.

Step 2.

Attach the receiver to your iPod nano. The receiver fits snugly into the Dock connector, located on the bottom of your iPod nano next to the headphone jack.

Step 3.

Hit the ground running with workout-based voice feedback, Nike Sport Music content, and an iPod nano that stays in tune with every step.

Figure 6: Nike + iPod Sport Kit User's Guide. The kit uses a piezoelectric accelerometer placed in the user's shoe to sense movement and transmit signals to the iPod, where each step is counted to calculate distance travelled [21].

The shortcoming of this type of accelerometer activity monitoring device is that the accelerometer is placed on an extremity, in this case the leg. Because the extremities often have more movements than the rest of the body, inaccuracies in movement counts may occur. For example, if a subject is wearing the Nike+iPod device and is sitting in a chair gently tapping their foot on the ground, the accelerometer will detect the movements and the subject will have a much higher energy expenditure reading than they actually have. The Nike+iPod is only accurate when measuring movements during physical activity, and for diabetic patients, a device is needed to measure expenditure during all activities – physical and sedentary.

The Caltrac Calorie Counter, shown in Figure 7, is another accelerometer device that uses a single plane accelerometer to detect only vertical movements [22]. The counter has an accelerometer at the waist which measures vertical movements and displays a count of these movements on a display screen that houses the accelerometer circuitry. Although this device has been tested and verified in 60 clinical trials as a valid device for calculating energy expenditure, it does have several shortcomings. One major shortcoming is that the device does not accurately estimate energy usage for activities involving isometric movements such as cycling and rowing [18]. An accelerometer that measures movements in three axes may be better suited to calculate energy expenditure.

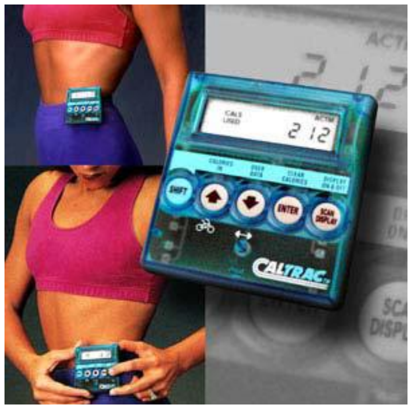


Figure 7: Caltrac Calorie Counter which uses a single plane accelerometer to detect vertical movements at the waste [22].

Multiple Parameter Activity Monitors

Devices that use only one parameter may have limitations on the types of activities that can be measured and how accurate the output is to the actual energy expenditure [22]. Connecting three Caltrac accelerometers in an orthogonal fashion has a much better accuracy than just one stand alone accelerometer [22].

In a study by Eston, Rowlands, and Ingledew, four different activity monitors were tested and compared to one another based on the energy expenditure output in children [23]. The devices used in this study included a uni-axial accelerometer, a tri-axial accelerometer, a pedometer, and a heart rate monitor. These devices were worn by 30 children who participated in several activities. The accuracy of each of the devices was evaluated for each activity.

This study showed that the tri-axial accelerometer was the most accurate while the uni-axial and heart rate monitor were also valid as less expensive options. The study also concluded that using an accelerometer in conjunction with a heart rate monitor would enable the device to output the most accurate energy expenditure. Figure 8 below shows the mean error and standard deviation of each type of device.

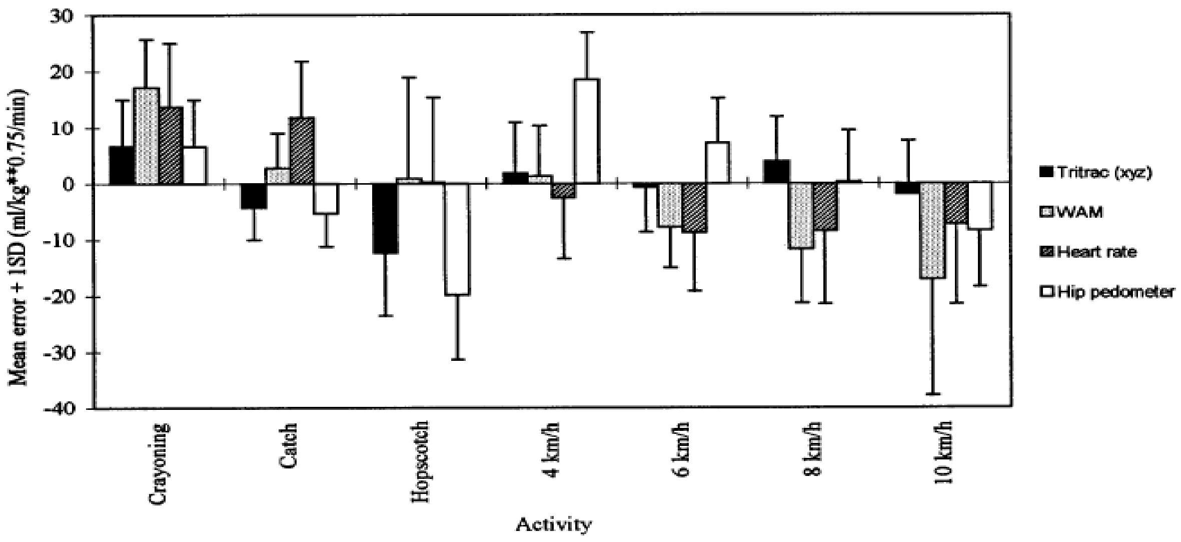


Figure 8: Mean error and standard deviation of activity monitors tested in testing by Eston, Rowlands, and Ingledow [23].

Actiheart, shown in Figure 9, is an example of a device that utilizes both heart rate and accelerometry to determine energy expenditure. This device clips onto two chest electrodes and contains a triaxial piezoelectric accelerometer. Heart rate is calculated by first converting the analog ECG signal to digital and by calculating the R to R ratio. By using this method to calculate heart rate, the output is more accurate than the traditional peak detection method. Combining heart rate with accelerometry combines the best aspects of each technology to output the most accurate energy expenditure result.

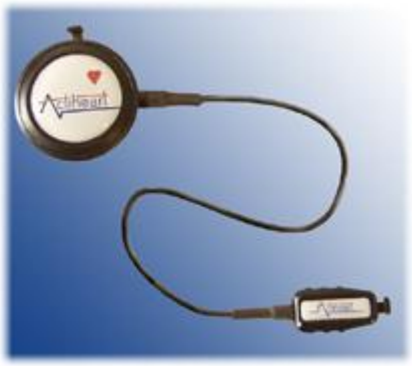


Figure 9: Actiheart monitor which uses heart rate and accelerometry to output energy expenditure. The device slips onto two chest electrodes and has a triaxial piezoelectric accelerometer [24].

A study was done in 2007 to determine the accuracy of the Actiheart system [25]. This study was conducted by SE Crouter, JR Churilla, and DR Bassett, Jr of the Depart of Exercise, Sport, and Leisure Studies at the University of Tennessee, Knoxville. The intention of this study was to see how accurate the Actiheart system was outside of laboratory conditions. To do this, it included 18 tests that were performed by having subjects wear the Actiheart system and the Cosmed K4b2, a device that uses indirect calorimetry to determine energy expenditure. The Actiheart system is a device that will clip onto standard ECG electrodes. This clip houses the accelerometers. The Cosmed device is a portable metabolic machine. It includes a gas mask and a processing unit.

The tests included in the study are as follows: lying, computer work, standing, filing papers, washing dishes, washing windows, slow walk (82 m min-1), vacuum, sweep/mop, raking grass/leaves, fast walk (103 m min-1), lawn mowing, stationary cycling (avg. 99 W), ascending/descending stairs, racquetball, basketball, slow run (157 m min-1), and fast run (191 m min-1) [25]. Upon completion of the study, the investigators determined that the Actiheart system is comparable to other activity monitors, though improvements could be made.

Activity monitors for commercial use output energy expenditure using a variety of parameters including heart rate, accelerometry, and pedometry. Although most devices that utilize one parameter are acceptably accurate, monitors that use a combination of parameters have an increased accuracy.

3 Design Methods and Alternatives

The initial client statement given to the design team in August of 2008 was as follows:

"To design and build a continuous activity monitoring device for insulin dependent diabetic individuals that will output cumulative energy expenditure and rate of energy expenditure. Energy expenditure will be calculated using multiple parameters for increased accuracy in monitoring daily activity. This device will be low cost, aesthetically pleasing, and convenient for everyday use. "

The design team thus set forth to obtain design constraints, objectives, and alternative designs based off of this statement. A weighted objectives method was used to compare the alternative designs and ultimately determine the design to proceed with.

3.1 Obtaining Design Constraints

Constraints in engineering are limitations set on the design space by both the client's problem statement and the design team's familiarity with various branches of engineering sciences [2]. The client statement addresses the wants and needs of the client for a particular design, as well as what it must and should complete. In the case of the design of a diabetic activity monitor, the requirements for the device are that it outputs accurate energy expenditure data frequently (at least once per minute), that it is inconspicuous when worn, and that it is inexpensive enough so as to be reasonable for purchase by every-day civilians. As a result of these requirements the team was able to deduce that wearability, durability, accuracy, ability to be produced within the allotted MQP time (A-C term), and a total budget of $625 are all constraints which govern the design and production of this device.

3.2 Obtaining Design Objectives

Objectives are the end goals a design aims to achieve [2]. Design objectives were initially created by the design team in list form and then organized into an objectives tree which contained objectives and sub-objectives. The design team concluded that a minimum of three sub-objectives should also be included for each of the six main objectives for each main objective to hold enough importance in the outcome of the overall design. Figure 10 below is a diagram displaying all identified objectives and sub-objectives.

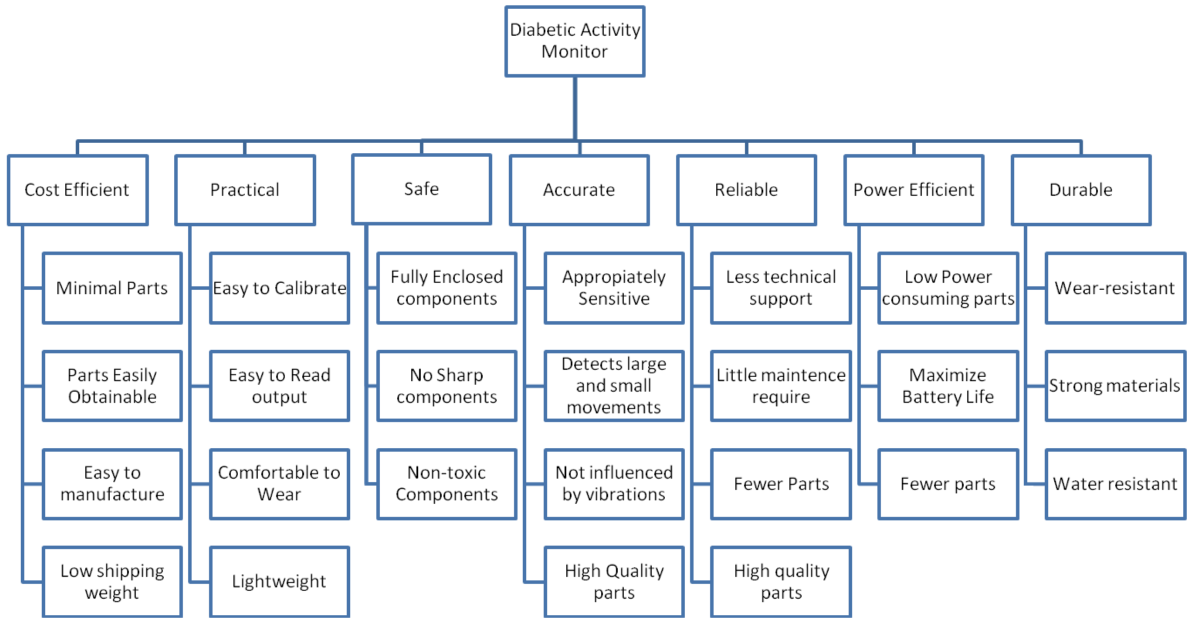


Figure 10: Objectives tree designed by the design team prior to detailed weighted objectives analysis.

With the six main objectives and a minimum of three sub-objectives each, the team completed a pair-wise comparison chart of the main objectives to obtain ratios of importance of each main objective relative to the total design, shown in Table 2 on the following page. Following this analysis, each of the sub-objectives was also weighted using a pair-wise comparison chart. These pair-wise comparison charts allowed for the completion of the weighted objective tree. The weighted importance of the sub-objectives of each main objective can be found in the tables in Appendix A.

Using these tables, each of the design alternatives generated by the team was ranked according to their potential ability to fulfill each objective thus far identified as important for the project. The weighted objectives chart resulting from these calculations can be seen in Figure 11 below.

Table 2: Pair wise Comparison Chart for Main Objectives. To prioritize the objectives the team devised a pair wise comparison chart which couples two objectives with one another, ranking one over the other, until all of the objectives have been ranked fro most important to least important. A score of 1 is given to the cell if the objective in the corresponding row is of greater importance than the objective in the corresponding column. A score of 0.5 is given if the objectives in the corresponding row and column are of equal importance. A score of 0 is given when the objective in the corresponding row is of less importance than the objective in the corresponding column.

Objective	Cost Efficient	Practical	Safe	Accurate	Reliable	Power Efficient	Durable
Cost Efficient	* * *	0.5	0	0	0	1	0.5
Practical	0.5	* * *	0	0	0.5	1	0.5
Safe	1	1	* * *	0.5	0.5	1	1
Accurate	1	1	0.5	* * *	0.5	1	0.5
Reliable	1	0.5	0.5	0.5	* * *	1	0.5
Power Efficient	0	0	0	0	0	* * *	0.5
Durable	0.5	0.5	0	0.5	0.5		* * *

Objective	Score + 1
Cost Efficient	3
Practical	3.5
Safe	6
Accurate	5.5
Reliable	5
Power Efficient	1.5
Durable	3
TOTAL	28

High Ranking	**Weighted %**
Cost Efficient	10.91%
Practical	12.73%
Safe	21.82%
Accurate	20%
Reliable	18.18%
Power Efficient	5.45%
Durable	10.91%
TOTAL	100%

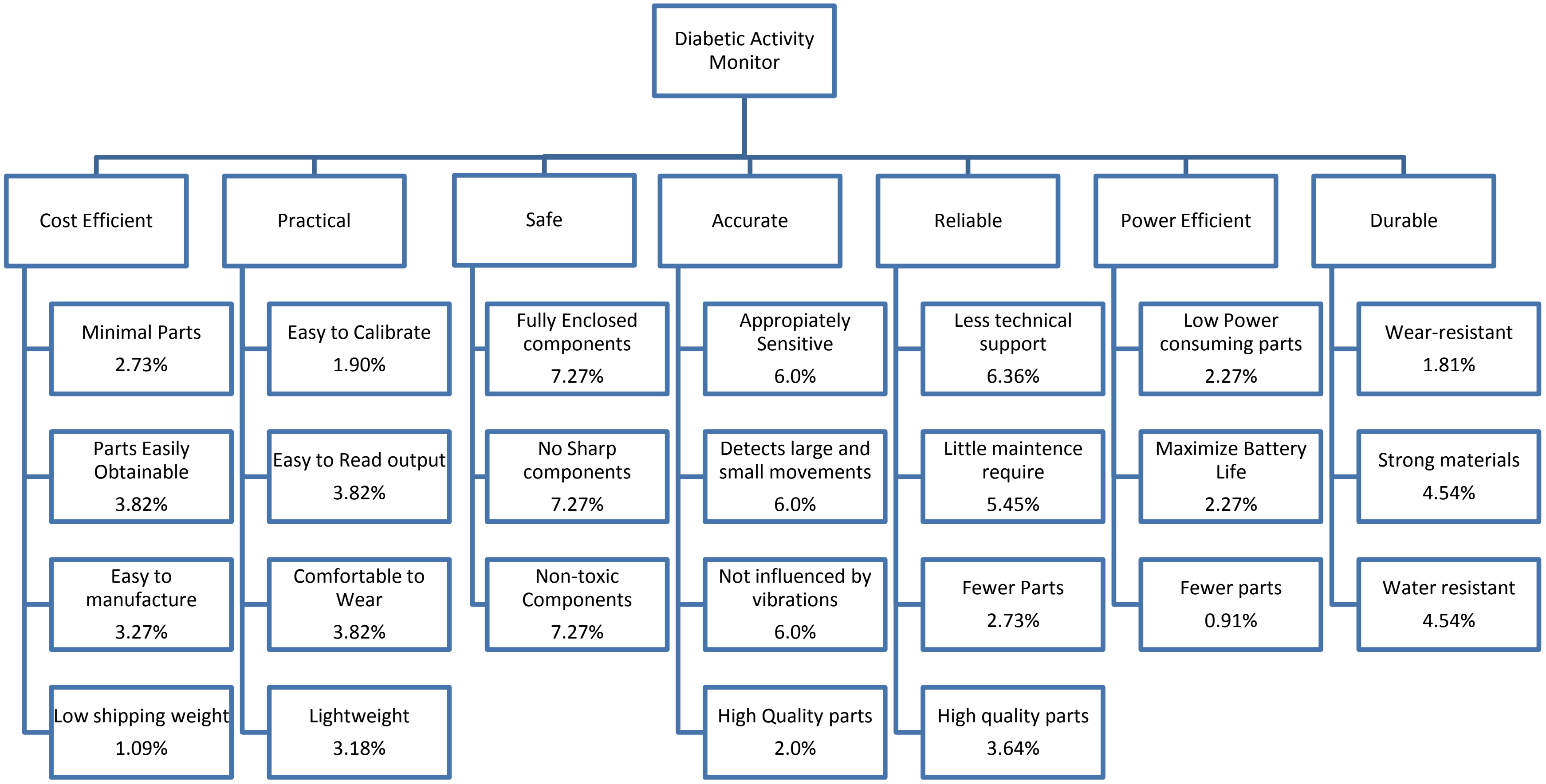

Diabetic Activity Monitor
Cost Efficient
Practical
Safe
Accurate
Reliable
Power Efficient
Durable
Minimal Parts 2.73%
Parts Easily Obtainable 3.82%
Easy to manufacture 3.27%
Low shipping weight 1.09%
Easy to Calibrate 1.90%
Easy to Read output 3.82%
Comfortable to Wear 3.82%
Lightweight 3.18%
Fully Enclosed components 7.27%
No Sharp components 7.27%
Non-toxic Components 7.27%
Appropiately Sensitive 6.0%
Detects large and small movements 6.0%
Not influenced by vibrations 6.0%
High Quality parts 2.0%
Less technical support 6.36%
Little maintence require 5.45%
Fewer Parts 2.73%
High quality parts 3.64%
Low Power consuming parts 2.27%
Maximize Battery Life 2.27%
Fewer parts 0.91%
Wear-resistant 1.81%
Strong materials 4.54%
Water resistant 4.54%

Figure 11: Weighted objectives for the Diabetic Activity Monitor. After completing the pair wise comparison chart, the team calculated that the safety of the device is the most important aspect of its design. Accuracy of the device was the second most important, while reliability of the device ranked third. The team found that power efficiency of the device was the least important aspect of the design of the device.

3.3 Design Alternatives

Prior to determining various conceptual designs for the diabetic activity monitor, many methods of measuring human physical activity were analyzed to determine the best means of obtaining accurate energy expenditure values for the device. The discussed methods of measuring physical activity included monitoring of heart rate, respiratory rate, and movement detection. After determining these relevant forms of measurement the group held a meeting to begin a design brainstorming session. This brainstorming session included determining the necessary functions and function means in order to create workable activity monitor for diabetic individuals. The functions identified allowed for the design team to obtain a better grasp upon the design space and to begin brainstorming possible designs. The functions and means of attaining these functions are shown below in Table 3.

Table 3: Function means chart to determine possible means to achieving device goals.

Function	**Possible Means**				
Measure RR	Strain Gauge	Auscultatory	Plethysmography	Pulse Oximetry	Piezoelectric
Digital Processing	MSP430 Chip	PSoC Chip	PIC	Code written in Assembly	Code written in C
Measure HR	Electrode	Polar Strap			
Obtain movement count	Accelerometry	Strain Gauge			
Determine Carbohydrates	Actiheart Algorithm				

The design team determined that any one of these forms of measurement, or parameters, would not be acceptable as stand alone measurements for an energy expenditure output, for several reasons.

Heart rate was identified as a potential parameter for physical activity measurement as the rate of heart beats typically corresponds with the amount of exercise being performed by an individual. The means to acquiring heart rate which were identified include electrode placement and the use of a Polar strap. While heart rate can be an accurate measure of level of physical activity in athletic situations, it may be superficially stimulated, such as when a person becomes excited or nervous. It is in this case that it becomes necessary for a separate parameter to be used and compared with that of heart rate in order to determine the true energy expenditure in the moment.

Respiratory rate was identified as another possible means for determining level of physical activity as an individual's rate of breathing typically directly corresponds with that person's level of activity. Possible means to acquiring respiratory rate which were identified are strain gauge methods, bioacoustic methods, plethysmography, and pulse oximetry. Much like heart rate, it was determined to be a relatively accurate measure for physical activity, but in cases such as hyperventilation and hypoventilation due to nerves or physical environment (quality of air can be a factor) another parameter would be necessary to ensure that energy expenditure readings were correct. Because accurate and discrete respiration monitors are hard to come by, respiratory rate was eliminated as a parameter during the creation of alternative designs.

Accelerometers are devices which are able to detect magnitude and direction of acceleration as a vector quantity. Accelerometry was identified as a parameter which could be useful in determining physical activity as it is based off of movement. A comparison between accelerometry and either of the two aforementioned parameters could easily determine whether or not the device user is physically active at the time of measurement and what the energy expenditure of the user was during times of activity.

As it was clearly determined that the utilization of at least two parameters in combination was necessary for accurate energy expenditure calculations digital processing and determining carbohydrates were considered to be two necessary device functions. Research from the previous diabetic activity monitor MQP showed that ActiHeart has an algorithm which allows for the comparison of heart rate with accelerometry in order to obtain energy expenditure [25]. Thus electrical components which could be programmed to accept two signals, use algorithms to compare these signals, and output those calculations to a display were considered.

The following four sections detail the conceptual design ideas created by the design team while taking the information compiled in the function means chart into consideration.

3.3.1 Conceptual Design 1 – Electrodes and Accelerometer Design

The first conceptual design evaluated by the team was the electrodes and accelerometer MQP device from the 2007-2008 year. This design involves electrodes placed on the chest in order to monitor heart rate and an accelerometer worn on the hip to monitor whole body movements. Figures 12 and 13 below show the device design.

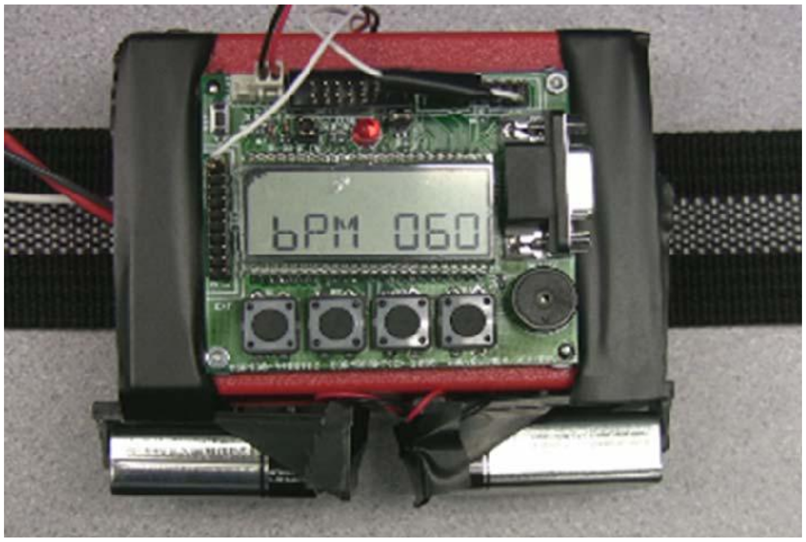


Figure 12: Electrodes and accelerometer design from 2007-2008 MQP [18].

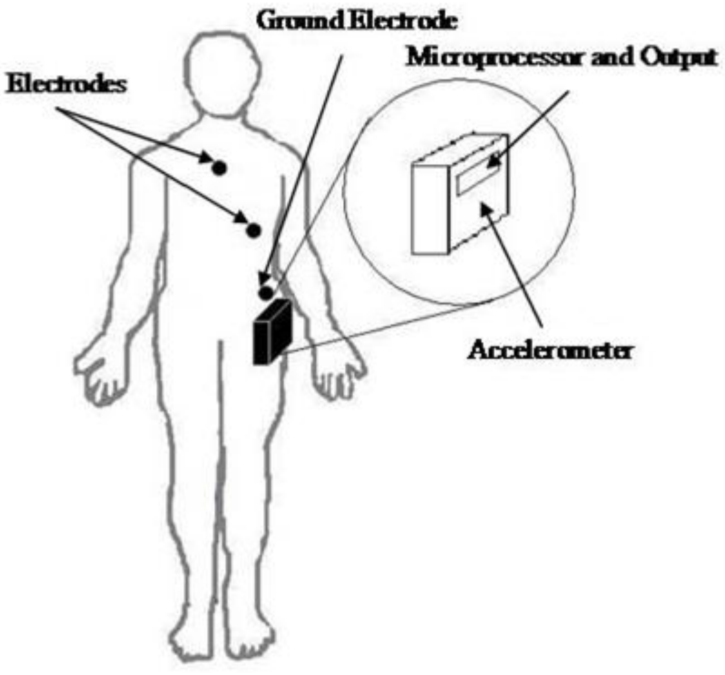


Figure 13: Diagram of 2007-2008 MQP device. An accelerometer and microprocessor housed on the hip of the user, with electrodes placed on the hip and sternum to measure the user's leg [18].

Design idea 1 houses an accelerometer and a microprocessor on the hip of the user and places electrodes on the hip the sternum of the user. Electrodes measure the ECG of the user. The microprocessor mounted on the hip processes the accelerometry and ECG signals and outputs data for the user. The top portion of the figure was the MQP device from the 2007-2008 year.

Reasons for Elimination

Some of the main reasons for deciding not to continue with the use of this design were that: (1) the processing was contained in a bulky housing unit worn around the waste; (2) it was run off of two poorly placed 9V batteries; (3) the device was heavy and bulky; and (4) the device did not output carbohydrates or allow the individual to keep track of carbohydrate usage. It was also very inconvenient to try and place the electrodes correctly. The electrodes and accelerometer design was reviewed both verbally by the design team and through the use of weighted objectives charts. It complied by the team's weighted objectives by 57%, which was decidedly not adequate. The individual objective comparisons made for the device and each of the following devices can be seen in Appendix A.

3.3.2 Conceptual Design 2 – Parameter Detection Glove

The parameter detection glove conceptual design uses detection of pulse rate and acceleration to determine energy expenditure. The device idea consists of one part, a form fitting glove that houses a pulse sensor, accelerometer, and a display. The form fitting glove would allow the sensor to be pressed tightly against the wrist of the individual, and the display would be located on the proximal side of the hand, not to interfere with the grasp of the user. The battery, accelerometer and computational components would be housed beneath the display so that cyclical wear would not affect the electronics. Figures 14 and 15 below shows an example of this design.

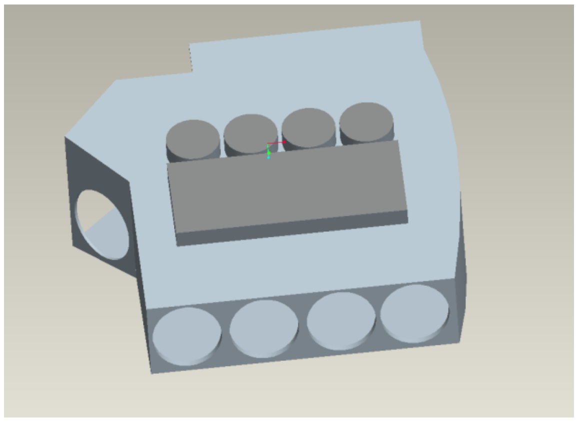


Figure 14: Three dimensional representation of the parameter detection glove. Pulse rate and acceleration would be sensed in the wrist of the individual, with a display of energy expenditure output located on the proximal side of the hand.

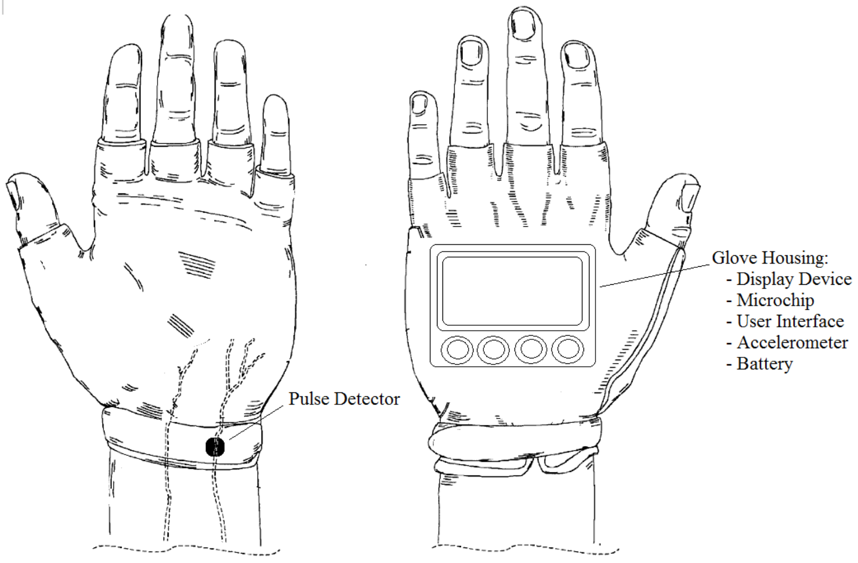


Figure 15: Illustration of parameter detection glove. An accelerometer and pulse sensor housed in a sensor module on the glove. The sensors would be pressed firmly against the wrist of the user. Energy expenditure data displayed on the proximal side of the hand.

Reasons for Elimination

The main problem with this design is that since the accelerometer is located on an extremity that moves on a regular basis, it will not allow for an accurate representation of the user's activity. These inaccuracies would make it difficult to properly calibrate the device, as well as create an appropriate sensitivity range. Also, as the device would be worn like a glove, the material would have to be water resistant and wear resistant. It would be difficult to manufacture a device with these mechanical constraints while preserving an ergonomic and comfortable form factor. The glove conceptual design complied by the team’s weighted objectives by 63%, which was determined to be inadequate for this project.

3.3.3 Conceptual Design 3 – Parameter Detection Sock

The sock conceptual design uses pulse rate and acceleration to determine energy expenditure. The device has three components to it: a form fitting ankle guard, an electronic anklet, and an LCD to display vital readings. The form fitting ankle guard would be made out of a washable material, encompassing a sensor that would contact the back portion of the ankle to measure a pulse. The electronic anklet would house the accelerometer, wireless transmitter, computational electronics, and power. The anklet would be connected to the form fitting ankle guard via a flexible detachable wire. The housing containing the LCD display would also contain a wireless receiver and transmitter as well as buttons to input calibration data. This design can be seen below in Figures 16 and 17.

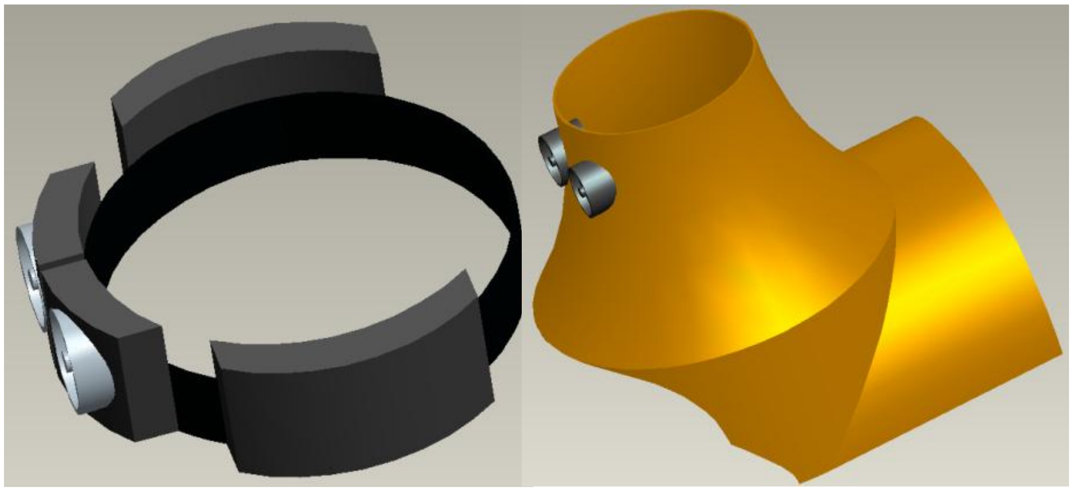


Figure 16: Electronic anklet and ankle guard. The electronic anklet would house the accelerometer, wireless transmitter, computational electronics, and power. The ankle guard would encompass a sensor to measure a pulse from the back of the ankle.

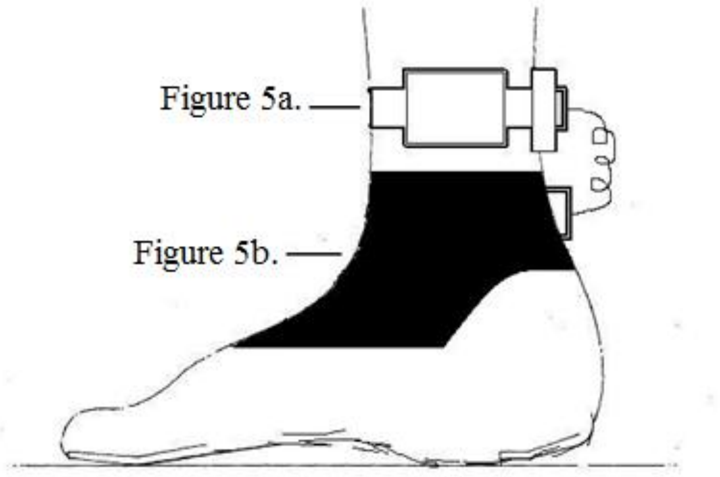


Figure 17: Illustration of electronic anklet and form fitting ankle guard.

Reasons for Elimination

Some of the pitfalls of this design deal with its inability to distinguish accelerations due to a swinging leg when sitting and the acceleration of a body when running. Due to this shortcoming, it decreases overall sensitivity, ease of calibration while increasing the need for technical support. Since the pulse sensor would be located at a joint, it would be quite difficult to filter out any unwanted vibrations. Since the ankle guard would incur cyclic loading by the user, the material would have to be highly resistant to constant wear and because that portion of the device will be located on around the foot, the material would have to water resistant. The type of material needed to create this device is too specific to expect that there will not be errors in the manufacturing of the device. Overall the sock conceptual design complied by the design team's objectives by 59%, which was not sufficient enough to continue with the design process.

3.3.4 Conceptual Design 4 – Polar Strap and Accelerometer Design

There are two main measurement components to this design: the accelerometer and heart rate detector. Heart rate is an indication of energy expenditure; as people exert themselves, they must supply more oxygen to their muscles, making their hearts work harder. The faster a person's heart rate, the more energy is being used. However, heart rate is not a perfect indicator of energy usage. Stimulants, such as caffeine, can increase heart rate while energy expenditure remains unchanged. Additionally, a physically fit person's heart rate will plateau with exercise even as they continue to use more energy. For example, while a person runs uphill, his or her heart rate may remain the same as when he or she is running on flat ground despite working harder. As a result, an accelerometer will be used to account for times when changes in heart rate do not correspond with changes in movement. The concepts behind this design are similar to last year's diabetic activity monitor MQP (reviewed in section 3.3.1). However, this design differs from the previous MQP in that it uses a polar strap and heart rate monitor.

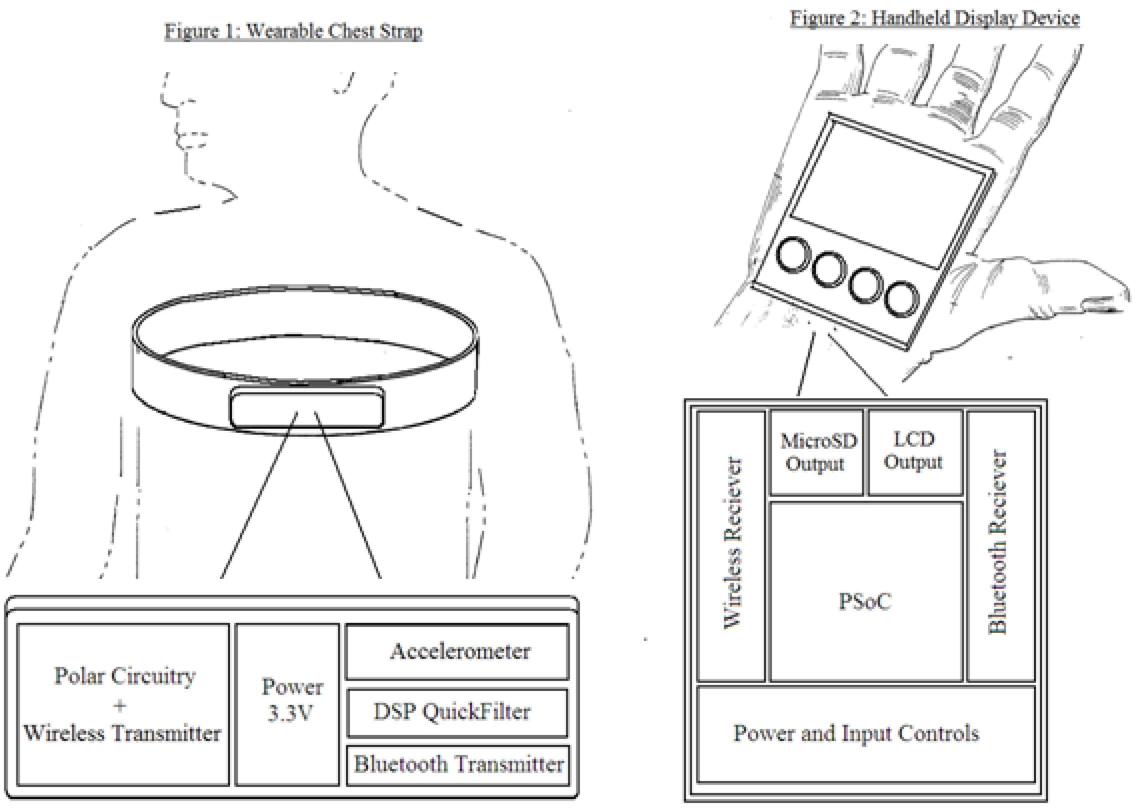


Figure 18: The Polar Strap and Accelerometer Design. This design is composed of two distinct parts - the Polar strap band warn around the chest by the user, and a handheld digital display system for ease-of-use purposes.

There are several advantages of using a polar strap over electrodes. First, the Polar strap is easy to put on and comfortable to wear. Electrodes must be applied separately after cleaning the skin and stick to the skin with an adhesive, which can be painful to remove, cause irritation, and fall off over the course of daily activity. The polar strap is an adjustable elastic strap that goes around the chest; in this case the electrodes are built into the strap and do not need to be adhered to the skin. This makes the device more discreet, convenient, and comfortable to wear. The polar strap has two connectors that snap onto a small package containing the circuitry that is needed to convert the heart rate signals into a pulse and then transmit it. The package is centrally located on the torso and would also be a good place to include the accelerometer in order to minimize measurement of non-energy expenditure related motion.

In this design, the accelerometer and filter circuitry would be encased in the polar package. A separate device would be necessary to contain the microprocessor, memory, and user interface. This device would be worn in an easily accessible place, so that the display will be visible to the wearer. Since it will be observable, the package must be discrete, resembling a watch or PDA. The device would receive wireless signals from the polar package, providing the processor with heart rate and accelerometry readings. These readings would be used to calculate the rate of calorie usage, which would be displayed for the wearer.

This design requires a minimal amount of parts encased in two or three water-resistant packages. Having fewer parts will not only make the product more cost efficient but also smaller and more discreet. Overall the polar strap and accelerometer design complied by the design team's objectives by 85%, and was thus determined to be the most reasonable design to use.

4 Detailed Design

The final design chosen for this device was the Polar Strap and Accelerometer Design. This device is composed of two parts, the signal acquisition module, which contains the circuitry for acquiring heart rate and full body acceleration, and the display module, which contains the programming for signal processing, the energy expenditure algorithm, and the user interface.

4.1 Signal Acquisition Module

The signal acquisition module is the portion of the device which gathers information from the device user. It uses a Polar strap to gather heart rate information, and an accelerometer to gather full body acceleration. The acquisition of these signals is detailed in the sections below.

4.1.1 Acquiring Heart Rate

The previous MQP group used the 3-lead ECG method to obtain the electrocardiogram signal, and processed the signal with analog circuitry to obtain a heart beat signal. The digital system measured the time interval between each heart beat to calculate heart rate. Several issues are associated with this design. The 3-lead ECG method to acquire heart rate is highly impractical when considering the usability requirements of this device. The device is intended for all day use, thus it must be easy to set up and operate. The previous MQP noted the difficulty in applying electrodes on different body types and ensuring good contact throughout the clinical test for each patient. The electrode wires are another reason the previous device was not feasible for market. These wires could easily get in the way of normal activity or cause the electrodes to be pulled off the skin.

The alternative of acquiring heart rate without a 3-lead ECG system is through employing an existing, market device known as the Polar system. This system is composed of a Polar strap with a built-in transmitter and a Polar RMCM01 receiver. The transmitter is worn around the chest, and it generates a 1 msec digital pulse for each heartbeat it detects, which it then transmits to the receiver. The receiver wirelessly detects the output of the transmitter, and upon arrival of a pulse, it will generate a digital pulse that electronics further down the line can use. The wireless communication between the transmitter and receiver is done with a low frequency electromagnetic field. Therefore, they must be aligned in parallel in order to obtain optimal performance.

The simplest heart rate measuring system connects the RMCM01 output to a digital I/O of a microprocessor, and calculates the heart rate by measuring the time interval between each digital pulse. It must be able to measure the time interval between input triggers with a reasonable resolution. The resolution of time would directly influence how accurate the heart rate figure would be.

4.1.2 Measuring Full-body Acceleration

The purpose of the signal acquisition module was to acquire the full body acceleration of the diabetic individual; full body acceleration is the sum of a person's movement in three axes. The acceleration was measured on the torso because it is more representative of the overall physical activity of the user. If the acceleration were to be measured on the extremities, such as a hand, the results would be skewed. For example, when a person gestures as they talk, the accelerometer would read larger movements, with high gravity values (up to 120 g), which would translate into higher activity levels. However, the individual would actually be relatively inactive and expending very little energy.

4.1.2.1 Accelerometer: Freescale MMA7261QT

The accelerometer used in this design was Freescale's MMA7261QT. It is a triaxial, analog accelerometer with a selectable g (gravity) range. The range of g values that the accelerometer will experience while attached to a human torso is ±6 g. However, the accelerometer was configured to operate in the ±10 g range so that the output signal voltage range will fall within the acceptable range of input signal voltages for the Quickfilter chip. This particular chip was selected for this reason as well as its low power consumption.

4.1.2.2 Digital Signal Filtering: Quickfilter QF4A512

The accelerometer outputs three analog voltages, one for the x, y, and z axes each; these signals fluctuate around a zero-g offset (a fixed voltage representing no acceleration in the direction of the axes.) A graphical representation of this scenario can be seen below in Figure 19. When the three axial outputs are summed, the result should only include voltages outputted from the accelerometer due to movement. The zero-g voltage, which results from no movement, and therefore no energy expenditure, must be eliminated from the axial outputs before they are summed. This offset is a direct current (DC) signal component, meaning that it has a frequency of zero Hz. This DC part of the acceleration signal can be filtered out using a specially designed filter.

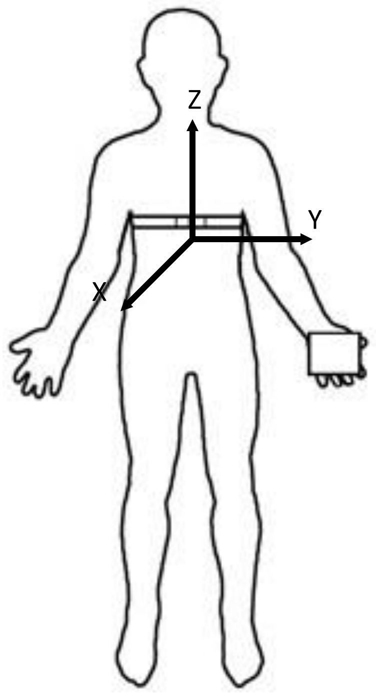


Figure 19: Placement of the accelerometer and the three axes involved in full body acceleration measurements. Summation of the axes is calculated as voltage$_x$ + voltage$_y$ + voltage$_z$.

This configuration is also capable of producing negative voltages when the wearer moves "negatively" in the direction of an axis. For example, when the wearer jumps up, the accelerometer outputs a positive voltage greater than zero-g voltage in the z-axis, but when the wearer falls back down, the voltage will swing below zero-g voltage. When zero-g voltage is subtracted from the output, the axial reading will become negative during the fall back down. Because movement in any direction requires energy expenditure, movement that is in a "negative" direction must be rectified before being summed with the other axes. If this is not done, a "negative" movement would subtract from the summed voltage and the total calculated movement would appear less than what it actually is.

Most human movement, as measured with an accelerometer placed on the torso, is between 0.8 and 5 Hz [30]. Any movement detected by the accelerometer that is outside this range is undesirable noise, such as vibrations from a car, or the movement of the chest during breathing. Aside from filtering noise, a cutoff at 0.8 Hz eliminates the DC offset, whose frequency is 0 Hz. In the previous MQP, this was achieved with a significant amount of analog components, including an accelerometer, three buffers, three high pass filters, three rectifiers, a summing amplifier, low pass filter, and analog-to-digital conversion (ADC). Many of these components were eliminated by completing the summation within a processor chip and then filtering the signal with a digital signal processor.

The initial design concept was to use a digital accelerometer and a digital signal controller (possibly a dsPIC from Microchip) to filter the signals and perform the rectification and summation in one chip. However, the dsPIC from MicroChip comes with many extraneous features, such as pulse width modulation (PWM) channels for motor controlling, because the product is geared more for complicated motor controlling rather than simple signal filtering. The dsPIC also costs more and is an unfamiliar platform.

During the research and part selection process, another device was identified that was simpler, less expensive, and specified for filtering: Quickfilter's QF4A512 Programmable Signal Converter. The QF4A512 is a 4-channel signal conditioner that can be programmed with a gain, cutoff frequencies, and analog-to-digital sampling frequency. Each channel can be programmed separately using the Quickfilter Pro software, although for this design, three channels with identical programming were used. The Quickfilter chip performed the DC offset elimination and filtering for each of the three signals before outputting them as 16-bit samples. The Quickfilter was programmed with a McClellan band pass filter with a pass band frequency of 0.5 to 8 Hz. The filter coefficient values were generated by the Quickfilter software and programmed using their software adapter and development board.

4.1.2.3 Microcontroller: TI MSP430F449

Quickfilter Technologies provides a code sample that interfaces the Quickfilter QF4A512 with Texas Instrument's MSP430F449. This code contains the functions to configure and run the QF4A512 as well as read and write to its EEPROM. The code had to be modified for the specific needs of the activity monitor, including adjusting the "main.c" file to process the acceleration signals and adding code for the other Universal Asynchronous Receiver Transmitter (UART) interface. The new code also eliminates some of the debugging features, such as testing point and LED toggling, whose peripherals will take up too much valuable printed circuit board (PCB) space.

The Quickfilter code example used all four channels on the QF4A512, whereas the accelerometer only required three, one for each axis. To save the time that reading an extra, unused channel will take, the code was also rewritten to check only the first three channels. The MSP430 was programmed to read 16-bit samples from each channel into a buffer, where the sample was then rectified and the DC offset removed.

When the samples were retrieved, a timer was initialized that counts ten seconds before generating an interrupt. During these ten seconds, the samples from the three Quickfilter channels were rectified and summed and a count was incremented to keep track of the number of samples read. When the interrupt occurred, the value of the samples accumulated in the sum was divided by the number of samples retrieved to calculate the average full body acceleration over a ten second period. This information was then used in the algorithm to estimate caloric usage. Figure 20 below represents the processes carried out by the electrical components of the signal acquisition module.

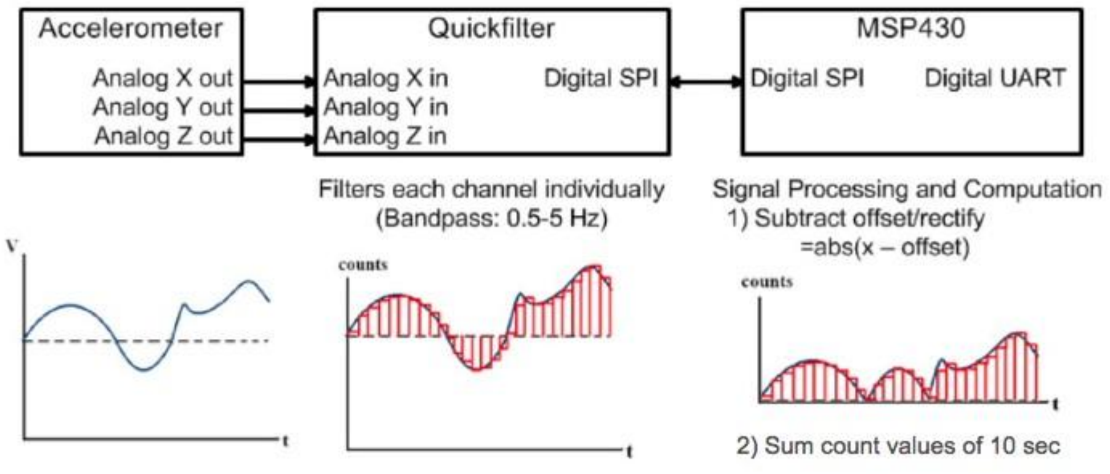


Figure 20: Block diagram of the electrical interactions between the accelerometer, Quickfilter chip, and the MSP430.

A ZigBee device called XBEE was used to wirelessly transmit the data to the user-interface module, where the modified Actiheart algorithm was implemented on another MSP430F449. The Zigbee device was suitable for this purpose because it is a low power, low cost wireless platform. The XBEE has a range of approximately 100 feet, which is more than enough for this application since the user will wear both devices. The data is transmitted via UART to the XBEE from the MSP430 in 8-bit chunks along with a stop and start bit.

4.2 Display Module

Signal processing of both the heart rate and full body acceleration signals was performed in order to convert the signals to more pertinent information. This signal processing allowed for not only the display of individually calculated parameters, but also for energy expenditure calculations to be displayed both continuously and cumulatively. These calculations and the display of their results were included on a separate module henceforth referenced as the display module. The display module contains a microprocessor chip which was programmed to perform all tasks relating to the module, such as the user interface, display of real-time data, and memory.

4.2.1 Microprocessor Selection

An MSP430F449 was also chosen as the microprocessor for the display module. MSP430 microchips are widely used in electrical engineering applications, especially in student design projects. The development board available for the MSP430F449 includes an LCD custom display and four buttons, which provides an acceptable user interface at this stage in this device design. It is as a result of this, as well as student familiarity with the MSP430 programming platform that this chip was selected for use in the device design.

The responsibilities of the MSP430F449 within the display module include: (1) reading accelerometer and heart rate signals, (2) outputting a count of both signals, (3) incorporating the ActiHeart algorithm to output instantaneous carbohydrate expenditure, (4) display of current heart rate, accelerometer count, carbohydrate expenditure rate, (5) cumulative carbohydrate expenditure, and (6) input of patient-specific parameters such as sex, age, and weight.

The block diagram shown in Figure 21 illustrates the order in which these functions would be performed by the micro processing chip.

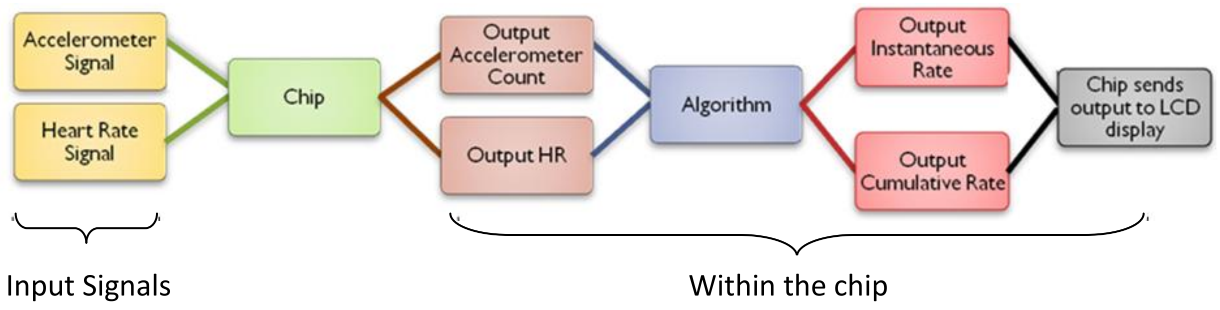


Figure 21: Block diagram of the device as seen by the microprocessor chip. The microchip receives signals from the accelerometer and the heart monitor then converts the signals into counts per unit time. Using those parameters in the Actiheart algorithm, the device will record cumulative carbohydrate expenditure and output instantaneous carbohydrate expenditure

4.2.2 Device States

In order to accomplish the tasks outlined in Figure 21, the device has been programmed to operate within two states: *parameter input state* and *analysis state*. In parameter input state the device works to gather the user information necessary to provide a patient-specific energy expenditure output to the display. In the analysis state, the device receives the signals from the signal acquisition module and performs signal processing operations to display appropriate heart rate, full body acceleration, cumulative energy expenditure, and energy expenditure per minute. The operations performed in these states are detailed below.

4.2.2.1 Parameter Input State

The device enters the parameter input state upon initialization. The user is prompted to enter first their gender (0 for male, 1 for female), then their age, and finally their weight. This is accomplished in each case by first sending commands to the LCD to display the words representing the necessary input. These are “GENDER”, ”AGE”, and “WEIGHT”, respectively. This occurs when the code below is executed:

```
if (displayName == 1) {
  if (inputtype == 0)
    writeWord("SEX");
  else if (inputtype == 1)
    writeWord("AGE");
  else if (inputtype == 2)
    writeWord("WEIGHT");
  else
    writeWord("");
}
```

In this case the variable ‘inputtype’ is defined previously in the code as an incrementing number associated with how many LCD operations have been completed since the program started. When programming in C, the number 0 (as in inputtype==0) is associated with the first iteration of a procedure. When this code is executed, the first operation of the LCD will be to display the word “SEX”, the second will be to display “AGE”, and the third will be to display “WEIGHT”.

The user then enters the number representative of their gender, age, or weight. Once the user presses ‘enter’ the data is stored as the variable associated with the LCD display command, and sent to the modified Actiheart algorithm for use in the analysis state. This is accomplished using the code below:

```
if (buttonPressed)
  {
    if (inputtype == -1)
      inputtype = 0;
    else {
      if (buttonPressed & BTN1)
         variablevalue = digits[0] + 10*digits[1] + 100*digits[2];
         if (inputtype == 0)
           sex = variablevalue;
         else if (inputtype == 1)
           age =  variablevalue;
         else if (inputtype == 2)
           weight = variablevalue;


         inputtype++;

         digits[0]= 0;
         digits[1]=0;
         digits[2]=0;
         variablevalue=0;
         displayName = 1;
      }
     }      else
       displayName = 0;
```

This code indicates that if a button has been pressed and the inputtype is 0, 1, or 2 (gender, age, or weight), the variable 'variable value' will be stored as the appropriate variable for the energy expenditure calculation. The 'variablevalue' operation takes the numbers entered by the user in the ones, tens, and hundreds place and calculates the correct number to store as the variable using Equation 1 below.

$$Equation\ 1 \qquad Variable\ Value = 100x + 10y + 1z$$

For example, if the user wanted to enter their weight as 134 pounds, the numbers entered as x, y, and z would be 1, 3, and 4, respectively. The variable value would then be calculated as shown below in Equation 2.

$$Equation\ 2 \qquad Variable\ Value = (100*1) + (10*3) + (1*4) = 100 + 30 + 4 = 134$$

The "if else-if" statements in this section determine whether or not the variablevalue should be stored as gender, age, or weight. So if a statement for inputtype==1 is executed, the number entered will be stored as age, and the if-else statement will be exited. This leaves the statement inputtype++; as the next section of the code to be executed. This simply informs the device that the inputtype now being analyzed is the inputtype after the previous one - or in the case of this example - inputtype ==2, weight. After the inputtype is changed the numerical values for the ones, tens, and hundreds places are restored to zero for the next input.

Once this process has been completed for all three of the variables the device exits parameter state and enters the analysis state.

4.2.2.2 Analysis State

During the analysis state the chip continuously receives signals from the accelerometer and heart rate monitor, converts them into counts per unit time, and displays the result. Results are displayed in a five-second cyclic manner. In this state, the cumulative carbohydrate value calculated using the parameter input values is available to the user on call via the second button. To reset the carbohydrate count to zero, the user may press the first button.

4.2.2.2.1 Heart Rate Calculation

Heart rate is defined as the number of heart beats that occur during one minute. These beats are observed by the QRS wave portion of the electrical signal generated by the heart. The most simplistic way of measuring heart rate would be to actually count how many times a person’s heart contracted within one minute, however, that is too time consuming to give worthwhile results. Alternatively, the actual heart rate can be calculated by counting the number of beats in a 15 second period of time, and multiplying that count by four to give the predicted output for a sixty second time period. This equation can be seen below in Equation 3.

$$Equation\ 3 \qquad Heart\ Rate\ = 4[N]_t^{t+15}$$

Unfortunately, this provides a somewhat large room for error if the detector misread a heartbeat. If something like noise were to cause the device to see a pulse that was not actually a result of a heartbeat, not only would one extra beat be added to the per 15 second rate, but four would be added to the per minute rate due to the multiplier. As a result of this a more accurate algorithm has been determined for heart rate calculation.

The algorithm for calculating heart rate divides the time between heart beat pulses sent from the Polar® strap into sixty seconds, to represent a beats-per-minute rate. This calculation is made within the Display Module. Every time the Display Module sees a pulse, the current time is recorded onto a variable and the difference between that time and the last time is calculated. The calculation for two heart beats can be seen below:

$$Equation\ 4 \qquad Heart\ Rate = \frac{60}{time_2 - time_1}$$

This equation was then expanded to the equation which can be seen in Equation 5 in order to gather data over a 15 second period of time, and output the average rate acquired from that data. In the equation this time period, called an epoch, is represented by the limits of *t* to *t+15*, and the average of the data is represented by dividing by N, the number of beats observed within the epoch.

$$Equation\ 5 \qquad Heart\ Rate = \left[\sum_{1}^{N} \frac{60}{time_2 - time_1}\right]_{t}^{t+15} N$$

This algorithm is extremely accurate in that the most recent heart rate can be obtained with every heart beat. This allows for more accurate measurement than the typical beats-per-minute calculation outlined in Equation 2. In this case, if an incorrect reading were to be made by the device, only two beats would be affected maximally – the first beat which included the misreading as the second time, and the next beat which included it as the first.

Below, the programming code for the calculation of Equation 4 can be seen.

```
if(next == 0)

{
  occurence[0] = time;
  next = 1;
}
else
{
  occurence[1] = time;
  dt = (occurence[1] - occurence[0]);
  HR = (int)(60/dt);
  occurence[0] = occurence[1];
  avgHR += HR;
  HRcnt ++;

  if (sex == 0 || sex == 1 && age != 0 && weight !=0)
    HR = (int)(AEEhr(AC, age, sex));
}
```

This code segment is recording the time of the first heart beat, calculating the difference between the two beats, and then calculating the heart rate via the designated heart rate calculation algorithm.

4.2.2.2.2 Full Body Acceleration Count

The full body acceleration count over a ten second period of time is calculated in the signal acquisition module and is transmitted to the display module using the UART capabilities available on MSP430F449 micro processing chips and a pair of ZigBee devices. The task of the display module in this case is to obtain the two 8-bit samples sent by the accelerometer, reorganize them back into the 16-bit number that they represent, and then convert the binary number into a decimal number. This is accomplished using the code below.

```
int cd = 0;
 while(1)
 {
   //Receive byte
   receive();
   unsigned int tempByte = RXBUF0;


   if (cd % 2 == 0) {
     INbyte = tempByte + 0x00;
     INbyte = INbyte << 8;
     INbyte |= firstByte;
   }
   else {
     firstByte = tempByte;
     cd = 0;
   }
   INbyte = 76;
   //display_byte(INbyte);

   cd++;
 }
```

The information from the accelerometer is sent over as a byte which enters the receive buffer, RXBUF0. This byte is stored as "tempByte" each time. To determine what operation to do with the byte, it must be determined which byte in the sequence is being received – the first or the second. This is completed by defining the samples as the odd or even received byte. In the "if" statement above, the program says that if the sample is even, then the logical or of the tempByte is completed, and the byte is shifted. The "else" portion of the "if" statement indicates that if the sample is an odd sample, it is the first byte received, and it a logical and should be performed.

In order to display an accurate measure of counts per minute, this number contained within the byte is multiplied by six, to represent a 60 second period of time. This value is sent to the full body acceleration display as well as the modified Actiheart algorithm which uses it to calculate energy expenditure.

4.2.2.2.3 Energy Expenditure

Energy Expenditure Calculation

During the previous MQP project, it was determined that the best method to calculate energy expenditure from heart rate and accelerometer counts was developed by the ActiHeart study completed in 2007. This set of equations uses the subjects' age and gender in addition to the heart rate and accelerometer counts to determine how many kilocalories per kilogram minute are burned. The equations are located in the following table:

Table 4: Actiheart Activity Algorithm

Activity algorithm (AEE, kcals kg^{-1} min^{-1})	
<133 counts min^{-1}	$\frac{(0.203 \cdot 133) - (0.75 \cdot age) + (83 \cdot sex) + 46}{133} \cdot \frac{counts}{4186.8}$
≥133 counts min^{-1}	$\frac{(0.203 \cdot counts) - (0.75 \cdot age) + (83 \cdot sex) + 46}{4186.8}$
Heart rate algorithm (AEE, kcals kg^{-1} min^{-1})	
HRaS <23 beats min^{-1}	$\frac{(5.95 \cdot HRaS) + (0.23 \cdot age) + (84 \cdot sex) - 134}{23} \cdot \frac{HRaS}{4186.8}$
HRaS ≥23 beats min^{-1}	$\frac{(5.95 \cdot HRaS) + (0.23 \cdot age) + (84 \cdot sex) - 134}{4186.8}$
Combined activity and heart rate algorithm (AEE, kcals kg^{-1} min^{-1})	
<25 counts min^{-1} and HRaS <23 beats min^{-1}	$(0.1 \cdot AEE_{HR}) + (0.9 \cdot AEE_{accelerometer})$
HRaS between 23 and 80 beats min^{-1}	$(0.5 \cdot AEE_{HR}) + (0.5 \cdot AEE_{accelerometer})$
>25 counts min^{-1} and HRaS ≥80 beats min^{-1}	$(0.9 \cdot AEE_{HR}) + (0.1 \cdot AEE_{accelerometer})$

Parameters such as resting heart rate, sex, and age of an individual are incorporated into the Actiheart algorithm yields a kilo caloric expenditure at a given accelerometer count and heart rate [25].

Since the rate of caloric expenditure varies with activity level, the Actiheart algorithm determines which parameter best models the expenditure rate for that particular activity through prioritization levels.

Accelerometer Prioritization:

During events when there are low accelerometer counts, coupled with high heart rate measurements, Actiheart assumes that the high heart rate values are caused by stress or chemical imbalances and models the energy rate of the individual using an algorithm that emphasizes the accelerometer based algorithm.

Equal Prioritization:

When there is moderate rise in heart rate measurements, neglecting the accelerometer count value, Acitheart equally distributes the energy expenditure modeling between the heart rate based algorithm and the accelerometer based algorithm.

Heart Rate Prioritization:

During times when both heart rate values and accelerometer counts are significantly high, Actiheart believes the heart rate based algorithm best models the rate at which energy is expended, and shifts emphasis of the expenditure modeling more toward that algorithm and less toward the accelerometer based equation.

The device, ultimately, outputs kilocalories per kilogram minute, an uncommon and generally not a practical number for diabetic individuals to monitor their activity. Since the device will be used primarily for a single user, the individual will be required to input their weight, in kilograms, in order for the device to output real-time caloric expenditure.   To make the device more user friendly, another process is needed to convert kilocalories expended per minute to carbohydrates expended per minute.

Kilocalories to Carbohydrate Conversion

In order to calculate the grams of carbohydrates, composing expended calories, the volumetric respiratory exchange ratio (RER) of carbon dioxide emission versus oxygen consumption needed to be known. RER is calculated by dividing the amount of carbon dioxide exhaled by the amount of oxygen inhaled. Carbohydrates are known to be the exclusive source of caloric expenditure during complete anaerobic exercise, when carbon dioxide emitted from the body equals the amount of oxygen consumed [source]. As the activity level becomes more aerobic, when carbon dioxide emissions are less than oxygen consumption, calories expended consist more of proteins and fat and less of carbohydrates. For each section of the Actiheart equation, a corresponding RER value was implemented to convert those outputted calories into carbohydrates. These RER values were determined from a study conducted in 2000 [31]. The study determined relationships between volumetric oxygen and energy expenditure levels. The conclusions are shown in Figure 22 below.   The higher the energy output, the higher the RER and the more carbohydrates are expended.

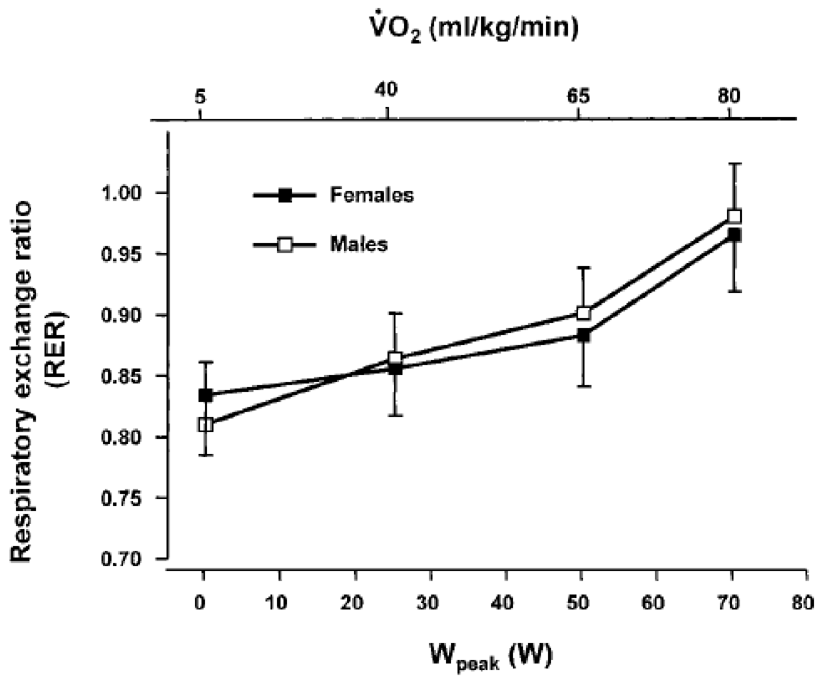


Figure 22: Relationship between volumetric oxygen and energy expenditure levels which allows for the calculation of carbohydrates from caloric expenditure.

Correlating the activity level with respiratory exchange ratios allow for caloric expenditure to be translated into carbohydrates which are values that are better suited for glucose management [31].

Table 5: Respiratory Exchange Ratios determined by study in 2000 which determined relationships between volumetric oxygen and energy expenditure levels.

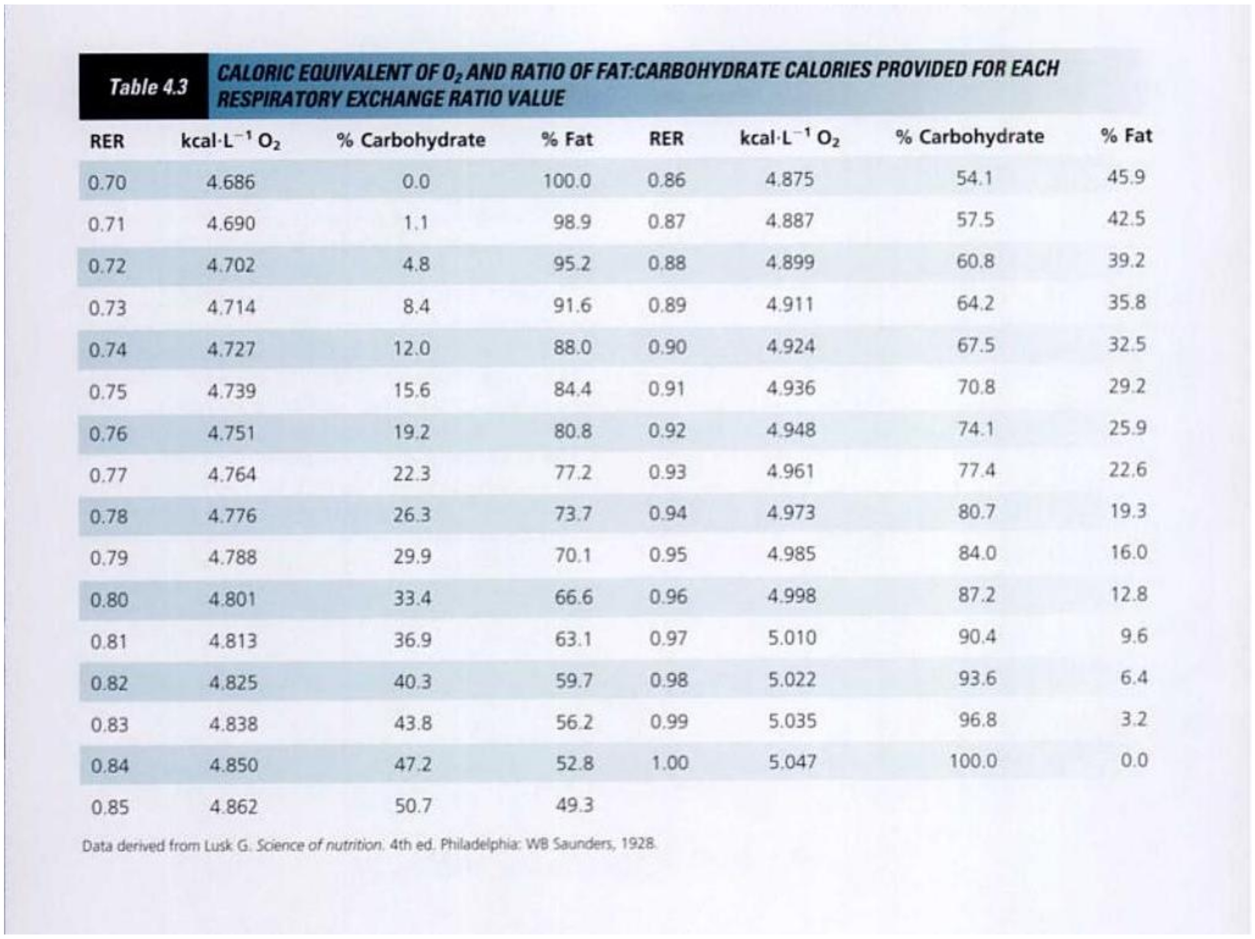


Table 4.3 ***CALORIC EQUIVALENT OF O_2 AND RATIO OF FAT:CARBOHYDRATE CALORIES PROVIDED FOR EACH RESPIRATORY EXCHANGE RATIO VALUE***

RER	kcal·L^{-1} O_2	% Carbohydrate	% Fat	RER	kcal·L^{-1} O_2	% Carbohydrate	% Fat
0.70	4.686	0.0	100.0	0.86	4.875	54.1	45.9
0.71	4.690	1.1	98.9	0.87	4.887	57.5	42.5
0.72	4.702	4.8	95.2	0.88	4.899	60.8	39.2
0.73	4.714	8.4	91.6	0.89	4.911	64.2	35.8
0.74	4.727	12.0	88.0	0.90	4.924	67.5	32.5
0.75	4.739	15.6	84.4	0.91	4.936	70.8	29.2
0.76	4.751	19.2	80.8	0.92	4.948	74.1	25.9
0.77	4.764	22.3	77.2	0.93	4.961	77.4	22.6
0.78	4.776	26.3	73.7	0.94	4.973	80.7	19.3
0.79	4.788	29.9	70.1	0.95	4.985	84.0	16.0
0.80	4.801	33.4	66.6	0.96	4.998	87.2	12.8
0.81	4.813	36.9	63.1	0.97	5.010	90.4	9.6
0.82	4.825	40.3	59.7	0.98	5.022	93.6	6.4
0.83	4.838	43.8	56.2	0.99	5.035	96.8	3.2
0.84	4.850	47.2	52.8	1.00	5.047	100.0	0.0
0.85	4.862	50.7	49.3				

Data derived from Lusk G. *Science of nutrition*. 4th ed. Philadelphia: WB Saunders, 1928.

The Actiheart algorithm was modified appropriately according to this information. Equation 6 below shows the modifications. The original Actiheart algorithm was divided into three branches. In the first branch, activity is mostly anaerobic and emphasis is placed on accelerometry counts. For this branch the estimated carbohydrate percentage of total energy expenditure is relatively low, around 0.45. In the second branch, activity is equally aerobic and anaerobic and the equation relies on both. In this case, carbohydrate percentage is estimated to be 0.62. In the third branch, activity is mostly aerobic and the energy expenditure relies heavily on heart rate. In this instance, carbohydrate percentage is estimated to be 0.87. Weight is included as a factor because the more a person weighs, the more carbohydrates are burned through metabolic processes.

Equation 6:

<23 HRaS and <25 counts min^{-1} : ((0.1*AEE HR)+(0.9*AEE Acc))*(weight)*0.45

23<HRaS<80: ((0.5*AEE HR)+(0.5*AEE Acc))*(weight)*0.62

80<HRaS and 25<counts min^{-1}: ((0.9*AEE HR) +(0.1*AEE Acc))*(weight)*0.87

This equation was used in the device to provide the user with carbohydrate expenditure, which in turn could aid in required dietary and insulin intake modifications.

Programming of the modified Actiheart Algorithm

The modified version of the Actiheart algorithm in Equation 6 was programmed into the MSP430449 to calculate values and display the results in real-time on screen. Both the heart rate and full body acceleration counts per minute were sent to the algorithm to output a rate of carbohydrate expenditure per minute. In order to allow for a cumulative count of the carbohydrates expended over time, the result of the algorithm is divided by 6 at each input to represent the value accumulated over a period of 10 seconds. This value is stored, and added to the next incoming value. This operation is completed continuously while the device is in use.

4.2.3 User Interface

The user interface was simplistically designed due to the need for the activity monitor's continuous use. Several measures were made in order to help the user enter their patient-specific information and call up their heart rate, full body acceleration, and energy expenditure with limited effort. The module used for this system is shown below in Figure 23.

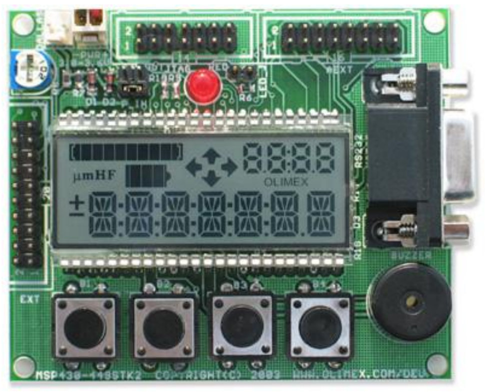


Figure 23: MSP430F449 prototype development board. The four buttons shown are referenced through the text as buttons 1-4, from left to right.

Numerical input is completed on a button-to-button basis. The 2nd, 3rd, and 4th buttons, shown in Figure 23, represent the numerical input places. Originally, the device was programmed to scroll through 0-9 on a rotating basis for each place, however, it was decided that setting limits for each input would make the device easier to use. The limits employed were as follows:

Gender Input - The limit was set to 1. This way if the user is male and accidentally enters the number '1' for gender, pressing the button again will return the value to 0.

Age Input - The maximum age value able to be entered by the user is 129 years of age. The oldest person that ever lived was 22. Added some years to give flexibility just in case)

Weight Input - The maximum value remains at 999 lbs so as to avoid discrimination. Describe in better detail.

These limits were implemented by simply programming each of the placeholder buttons for input type case. For example, if the age is being entered, the input type is "1", and thus button 2 is programmed to scroll to a maximum value of 1, button 3 is programmed to scroll to a maximum value of 9 if 0 was entered on button 2, and only a value of 2 if 1 was entered, and button 4 was programmed to scroll through all 9 values regardless.

Each of these values is stored as its respective variable by the user pressing the first button, which represents 'enter'. This data is then sent to the modified Actiheart algorithm for use in the analysis state.

During the analysis state the user is able to see heart rate and full body acceleration on a rotating basis, and able to call cumulative carbohydrate values and carbohydrate expenditure rate to screen by pressing the second and third buttons, respectively. The results of cumulative carbohydrates display as "CG" for "cumulative grams", and the results of the carbohydrate expenditure rate display in crams per minute, which is represented on screen as "GPM". If the user were to want to clear the cumulative carbohydrate count, pressing button 1 would return the count to zero. This feature allows the user to determine how many carbohydrates are burned during a specific period of physical activity.

5 Testing

To ensure the overall device operates within specified parameters, several validation tests are needed. First tests on the individual components were completed to ensure proper function. Then testing on the assembled device needs to be done. This testing was intended to be done in two parts: calibration of the algorithm and validation of the conversion to carbohydrates. Our device used a different accelerometers than those used by Actiheart, thus we must determine the scaling factor of the algorithm in order to get the most accurate calculation. In this light, we must correlate certain accelerometer counts with carbohydrate usage.

5.1 Benchmark Testing

Sensor Module

Several studies have been completed to assess the accuracy of Polar straps to determine heart rate. A study done at North Dakota State University compared seven heart rate monitors, including two different Polar devices, against ECG readings. They found that at all but the very highest activity levels, i.e. professional sprinters, the Polar devices were incredibly accurate [31]. A study conducted by the American College of Sports Medicine also found that the Polar heart rate monitor was a good choice to measure heart rate [32].

The Quickfilter filter design software, given the filter parameters, successfully generated a 0.8 to 5Hz McClellan band pass filter. The software provides a Fast Fourier Transform (FFT) of the expected filter behavior. However, this FFT is only an idealized projection, so the filter was lab tested to be sure of its performance. The Quickfilter software has the ability to read the output of the chip and generate a real-time FFT. A QF4A512 filter chip was programmed with the filter coefficients and connected to the computer through the programming adapter attached the QF4A512 development board. A white noise audio file was applied to the channel one BNC input; the signal was filtered by the programmed chip and displayed in the Quickfilter software. The fast Fourier transform of the chip's filter is shown below in Figure 24.

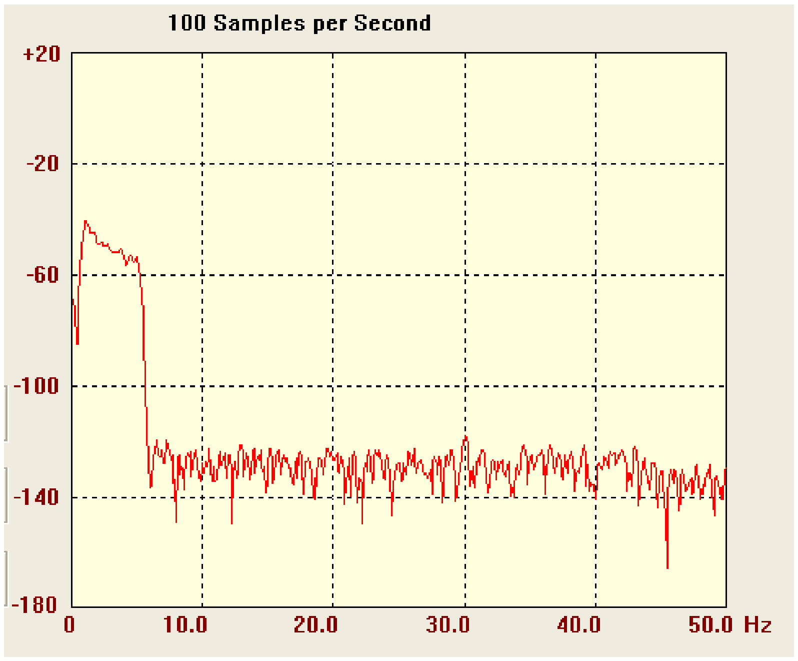


Figure 24: Fast Fourier Transform of QAF4A512 Filter of white noise audio file.

The resulting FFT of the band pass filter demonstrates an acceptable filter behavior for our device requirements. It filters out frequencies outside of the approximate range 08 to 5Hz with a suitable roll-off.

Display Module

The heart rate calculation used in the signal processing portion of the display module was tested in the laboratory by performing electrical stimulation tests with alligator clips. In these electrical stimulation tests, the design team simply applied an appropriately sized voltage signal to the pin on the MSP430F449 which would be receiving the digital pulses from the Polar® strap. In other words, each time the alligator clip touched the pin, this simulated a heart rate pulse as detected by the strap.

The alligator clips were supplied a voltage of 3V. The clips were applied to the appropriate pin over 10 second intervals. In the first trial, the pin was touched as many times as possible for 10 seconds, representing maximum heart rate. The maximum heart rate displayed was 255 beats per minute (BPM). In the next trial, the pin was touched 8 times over a 10 second interval. The time between electrical stimulations was measured and used to determine heart rate. The display recorded 60 BPM while the value calculated by the experimenter was 63 BPM. During the third trial the pin was touched 4 times with an expected display value of 24 BPM, while the display value was 21 BPM. There is some error between the expected result and the displayed result as there may have been some human error while recording the time between each electrical stimulation to calculate the expected results, as well as in applying the electrical stimulation. If full contact was not made between the alligator clip and pin, a signal would not have been observed by the device.

The method of measuring the time in between stimulations to calculate heart rate has been shown to be accurate. Figures 25-27 demonstrate the different rates of electrical stimulation while Table 6 shows the results of the stimulation testing. The device was shown to be fairly accurate, and is most likely more accurate than recorded, considering human error.

Table 6: Results from heart rate calculation and display testing on the MSP430F449.

Heart Rate Calculation and Display Testing				
Test Number	**Frequency of Application**	**Calculated Result**	**Displayed Beats-Per-Minute**	**Δ(Measured-Predicted) [BPM]**
1	15+	255	255	**0**
2	8	24	21	**3**
3	**4**	**60**	**63**	**3**

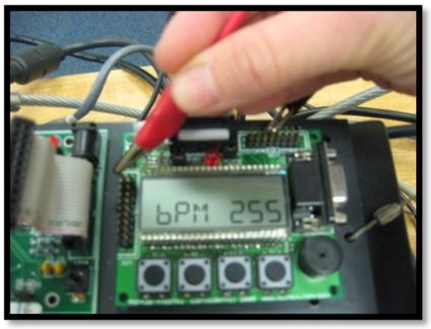


Figure 25: Maximal heart rate acquired from a frequency application of 15+.

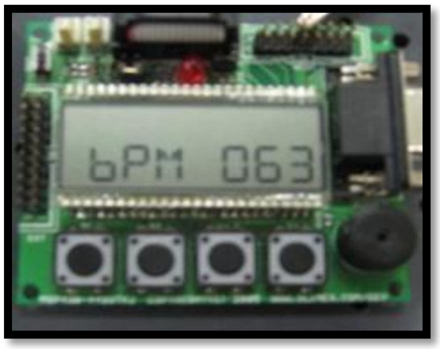


Figure 26: Heart rate acquired from eight electrical stimulations.

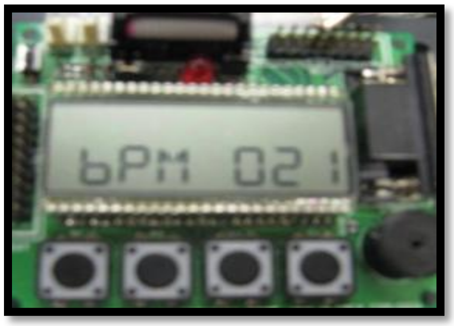


Figure 27: Heart rate acquired from four electrical stimulations.

The ability of the display module to receive the full body acceleration values from the signal acquisition module was tested by writing the code used to transmit data into the display module and feeding the program values. The goal was to receive the 2 bytes of information, reverse the order, and translate the values into the correct decimal number. In the test used, a screen display of "YES" indicated that the test worked, whereas a display of "NO" indicated that the conversion had not worked.

Initially the program resulted in a "NO" response from the system, however the design team learned that this was due to an issue with the address being used to store the information. Once the address was changed appropriately the test displayed a "YES" on screen (shown below in Figure 28).

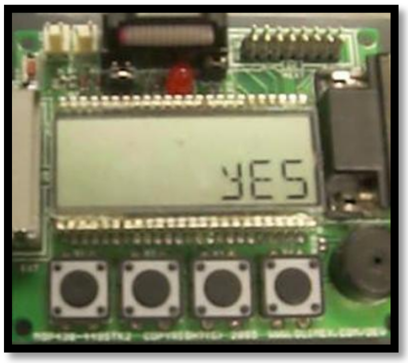


Figure 28: A "YES" resulting from correctly deciphering a transmitted value.

Finally, bench tests were performed in order to ensure that the programmed modified Actiheart algorithm was functioning properly. This was verified by writing values for heart rate and full body acceleration count into the display module code, collecting the output, and comparing it to calculations made by hand.

In order to represent practical energy expenditure values with these tests, values from the Actiheart study temporarily stored in the device. The Actiheart acceleration and (heart rate) HR values shown below in Table 7 represent the average results for full body acceleration and heart rate per minute gathered from the Actiheart whilst performing the listed activities. An analysis of the accuracy of the display output for the display module was completed by comparing the values obtained by longhand calculation to those displayed on screen. These values are also available in Table 8. Several other activities were analyzed, but those chosen were used in this test as they represent some activities which will be used in device testing. In order to better represent the progression of energy expenditure, values for slow and fast walking and running were included as well. A comparison of these results can be seen in Figure 29.

Table 7: Actiheart acquired values for full body acceleration counts/min and heart rates for several activities were programmed into the device to receive a projected carbohydrate expenditure output for these activities. The values obtained from by-hand calculations of the algorithm are shown in the column for calculated carbohydrates.

Activity	Actiheart Acceleration (counts min-1)	Actiheart HR (BPM)	Device Projected Carbohydrates (g)	Calculated Carbohydrates (g)
Lying	0.1	66	0	0
Filing Papers	5	83	0	0.60
Washing Dishes	10.6	104	1	1.7
Slow Walk (82 m/min)	101.9	105	2	2.0
Fast walk (avg 103 m/min)	597.1	103	2	2.9
Ascending/descending stairs	351	130	7	7.7
Slow run (157 m/min)	1776	155	11	11.5
Fast run (191 m/min)	1908.4	170	13	13.5

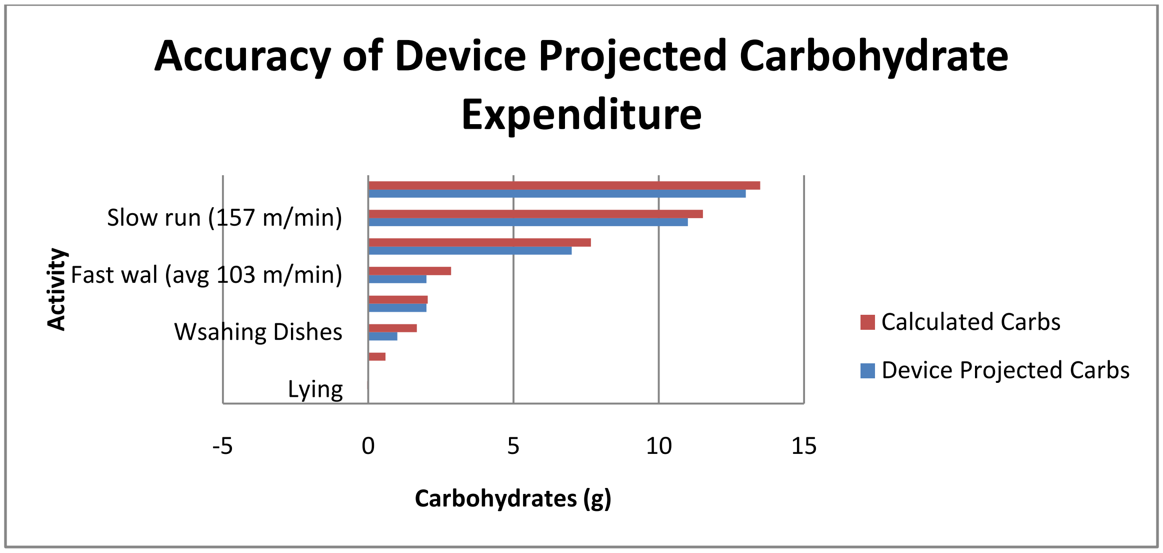


Figure 29: Bar graph representing the accuracy of the projected carbohydrate expenditure by the device display versus the calculated carbohydrate output.

The carbohydrate values calculated by hand appear to be slightly larger than those shown on screen in the device. This is as a result of double to integer conversion for display on screen. In order to remedy this, future devices could have a more accurate LCD display which was capable of showing decimals, or the program could be written to round up – which would result in a larger carbohydrate expenditure projected by the device than by calculation. The long term cumulative values would not suffer greatly as a result of this difference, because the programming does not actually drop the values when adding the results over time. Thus the cumulative energy expenditure value would at most be off by a fraction of a gram.

5.2 Human Subject Testing I

IRB Approval

In order to complete testing on human subjects, approval must be obtained from WPI's Institutional Review Board. This can be a lengthy involved process, however given that our device is an improvement and expansion of a previous project, we only had to obtain a continuation of the IRB approval given last year. An IRB application was filled out and filed with the Office of Sponsored Research along with a copy of the intended procedures, case report form, and subject consent form. As a requirement of approval, all group members were required to complete online training through the National Institute of Health (NIH) entitled Protecting

Human Research Participants and submit certificates of completion. Copies of these documents can be found in Appendix A, along with the letter granting IRB approval.

Procedures

The purpose of this project was to improve upon the 2007-2008 design. With this in mind it was decided that device testing should adhere to the same procedures used last year. In turn, these tests were derived from the original Actiheart study in order to better compare the device to Actiheart. However, in order to encourage more cooperation from testing subjects and keeping in mind location constraints not all of the tests used to determine the accuracy of Actiheart were considered. The tests that were performed follow:

- Standing
- Sitting
- Ascending/Descending Stairs
- Washing dishes
- Filing Papers
- Sweeping the floor
- Slow walk (3.6 mph)
- Fast walk (4.8 mph)
- Slow jog (5.6 mph)

The tests that were not included in our study were cycling, racquetball, basketball, lawn mowing, and leaf raking, among others. It was felt that having test subjects complete these tests would be very time consuming and introduce more risk. Given that our test subjects are all undergraduate students with a full academic schedule who are volunteering their time for this study, we made the decision to respect their time and only include those tests which could be completed in any classroom or on a treadmill. Overall these tests were proven to take less than two hours, even with location changes.

Potential subjects were asked to volunteer by members of the group. Those who responded favorably were included, with the exception of those who did not meet the inclusion criteria. The inclusion criteria were that the subject did not have asthma, irregular heartbeats, or any other medical condition that may be exacerbated by testing. This was not only for their safety and well being, but to also limit the liability of WPI. 38 individuals initially volunteered and were given the consent form to review and make their decision.

IRB protocols require that each subject be given a week to fully consider the possible risks of participating in the study before they sign the form and are officially enrolled in the study. Due to several modifications made to the device, testing was delayed, and several subjects withdrew their interest in volunteering.

At the beginning of the testing session the subject's heart rate was obtained while in a supine position. From this it was possible to calculate the sleeping heart rate needed for the energy expenditure algorithm. To verify that the device was accurately outputting the heart rate, the subjects pulse was taken by the student investigator by hand. Each individual test was performed for five minutes. This was to ensure a steady heart rate would be reached and that consistent values would be obtained for all subjects. After each period of 5 minutes, the total carbohydrate expenditure was recorded, and reset. These values were used as comparison point between subjects only, as no other device outputs energy expenditure in carbohydrates. In addition to carbohydrate usage, heart rate and accelerometer counts were also recorded to use as comparison points between devices. For each subject a case report form was completed. These can be found in Appendix A. At the completion of this study these forms will be filed with the Office of Sponsored Research and stored there for five years.

5.3 Human Subject Testing II

This testing period was to include metabolic testing to verify the carbohydrate conversion we implemented and make any adjustments needed. These tests would need to be done in a metabolic lab in order to use advance equipment. Possible locations were identified within central Massachusetts, however upon contact it was determined that all laboratory time was spoken for through the summer of 2009, long pass the conclusion of this project. A detailed explanation of the various methods possible for these tests can be found in Chapter 2 of this report.

5.4 Results

These results are ongoing, but once complete will be compared to a study done in 2007 to determine the accuracy of Actiheart and the 2007 – 2008 device.

6 Conclusions and Recommendations

The following chapter discusses how energy expenditure was calculated, how changes made to the previous major qualifying project improved the device, and the accuracy of the activity monitor. We also present several recommendations to further improve this device to increase its market potential.

6.1 Real-time Energy Expenditure Calculations

- Heart rate and fully body acceleration can be measured in real-time and can be used in combination to determine energy expenditure via the modified Actiheart algorithm. Outputting energy expenditure in grams of carbohydrates is useful for diabetic individuals due to its correlation with insulin levels in the body.

6.2 Improvements to Previous Major Qualifying Project

- Utilizing the various techniques of digital signal processing, signal detection with sensor arrays, transmitting data wirelessly, and custom-designed circuitry enclosures makes the device easy and simple to use as well as aesthetically pleasing.

6.2.1 Digital Signal Processing

- Acceleration signals are processed in the digital domain using a programmable filter chip. Digitally processing signals reduces the size of the circuitry and is simpler to implement than analog signal processing.

6.2.2 Signal detection using sensor arrays

- The Polar® heart rate detection system accurately detects the electrical signals of the heart using a sensor array, unlike the traditional approach of the electrode system.

- Heart rate signals detected by the Polar® system are wirelessly transmitted to a receiver chip, eliminating the need for electrical leads to carry the signal to be processed.

6.2.3 Wirelessly transmitting measured data

- Sending the heart rate and acceleration data from the signal acquisition module to the display module wirelessly decreases the size of the device as well as makes it more comfortable to wear and easy to use.

6.2.4 Custom made housing

- Custom housing built according to the specific dimensions of the device is sleek and aesthetically pleasing. The signal acquisition module housing is attached to the Polar® device, and the display module is handheld or can be placed in a pocket. This enables diabetic individuals to discreetly wear and use the activity monitor.

6.3 Recommendations

The following section discusses our recommendations for future work after the project was completed. These recommendations for the device, which include size reduction, decreased power consumption, more extensive testing, and data storage capabilities, are intended to improve the marketability of the device.

6.3.1 Size reduction in both signal acquisition and display modules

To reduce the size of the signal acquisition device, a smaller microcontroller such as the MSP430F149 can be used. In order to use this microcontroller the Quickfilter software code must be modified to accommodate the MSP430F449 rather than the current MSP430F449.

Size reduction in the display module can be accomplished by using a smaller, custom made LCD screen and button system. The current LCD and buttons is part of a development board specific to MSP430 devices. Smaller LCD and buttons will increase the compactness of the device and make the device more discrete to use. Ultimately, this upgraded system could be incorporated via a watch-like design for discreetness and ease of use.

6.4.2 Decrease power consumption

With the current power consumption of two 3V batteries, the device will continue to work for two to three hours without being charged if left on during this entire time period. In the future, power consumption should be decreased so that the user can leave the device on throughout the day to track their energy expenditure. Ideally, the batteries would only need to be changed after several weeks. This increased battery life will make the device easier to use and more accurate as it will calculate energy expenditure for longer amounts of time without needing to be charged.

6.4.3a Complete metabolic testing of the device

We recommend testing the accuracy of the number of carbohydrates burned according to our device to the number of carbohydrates that are actually burned. This type of testing can be done utilizing a gas exchange analyzer, or metabolic cart, in a metabolic lab such as one at UMASS Medical in Worcester, MA. The values calculated by our device should be quantitatively compared to those found during the metabolic testing.

6.4.3b Improving the carbohydrate conversion algorithm

In order to more accurately estimate carbohydrates being burned, real-time volumetric oxygen intake and carbon dioxide output values are needed. Since the current algorithm, converting calories into carbohydrates, does not take any respiratory measurements, it completes the conversion using estimates based off statistical respiratory values per activity level. To increase the accuracy of this conversion, the algorithm must be altered to tailor more towards the user. This can be achieved by looking into a correlation between age, gender and weight with respiratory exchange.

6.4.4 Add data storage features

Adding a storage feature to the device such as a Secure Digital (SD) card will enable the user to record the data from the activity monitor for later use. An SD card is a small, low power, flash-based storage device that could be used to save energy expenditure, heart rate, and full body acceleration information. Once inserted into the activity monitor it will store the data and can be removed and inserted into a computer for further analyzing. The current device PCB was manufactured to interact with a MicroSD, so the work required to incorporate this feature depends only on programming the MSP430F449.

6.4.5 Utilize specific inputs to output insulin dosages

Outputting specific insulin dosages will be more helpful to diabetic individuals than energy expenditure in carbohydrates alone. If glucose levels could be measured and stored as an input to the device, an algorithm could be used to calculate the specific insulin dosages needed.

References

[1] "Diabetes." Public Health. Kansas City, Missouri: Community Health Assessment, 2006.

[2] Dym, Clive L. and Patrick Little. Engineering Design: A Project-Based Introduction. 2nd edition. Wiley. 2003

[3] S Brage, N Brage, PW Franks, U Ekelund, M Wong, LB Andersen, K Froberg, and NJ Wareham, "Reliability and validity of the combined heart rate and movement sensor Actiheart," *Eur. J. Clin. Nutr.* 59: 561-570, 2005.

[4] American Diabetes Association. <http://www.diabetes.org>. Accessed 26 April 2009.

[5] Fox, Stuart Ira. Human Physiology. 10 edition. McGraw – Hill, New York. 2007.

[6] Jovanovic-Peterson, Lois, MD. "ExCarbs, a New Way of Control Through Exercise" *Diabetes Health*. July 1, 1995. < http://www.diabeteshealth.com/read/1995/07/01/397/excarbs-a-new-way-of-control-through-exercise/> Accessed 26 April 2009.

[7] Harvard School of Public Health. "Carbohydrates: Good Carbs Guide the Way". Accessed 10 October 2008 < http://www.hsph.harvard.edu/nutritionsource/what-should-you-eat/carbohydrates-full-story/index.html>

[8] Carpi, Anthony Ph D. "Carbohydrates". Vision Learning. Accessed 12 October 2008. < http://www.visionlearning.com/library/module_viewer.php?mid=61>

[9] American Diabetes Association. "All About Diabetes: Overview". Accessed 10 October 2008 < http://www.diabetes.org/about-diabetes.jsp>

[10] Sayers, George. "Insulin." *Encyclopedia Americana*. 2008. Grolier Online. 13 Oct. 2008 <http://ea.grolier.com/cgi-bin/article?assetid=0215870-00>.

[11] Bassett, Steven. "Anatomy and Physiology." Hoboken, NH: John Wiley & Sons, Incorporated, 2005. 211.

[12] Rea, Caroline RS. "Diabetes Health Center: Blood Glucose" July 25, 2007. Accessed 13 October 2008. < http://diabetes.webmd.com/blood-glucose?page=4>

[13] "Hypoglycemia." *Encyclopedia Americana*. 2008. Grolier Online. 13 Oct. 2008 <http://ea.grolier.com/cgi-bin/article?assetid=0211090-00>.

[14] Rizza, Robert A. "Diabetes Mellitus." *Encyclopedia Americana*. 2008. Grolier Online. 13 Oct. 2008 <http://ea.grolier.com/cgi-bin/article?assetid=0126230-00>.

[15] Tuch, Bernard. "Diabetes Research: A Guide for Postgraduates." London, UK: CRC Press, 2000. 1-14.

[16] Chen, Yue and Yang Mao. "Obesity and Leisure Time Physical Activity Among Canadians" Preventative Medicine 42.4 (2006): 261-265.

[17] M.I. Harris, "Frequency of blood glucose monitoring in relation to glycemic control in patients with type 2 diabetes", Diabetes Care 24 (2001) 979-982

[18] Kinnal, Elizabeth, Towa Matsumura, Shannon O'Toole, Nathan Occhialini. "An Activity Monitor for Diabetic Individuals". 2008. Worcester Polytechnic Institute Major Qualifying Project.

[19] Arsand, E., et al. "A System for Monitoring Physical Activity Data Among People with Type 2 Diabetes." eHealth Beyond the Horizon. 2008. 14 Oct. 2008 http://www.hst.aau.dk/~ska/MIE2008/ParalleSessions/PapersForDownloads/02.C&HBeH/SHTI136-0113.pdf

[20] "Garmin Forerunner 305." *Consumer Search*. 2009. 28 Apr 2009 <http://www.consumersearch.com/heart-rate-monitors/garmin-forerunner-305>.

[21] "Rock and Run." *Nike+Ipod*. 2009. Apple. 28 Apr 2009 <http://www.apple.com/ipod/nike/run.html>.

[22] Maliszewski, A., et al, "Validity of the Caltrac Accelerometer in Estimating Energy Expenditure and Activity in Children and Adults." *PES* 3(2)May 1991 Web.28 Apr 2009

[23] Eston, R., Rowlands, A., and Ingledew, D.. "Validity of heart rate, pedometry, and accelerometry for predicting the energy cost of children's activities." *J Appl Physiol* 1998 84:362-371. Web.28 Apr 2009.

[24] Image: www.salusa.se

[25] Crouter, SE, JR Churilla and DR Bassett. "Accuracy of the Actiheart for the Assessment of Energy Expenditure in Adults." European Journal of Clinical Nutrition (2007)

[26] Brown, Stanley P., Wayne C. Miller and Jane M. Eason. Exercise Physiology: Basis of Human Movement in Health and Disease. Baltimore: Lippincott Williams & Wilkins, 2006

<http://books.google.com/books?ct=result&id=gg4Xn4aGaBQC&dq=carbohydrate+expenditure+labs&ots=RrK2gSxrYz&pg=PA582&lpg=PA582&sig=ACfU3U0dMFpnaIe3K5tYjnNgK7zbFaS3Ug&q=table+4.3#PPA87,M1>

[27] Schoeller, D.A. and E. van Santen. "Measurement of energy expenditure in humans by doubly labeled water method" *Journal of Applied Physiology* 1982. 53:955-959

[28] Livingstone, M Barbara E, et al. "Simultaneous measurement of free-living energy expenditure byu the doubly labeled water method and heart-rate monitoring" *The American Journal of Clinical Nutrition* 1990; 52:59-65

[29] Riddell M, Perkins B. "Type 1 Diabetes and Exercise: Using the Insulin Pump to Maximum Advantage". Can J Diabetes. 2006; 30:72-79.

[30] Bouten, Carlijn, Karel Koekkoek, Maarten Verduin, Rens Kodde, and Jan Janssen. "A Triaxial Accelerometer and Portable Data Processing Unit for the Assessment of Daily Physical Activity." IEEE Transactions on Biomedical Engineering 44 (1997): 136-47.

[31] Terbizan, Donna J, Brett A Dolezal, and Christian Albano. "Validity of Seven Commercially Available Heart Rate Monitors." *Measurement in Physical Education and Exercise Science*. Vol. 6 Issue. 4 February 2002. Pg. 243 – 247 <http://www.informaworld.com/smpp/content~content=a785828031~db=all~order=page>

[32] Gamelin, Francois Xavier, Serge Berthoin, and Laurent Bosquet. "Validity of the Polar S810 Heart Rate Monitor to Measure R-R Intervals at Rest." *Medicine & Science in Sports and Exercise*. 38(5):887-893, May 2006. <http://www.acsm-msse.org/pt/re/msse/abstract.00005768-200605000-00013.htm;jsessionid=J9mc4B0S4PynRJyJSZ0MxSNJK0hH4hklynxXCdLLW6mykJhGC9JJ!-1862535748!181195628!8091!-1>

Appendices

A – Weighted Objectives Calculations

Table 8 Pair-wise comparison chart for Cost-Effective Sub-Objective

Cost Efficient (10.91%)	Minimal Parts	Parts Easily Obtainable	Easy to Manufacture	Low Shipping Weight
Minimal Parts	* * *	0	0.5	1
Parts Easily Obtainable	1	* * *	0.5	1
Easy to Manufacture	0.5	0.5	* * *	1
Low Shipping Weight	0	0	0	* * *

Cost Efficient	Score + 1
Minimal Parts	2.5
Parts Easily Obtainable	3.5
Easy to Manufacture	3
Low Shipping Weight	1
TOTAL	10

High Ranking	Weighted %	True Weighted %
Eliminate unwanted sounds	25%	2.73%
Good device contact	35%	3.82%
Easy to Manufacture	30%	3.27%
Low Shipping Weight	10%	1.091%
TOTAL	100%	10.91%

Table 9 Pair-wise comparison chart for Practical Sub-Objective

Practical (12.73%)	**Easy to Calibrate**	**Easy to Read Output**	**Comfortable to Wear**	**Lightweigh t**
Easy to Calibrate	* * *	0	0	0.5
Easy to Read Output	1	* * *	0.5	0.5
Comfortable to Wear	1	0.5	* * *	0.5
Lightweight	0.5	0.5	0.5	* * *
Practical	**Score + 1**			
Easy to Calibrate	1.5			
Easy to Read Output	3			
Comfortable to wear	3			
Lightweight	2.5			
TOTAL	10			
High Ranking	**Weighted %**	**True Weighted %**		
Easy to Calibrate	15%	1.901%		
Easy to Read Output	30%	3.82%		
Comfortable to Wear	30%	3.82%		
Lightweight	25%	3.18%		
TOTAL	100%	12.73%		

Table 10 Pair-wise comparison chart for Safe Sub-Objective

Safe (21.82%)	**Fully enclosed components**	**No sharp components**	**Nontoxic Components**
Fully enclosed components	* * *	0.5	0.5
No sharp components	0.5	* * *	0.5
Nontoxic components	0.5	0.5	* * *
Safe	**Score + 1**		
Fully enclosed components	2		
No sharp components	2		
Nontoxic components	2		
TOTAL	6		
High Ranking	**Weighted %**	**True weighted %**	
Fully enclosed components	33.3%	7.273%	
No sharp components	33.3%	7.273%	
Nontoxic components	33.3%	7.273%	
TOTAL	100%	21.82%	

Table 11 Pair-wise comparison chart for Accurate Sub-Objective

Accurate(20%)	**Appropriately Sensitive**	**Detects small and large movements**	**Not influenced by vibrations**	**High quality parts**
Appropriately sensitive	* * *	0.5	0.5	1
Detects small and large movements	0.5	* * *	0.5	1
Not influenced by vibrations	0.5	0.5	* * *	1
High quality parts	0	0	0	* * *
Accurate	**Score + 1**			
Appropriately sensitive	3			
Detects small and large movements	3			
Not influenced by vibrations	3			
High quality parts	1			
TOTAL	10			
High Ranking	**Weighted %**	**True Weighted %**		
Appropriately sensitive	30%	6%		
Detects small and large movements	30%	6%		
Not influenced by vibrations	30%	6%		
High quality parts	10%	2%		
TOTAL	100%	20%		

Table 12 Pair-wise comparison chart for Reliable Sub-Objective

Reliable(18.18%)	**Less Technical Support**	**Little maintenance required**	**Fewer Parts**	**High Quality Parts**
Less Technical Support	* * *	0.5	1	1
Little maintenance required	0.5	* * *	1	0.5
Fewer Parts	0	0	* * *	0.5
High Quality Parts	0	0.5	0.5	* * *
Reliable	**Score + 1**			
Less Technical Support	3.5			
Little maintenance required	3			
Fewer Parts	1.5			
High Quality Parts	2			
TOTAL	10			
High Ranking	**Weighted %**	**True Weighted %**		
Less Technical Support	35%	6.363%		
Little maintenance required	30%	5.454%		
Fewer Parts	15%	2.727%		
High Quality Parts	20%	3.636%		
TOTAL	100%	18.18%		

Table 13 Pair-wise comparison chart for Power Efficient Sub-Objective

Power Efficient(5.45%)	**Low power consuming parts**	**Maximize battery life**	**Fewer parts**
Low power consuming parts	* * *	0.5	1
Maximize battery life	0.5	* * *	1
Fewer Parts	0	0	* * *
Power Efficient	**Score + 1**		
Low power consuming parts	2.5		
Maximize battery life	2.5		
Fewer parts	1		
TOTAL	6		
High Ranking	**Weighted %**	**True Weighted %**	
Low power consuming parts	41.7%	2.273%	
Maximize battery life	41.7%	2.273%	
Fewer parts	16.6%	0.905%	
TOTAL	100%	5.45%	

Table 14 Pair-wise comparison chart for Durable Sub-Objective

Durable (10.91%)	**Wear-Resistant**	**Strong Materials**	**Water Resistant**
Wear-Resistant	* * *	0	0
Strong Materials	1	* * *	
Water Resistant	1	0.5	* * *
Durable	**Score + 1**		
Wear-Resistant	1		
Strong Materials	2.5		
Water Resistant	2.5		
TOTAL	6		
High Ranking	**Weighted %**	**True Weighted %**	
Wear-Resistant	16.6%	1.811%	
Strong Materials	41.6%	4.539%	
Water Resistant	41.6%	4.539%	
TOTAL	100%	10.90%	

*10.90 is a result of rounding

Table 15 Weighted benchmark chart for design alternatives

Weighted Benchmark Chart					
	Weight	**Polar Strap**	**Electrodes + Accel.**	**Sock Design**	**Hand Design**
Minimal parts	2.73%	90.00%	60.00%	70.00%	70.00%
Parts easily	3.82%	100.00%	70.00%	70.00%	70.00%
Easy to manufacture	3.27%	80.00%	50.00%	40.00%	65.00%
Low shipping weight	1.091%	80.00%	70.00%	90.00%	90.00%
Easy to calibrate	1.901%	80.00%	40.00%	30.00%	30.00%
Easy to read output	3.82%	90.00%	60.00%	90.00%	90.00%
Comfortable to wear	3.82%	90.00%	50.00%	60.00%	80.00%
Lightweight	3.18%	90.00%	50.00%	80.00%	75.00%
Fully enclosed	7.273%	100.00%	20.00%	90.00%	90.00%
No sharp	7.273%	100.00%	85.00%	95.00%	95.00%
Nontoxic	7.273%	100.00%	95.00%	95.00%	95.00%
Appropriately	6%	85.00%	60.00%	50%	60%
Detects small and large movements	6%	70.00%	60.00%	10%	10%
Not influenced by	6%	90.00%	40.00%	10%	10%
High quality parts	2%	80.00%	85.00%	60%	80%
Less technical	6.363%	70.00%	60.00%	30.00%	40.00%
Little maintenance	5.454%	70.00%	55.00%	40.00%	50.00%
Fewer parts	2.727%	90.00%	70.00%	80.00%	70.00%
Higher quality parts	3.636%	90.00%	90.00%	90.00%	80.00%
Lower power	2.273%	70.00%	40.00%	80.00%	70.00%
Maximize battery	2.273%	70.00%	30.00%	70.00%	60.00%
Fewer Parts	0.905%	90.00%	70.00%	70.00%	70.00%
Wear Resistance	1.811%	80.00%	40.00%	30.00%	50.00%
Strong Materials	4.539%	90.00%	30.00%	30.00%	40.00%
Water Resistance	4.539%	50.00%	25.00%	40.00%	50.00%
Totals	100%	85.00%	56.6%	58.8%	62.57%

B – IRB Application Process

Cover Letter

Professor Rismiller, Chairman of the Institutional Review Board:

This project is a continuation of a MQP project of the same name completed during the 2007-2008 academic year. The previous MQP was given IRB approval on March 13th, 2008, and considering this group's intention is to improve upon the previous project, the same study methodology will be used.

Sincerely,

Vinith Chemmalil

Marissa Gray

Jennifer Keating

Rebecca Kieselbach

Sarah Latta

Application

If your project has <u>any</u> federal sponsorship (e.g. federal funding), either prime or pass-through, the WPI IRB is <u>not authorized</u> to perform a review. Please contact Christina DeVries in Research Administration at (508) 831-6716 for direction to an appropriate IRB. <u>DO NOT</u> submit an application to the WPI IRB.

This application is for: *(Please check one)* ☒ Expedited Review ☐ Full Review

WPI IRB use only

Principal Investigator (PI) or Project Faculty Advisor: (*NOT a student or fellow; must be a WPI employee)*

Name:	Robert Peura	Tel No:	(508)769-4784	E-Mail Address:	rpeura@aol.com	☐
Department:	Biomedical Engineering					

Co-Investigator(s): *(Co-PI(s)/non students)*

Name:		Tel No:		E-Mail Address:		☐
Name:		Tel No:		E-Mail Address:		☐

Student Investigator(s):

Name:	Sarah Latta	Tel No:	(508) 713-5631	E-Mail Address:	slatta@wpi.edu	☐
Name:	Vinith Chemmalil	Tel No:	(617) 669-7606	E-Mail Address:	xxxvtcxxx@wpi.edu	☐

Check if: ☒ **Undergraduate project** *(MQP, IQP, Suff., other)* MQP - Diabetes Activity Monitor

☐ **Graduate project** *(M.S. Ph.D., other)*

Has an IRB ever suspended or terminated a study of any investigator listed above?

No ☒ Yes ☐ *(Attach a summary of the event and resolution.)*

Vulnerable Populations: The proposed research will involve the following (Check all that apply):

pregnant women ☐ human fetuses ☐ neonates ☐ minors/children ☐ prisoners ☐

students ☒ individuals with mental disabilities ☐ individuals with physical disabilities ☐

Collaborating Institutions: (*Please list all collaborating Institutions.)*

Locations of Research: (*If at WPI, please indicate where on campus. If off campus, please give details of locations.)*

WPI Fitness Center and Salisbury Laboratories 311

Project Title: Diabetes Activity Monitor

Funding: *(If the research is funded, please enclose one copy of the research proposal or most recent draft with your application.)*

Funding Agency: WPI Fund:

Human Subjects Research: *(**All** study personnel having direct contact with subjects **must** take and pass a training course on human subjects research. There is a link to a web-based training course that can be accessed under the Training link on the IRB web site http://www.wpi.edu/Admin/Research/IRB/training.html. The IRB requires a copy of the completion certificate from the course or proof of an equivalent program.)*

Anticipated Dates of Research:

Start Date: 12/15/2008 Completion Date: 5/10/2008

Instructions: Answer all questions. If you are asked to provide an explanation, please do so with adequate details. If needed, attach itemized replies. Any incomplete application will be returned.

1.) Purpose of Study: *(Please provide a concise statement of the background, nature and reasons for the proposed study. Insert below using non-technical language that can be understood by non-scientist members of the IRB.)*

The device built during this project needs to be tested to ensure its level of accuracy. The device will use a strap placed around the chest to measure heart rate and movement. This information will be sent to a wireless device with a display of the information.

2.) Study Protocol: *(Please attach sufficient information for effective review by non-scientist members of the IRB. Define all abbreviations and use simple words. Unless justification is provided this part of the application must not exceed 5 pages. Attaching sections of a grant application is not an acceptable substitute.)*

A.) For **biomedical, engineering and related research**, please provide an outline of the actual experiments to be performed. Where applicable, provide a detailed description of the experimental devices or procedures to be used, detailed information on the exact dosages of drugs or chemicals to be used, total quantity of blood samples to be used, and descriptions of special diets.

B.) For applications in the **social sciences, management and other non-biomedical disciplines** please provide a detailed description of your proposed study. Where applicable, include copies of any questionnaires or standardized tests you plan to incorporate into your study. If your study involves interviews please submit an outline indicating the types of questions you will include.

C.) If the study involves **investigational drugs or investigational medical devices**, and the PI is obtaining an Investigational New Drug (IND) number or Investigational Device Exemption (IDE) number from the FDA, please provide details.

D.) Please note if any **hazardous materials** are being used in this study.

E.) Please note if any **special diets** are being used in this study.

3.) Subject Information:

A.) Please provide the exact number of subjects you plan to enroll in this study and describe your subject population. *(eg. WPI students, WPI staff, UMASS Medical patient, other)*

Males: 20 Females: 20 Description: WPI Students

B.) Will subjects who do not understand English be enrolled?

No ☒ Yes ☐ *(Please insert below the language(s) that will be translated on the consent form.)*

C.) Are there any circumstances under which your study population may feel coerced into participating in this study?

No ☒ Yes ☐ *(Please insert below a description of how you will assure your subjects do not feel coerced.)*

D.) Are the subjects at risk of harm if their participation in the study becomes known?

No ☒ Yes ☐ *(Please insert below a description of possible effects on your subjects.)*

E.) How will subjects be recruited for participation? *(Check all that apply.)*

☒ Referral: *(By whom)* Project Memebers

☐ Other: *(Identify)* ______________________

☐ Database: *(Describe how database populated)* ______________________

☐ Direct subject advertising, including: *(Please provide a copy of the proposed ad. All direct subject advertising must be approved by the WPI IRB prior to use.)*

☐ Newspaper	☐ Bulletin board
☐ Radio	☐ Flyers
☐ Television	☐ Letters
☐ Internet	☐ E-mail

F.) Have the subjects in the database agreed to be contacted for research projects? No ☐ Yes ☐ N/A ☒

G.) Are the subjects being paid for participating? *(Consider all types of reimbursement, ex. stipend, parking, travel.)*

No ☒ Yes ☐ *(Check all that apply.)* ☐ Cash ☐ Check ☐ Gift certificate ☐ Other: ______________

Amount of compensation ______________________

4.) Informed Consent:

A.) Who will discuss the study with and obtain consent of prospective subjects? *(Check all that apply.)*

☐ Principal Investigator ☐ Co-Investigator(s) ☒ Student Investigator(s)

B.) Are you aware that subjects must read and sign and Informed Consent Form prior to conducting any study-related procedures and agree that all subjects will be consented prior to initiating study related procedures? No ☐ Yes ☒

C.) Are you aware that you must consent subjects using only the IRB-approved Informed Consent Form? No ☐ Yes ☒

D.) Will subjects be consented in a private room, not in a public space? No ☐ Yes ☒

E.) Do you agree to spend as much time as needed to thoroughly explain and respond to any subject's questions about the study, and allow them as much time as needed to consider their decision prior to enrolling them as subjects? No ☐ Yes ☒

F.) Do you agree that the person obtaining consent will explain the risks of the study, the subject's right to decide not to participate, and the subject's right to withdraw from the study at any time? No ☐ [✓ Yes ☒]

G.) Do you agree to either 1.) retain signed copies of all informed consent agreements in a secure location for at least three years or 2.) supply copies of all signed informed consent agreements in .pdf format for retention by the IRB in electronic form? No ☐ [✓ Yes ☒]

(If you answer No to any of the questions above, please provide an explanation.)

5.) Potential Risks: *(A risk is a potential harm that a reasonable person would consider important in deciding whether to participate in research. Risks can be categorized as physical, psychological, sociological, economic and legal, and include pain, stress, invasion of privacy, embarrassment or exposure of sensitive or confidential data. All potential risks and discomforts must be minimized to the greatest extent possible by using e.g. appropriate monitoring, safety devices and withdrawal of a subject if there is evidence of a specific adverse event.)*

A.) What are the risks / discomforts associated with each intervention or procedure in the study?

The band may be uncomfortable for some subjects and must be worn beneath clothing.

B.) What procedures will be in place to prevent / minimize potential risks or discomfort?

The band will be placed by the subject. If any adjustments are needed a member of the same gender will assist the subject.

6.) Potential Benefits:

A.) What potential benefits other than payment may subjects receive from participating in the study?

N/A

B.) What potential benefits can society expect from the study?

This device will aid diabetic patients in determining correct insulin dosages.

7.) Data Collection, Storage, and Confidentiality:

A.) How will data be collected?

Through datasheets filled out by the student investigators.

B.) Will a subject's voice, face or identifiable body features *(eg. tattoo, scar)* be recorded by audio or videotaping?
No ☒ Yes ☐ *(Explain the recording procedures you plan to follow.)*

C.) Will personal identifying information be recorded? No ☒ Yes ☐ *(If yes, explain how the identifying information will be protected. How will personal identifying information be coded and how will the code key be kept confidential?)*

D.) Where will the data be stored and how will it be secured?

The data will be stored on only one computer and all hard copies of data will be collected at the end of the study and destroyed.

E.) What will happen to the data when the study is completed?

All data, excepting what generalizations are reported, will be destroyed.

F.) Can data acquired in the study adversely affect a subject's relationship with other individuals? *(i.e. employee-supervisor, student-teacher, family relationships)*

No.

G.) Do you plan to use or disclose identifiable information outside of the investigation personnel?

No ☒ Yes ☐ *(Please explain.)*

H.) Do you plan to use or disclose identifiable information outside of WPI including non-WPI investigators?

[✓ No ☒] Yes ☐ *(Please explain.)*

8.) Deception: *(Investigators must not exclude information from a subject that a reasonable person would want to know in deciding whether to participate in a study.)*

Will the information about the research purpose and design be withheld from the subjects?

[✓ No ☒] Yes ☐ *(Please explain.)*

9.) Adverse effects: *(Serious or unexpected adverse reactions or injuries must be reported to the WPI IRB within 48 hours. Other adverse events should be reported within 10 working days.)*

What follow-up efforts will be made to detect any harm to subjects and how will the WPI IRB be kept informed?

Upon their completetion in the study, subjects will have a one on one follow-up meeting to determine if there were any adverse effects. If there were, the WPI IRB will be immediately informed.

10.) Informed consent: (*Documented informed consent must be obtained from all participants in studies that involve human subjects. You must use the templates available on the WPI IRB web-site to prepare these forms.* ***Informed consent forms must be included with this application.*** *Under certain circumstances the WPI IRB may waive the requirement for informed consent.)*

Investigator's Assurance:

I certify the information provided in this application is complete and correct.

I understand that I have ultimate responsibility for the conduct of the study, the ethical performance of the project, the protection of the rights and welfare of human subjects, and strict adherence to any stipulations imposed by the WPI IRB.

I agree to comply with all WPI policies, as well all federal, state and local laws on the protection of human subjects in research, including:

- ensuring the satisfactory completion of human subjects training.
- performing the study in accordance with the WPI IRB approved protocol.
- implementing study changes only after WPI IRB approval.
- obtaining informed consent from subjects using only the WPI IRB approved consent form.
- promptly reporting significant adverse effects to the WPI IRB.

Signature of Principal Investigator ______________________________ Date ____________

Print Full Name and Title ____

Please return a signed hard copy of this application to the WPI IRB c/o Research Administration.

If you have any questions, please call (508) 831-6716.

Protocol

Protocol Number:	DAM-020
Protocol Title:	Preliminary Calibration and Testing of the Combined Heart Rate and Accelerometer for Determining Energy Expenditure
Study Sponsor:	Worcester Polytechnic Institute, Worcester, MA
Investigational Product:	The Combined Heart Rate and Accelerometer Activity Monitoring Device
Study Objectives:	Primary Study Objectives Include: • Calibrate the combined heart rate and accelerometer monitoring device to accurately predict energy expenditure; • Screening for potential safety issues.
Study Design:	Overview: • Prospective, simultaneously-recruited, un-blinded, preliminary study • Number of subjects = up to 40 • The primary study endpoint -device calibration- will be achieved through having subjects perform a variety of different activities while measuring their heart rate and accelerometer outputs in order to adjust the activity algorithm. Healthy volunteers will be enrolled accordingly to the inclusion/exclusion criteria. Prior to the activity period, the subjects will lie down in a comfortable area for five minutes while wearing the investigational device and a pulse oximeter. The resting heart rate from both devices will be obtained at the end of five minutes. During the first activity period, the subjects will wear the investigational device while performing different activities for five minutes with at least five minutes of rest in between activities. The activities that the subjects will be performing and the order that they will perform them in are listed as follows: sitting, slow walk on the treadmill (3.10 mph), fast walk on the treadmill (3.80 mph), and running on the treadmill (5.90 mph). The activity counts and heart rate will be obtained from the investigational device during the final minute of each activity. Within one minute after the completion of the first activity period, the subject’s heart rate will be recorded using both the investigational device and a pulse oximeter. During the second activity period, the subjects will be wearing the investigational device while performing the following activities in the given order with five minutes rest in between: ascending/descending stairs, sweeping, washing dishes, filing papers, and standing. The activity counts and heart rate will be obtained from the investigational device during the final minute of each activity. Within one minute after the completion of the first activity period, the subject’s heart rate will be recorded using both the investigational device and a pulse oximeter. The student investigators will compare the accelerometer counts per minute against the counts per minute of the Actiheart combined heart rate and accelerometer activity monitor published in the European Journal of Clinical Nutrition. Using linear regression techniques, the student investigators will determine the correlation coefficient between the two accelerometer’s counts per minute. This number will be used to adjust the existing Actiheart activity algorithm to our accelerometer specifications.
Study Methods	*Primary Study Endpoints:*

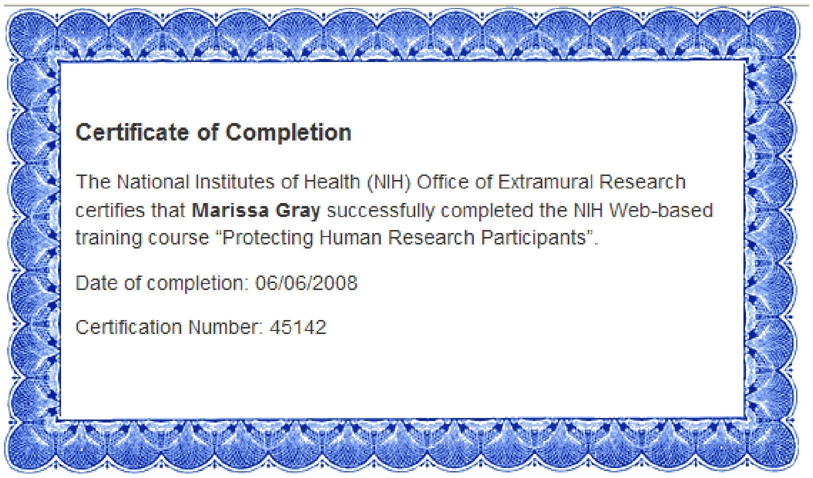
	• Activity monitor calibration, preformed by adapting published algorithms for an existing combined heart rate and movement sensor. *Key Secondary Study Endpoints Include:* • Heart Rate stand alone accuracy, measured by the difference between the heart rate measurements obtained by the investigational device and the measurements obtained by a pulse-oximeter. *Key Safety Assessments Include:* • Potential effects of electrode placement, as measured by observed and reported discomfort and changes in skin during and after the testing period.
Study Conduct	In Accordance With: • Principles of Good Clinical Practice; • Principles of Declaration of Helsinki (1989); • Title 21 Parts 50, 54, 56, and 812 of the U.S. Code of Federal Regulations; • The Medical Research Council's Code of Ethical Conduct for Research Involving Humans (1997); and • All federal, provincial, state, and local laws of the pertinent regulatory authorities.
Subject Selection	Major Study Inclusion Criteria Include: • Men and women between 18 and 25 years of age; and • Healthy volunteers Major Study Exclusion Criteria Include: • Subjects with a history of respiratory problems (e.g., asthma) • Subjects with a history of heart problems (i.e. irregular heartbeat) • Subjects who are recovering from injury or surgery; and • Subjects who for any reason are unable to perform physical activities.
Study Plan	Study Periods Include: • Baseline Period (up to 30 days); • Calibration and Testing Period (up to 4 hours); • Follow Up Period (approximately 24 hours); and • Analysis Period (up to 30 days).
Statistical Methods	Statistical Methods Include: • Primary study endpoint: Tests for differences in accuracy will be conducted using a paired t-test. Study Sample Size: • Total Male Enrollment Population: up to 20 subjects • Total Female Enrollment Population: up to 20 subjects • Total Study Subject Enrollment Population: up to 40 subjects.

NIH Certificates

The National Institutes of Health (NIH) Office of Extramural Research certifies that Rebecca Kieselbach successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 10/27/2008

Certification Number: 125235

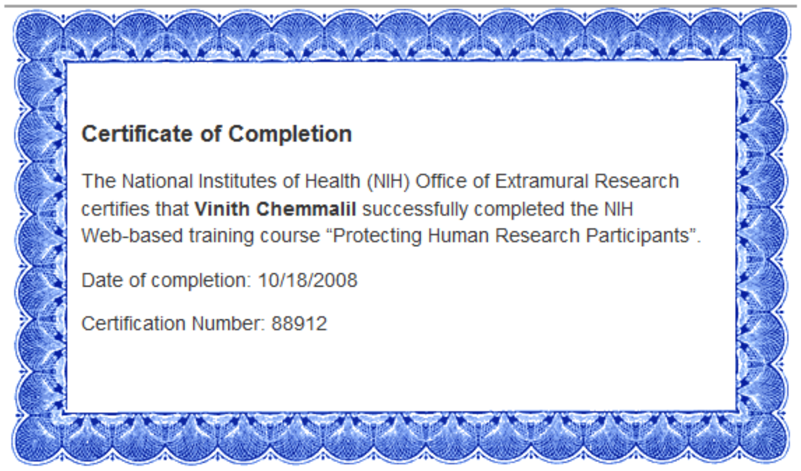


Certificate of Completion

The National Institutes of Health (NIH) Office of Extramural Research certifies that **Marissa Gray** successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 06/06/2008

Certification Number: 45142

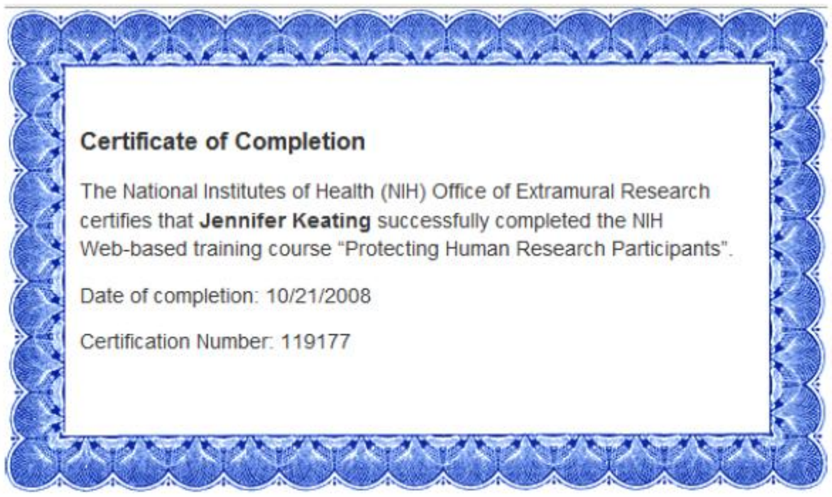


Certificate of Completion

The National Institutes of Health (NIH) Office of Extramural Research certifies that **Vinith Chemmalil** successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 10/18/2008

Certification Number: 88912

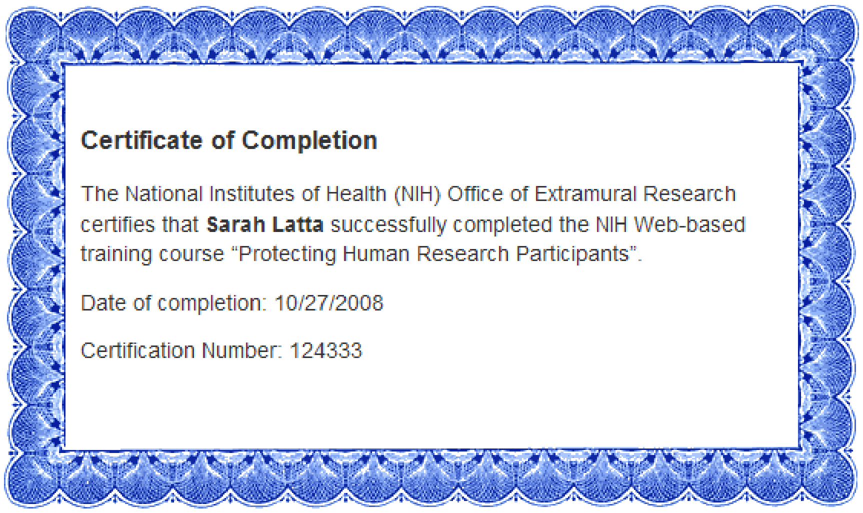


Certificate of Completion

The National Institutes of Health (NIH) Office of Extramural Research certifies that **Jennifer Keating** successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 10/21/2008

Certification Number: 119177



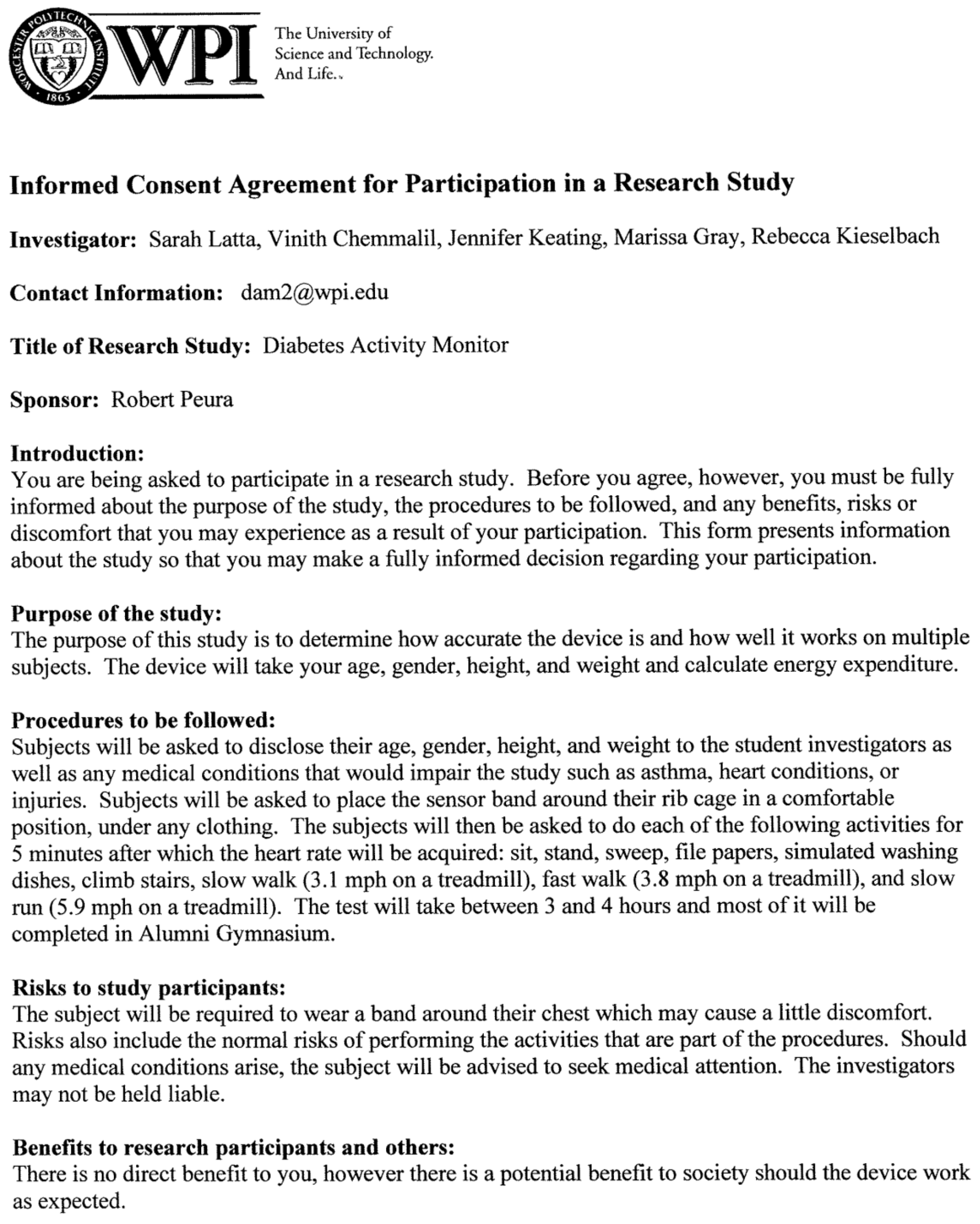
Certificate of Completion

The National Institutes of Health (NIH) Office of Extramural Research certifies that **Sarah Latta** successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 10/27/2008

Certification Number: 124333

IRB Approved Consent Form



The University of
Science and Technology.
And Life.

Informed Consent Agreement for Participation in a Research Study

Investigator: Sarah Latta, Vinith Chemmalil, Jennifer Keating, Marissa Gray, Rebecca Kieselbach

Contact Information: dam2@wpi.edu

Title of Research Study: Diabetes Activity Monitor

Sponsor: Robert Peura

Introduction:
You are being asked to participate in a research study. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

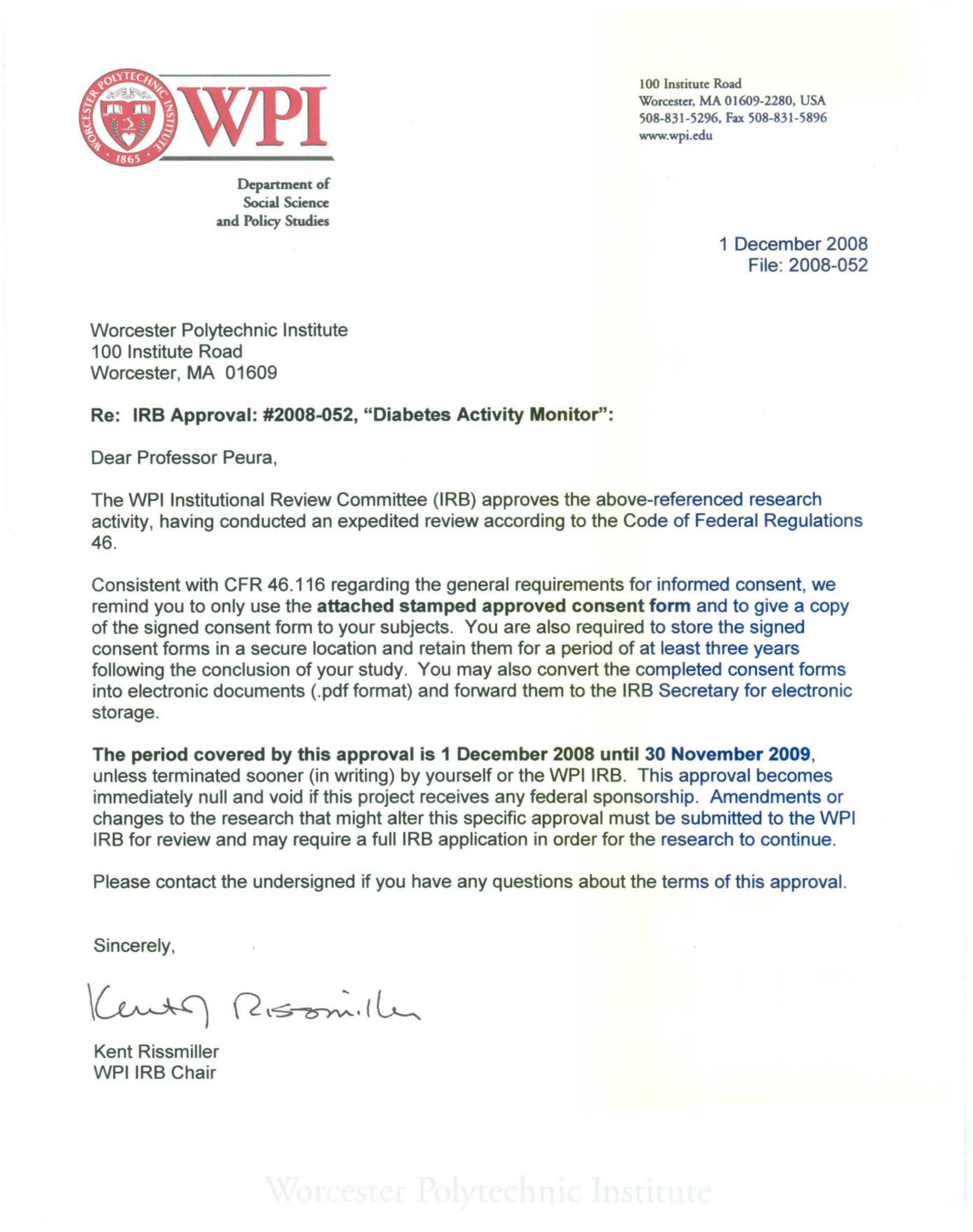
Purpose of the study:
The purpose of this study is to determine how accurate the device is and how well it works on multiple subjects. The device will take your age, gender, height, and weight and calculate energy expenditure.

Procedures to be followed:
Subjects will be asked to disclose their age, gender, height, and weight to the student investigators as well as any medical conditions that would impair the study such as asthma, heart conditions, or injuries. Subjects will be asked to place the sensor band around their rib cage in a comfortable position, under any clothing. The subjects will then be asked to do each of the following activities for 5 minutes after which the heart rate will be acquired: sit, stand, sweep, file papers, simulated washing dishes, climb stairs, slow walk (3.1 mph on a treadmill), fast walk (3.8 mph on a treadmill), and slow run (5.9 mph on a treadmill). The test will take between 3 and 4 hours and most of it will be completed in Alumni Gymnasium.

Risks to study participants:
The subject will be required to wear a band around their chest which may cause a little discomfort. Risks also include the normal risks of performing the activities that are part of the procedures. Should any medical conditions arise, the subject will be advised to seek medical attention. The investigators may not be held liable.

Benefits to research participants and others:
There is no direct benefit to you, however there is a potential benefit to society should the device work as expected.

IRB Approval Letter



100 Institute Road
Worcester, MA 01609-2280, USA
508-831-5296, Fax 508-831-5896
www.wpi.edu

Department of
Social Science
and Policy Studies

1 December 2008
File: 2008-052

Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609

Re: IRB Approval: #2008-052, "Diabetes Activity Monitor":

Dear Professor Peura,

The WPI Institutional Review Committee (IRB) approves the above-referenced research activity, having conducted an expedited review according to the Code of Federal Regulations 46.

Consistent with CFR 46.116 regarding the general requirements for informed consent, we remind you to only use the **attached stamped approved consent form** and to give a copy of the signed consent form to your subjects. You are also required to store the signed consent forms in a secure location and retain them for a period of at least three years following the conclusion of your study. You may also convert the completed consent forms into electronic documents (.pdf format) and forward them to the IRB Secretary for electronic storage.

The period covered by this approval is 1 December 2008 until 30 November 2009, unless terminated sooner (in writing) by yourself or the WPI IRB. This approval becomes immediately null and void if this project receives any federal sponsorship. Amendments or changes to the research that might alter this specific approval must be submitted to the WPI IRB for review and may require a full IRB application in order for the research to continue.

Please contact the undersigned if you have any questions about the terms of this approval.

Sincerely,

Kent Rissmiller

Kent Rissmiller
WPI IRB Chair

Worcester Polytechnic Institute

Appendix C – Programming

Signal Acquisition Module Code

Main.c

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  main.c

  Program to demonstrate writing/reading of data to/from the QF4A512 using a TI
MSP430F449.

  $Id: main.c 121 2006-07-24 20:17:09Z jhopson $


*************************************************************************************
********

  @formats
      - C
      - Doxygen (comment markup to produce HTML docs)

  @dependencies
      - Operating System:  none
      - Toolset:           IAR MSP
      - Platform:          TI MSP-TS430PZ100  (any MSP430 should work)
      - CPU Architecture:  MSP-430
      - CPU Variant:       '149   (any MSP430 should work)
      - Device:            USART0 and/or USART1
      - CPU byte order:    little endian

*//**********************************************************************************
*********
        Below will appear on the main page (index.html) of the Doxygen HTML output.

*************************************************************************************
******//*!

  @mainpage   notitle

  @htmlonly
  <br><p><div><table width="100%" border="0"><tr><td width="150">
  @endhtmlonly
  @image html QuickfilterLogo.png
  @htmlonly </td>
  <td padding-top="100px" padding-left="100px">  <h2>APPLICATION
NOTE&nbsp;&nbsp;&nbsp; QFAN003</h1>
       <h2>MSP-430 Host Software Example for the QF4A512</h2></td></tr></table><p>
       <hr>
  @endhtmlonly

  @section  Introduction

  The QF4A512 is controlled with a set of registers visible through its SPI
interface.  This
```

```
  Application Note provides a working example of the QF4A512 when connected to a TI
MSP-430
  microcontroller.

  The goals of this Project are –

    - Demonstrate QF4A512 operation in detail on a popular microcontroller (TI MSP-
430)
    - Provide a means for the developer to get a target running quickly with high-
level C code
    - Provide a clean user interface, which abstracts much of the QF4A512 interface
detail

  A .zip archive file, named <b>QFAN003-Project Content.zip</b>, contains the
technical content
  for this Application Note.

  @section  demo  Demonstration Circuit
  The platform for this demonstration is the TI MSP-TS430PZ100 Target Socket Module
connected to
  a Quickfilter QF4A512-DK Development Board, via the USART0 SPI interface.


  @verbatim

         3.3           MSP430F449              3.3       J5              QF4A512
3.3
          ^        --------------------         ^                    --------------
-         ^
          | 1,60  |                    |  94    |        11     17  |
| 20,32  |
          |  100  |               /RST |<-------o------------------>| /RST
|--------+
          +-------|                    |  70              3     15  |
|
                  |         SIMO0/P3.1 |--------------------------->| SDI
|
               57 |                    |  69              5     14  |
|       1.8
             o----|         SOMIO/P3.2 |<---------------------------| SDO
|  9,19, ^
    Test       58 |                    |  68              1     16  |
| 26,28, |
    Points   o----|         UCLK0/P3.3 |--------------------------->| SCLK
| 29     |
               59 |                    |  87              9     13  |
|--------+
             o----|               P1.0 |<---------------------------| /DRDY
|
                  |                    |  71              7     12  |
|
               82 |               P3.0 |--------------------------->| /CS
| 10,11,18
             o----| ACLK/P1.5          |                            |
| 24,25,27
    Clock      83 |                    |  12                        |
| 30,31
    Test     o----| SMCLK/P1.4    P5.1 |-----+                      |
|--------+
               86 |                    |     | Activity             |  XIN   XOUT
|        |
             o----| MCLK/P1.1          |    _|_  _              ---------------
-        ---
                  |                    |    \ /  /|                | 22   | 23
GND
                  |                    |    _v_                    |      |
         61,98,99 |                    |     |                     +-|[]|-+
          +-------|    XIN   XOUT      |     \
          |        --------------------      /  560                  20 MHz
         ---            |  8   |  9          |  Ohms
         GND            |      |            ---
                       +-|[]|-+             GND

                       32.768 KHz
        ^                                ^                                       ^
        |                                                |
|
        +-------------   MSP-TS430PZ100   --------------+-----------   QF4A512-DK
-------------+
  @endverbatim

  The QF4A512 SPI pins are available on the Quickfilter QF100-DB23 Development
Board via J5, the
  External SPI header.

  The pinout for J5 is shown in the diagram above.  SPI Header pins 2, 4, 6, 8, 10,
12 & 14 are ground.
  Leave pin 13 pulled high through R56.

  See Qf4a512-access.h for more information on the internals of the QF4A512.


  @section  design  Software Design

  The QF4A512 may be attached to devices of widely varying scale and integration.
To support
  both ends of the spectrum, there are two target models for the QF4A512, Small and
Large.  This
  example code represents the @b Small model, which targets small microcontroller-
based
  systems.  Some features and assumptions of the Small Model library are -

    - Assumes no operating system
      -# No file system
      -# No kernel logging
      -# Unstructured interrupt handling
    - Minimal memory footprint
      -# No heap (statically allocated buffers)
      -# Limited buffer size
    - Minimal runtime overhead in Release build
    - Minimal power consumption
      -# CPU idle while not processing data
    - Written in C (not C++)
      -# Unstructured exception handling
      -# Statically linked
      -# No namespaces
    - 8/16-bit SPI hardware
    - All channels sample at the same rate
    - Any number of channels supported
      -# Channel count is a compile-time option
      -# Channels must start at 0 and expand up

  Features and assumptions that are common to all models are -

    - Supports multiple logical devices
    - Interrupt and/or DMA driven
    - Structured C
    - Does not conform to a specific driver framework
    - QF4A512 driver decoupled from hardware by thin, simple hardware access API
    - Defensive coding measures in Debug build
        -# Extensive use of assertions
    - Source code internally documented to produce an HTML User Guide.

  @section  structure Software Structure

  This code is intended to be a structured example that demonstrates the
configuration and
  operation of the QF4A512.  Among its features are Configure mode reads and
writes, processing
  of Run mode data, and writing and reading the on-chip EEPROM.  Its structure is
shown below.


  @verbatim
                                                                  Implemented in
                                                                  --------------
    +-------------+------------------------------------------+
    |             |              Application Code            |   main.c
    |             |                                          |
    |  Standard   |            +-----------------------------+
    |      C      |            |  QF4A512 Functional Driver  |   Qf4A512-
functional.c
    |  Libraries  +------------+-----------------------------+
    |             |            QF4A512 Access Driver         |   QF4A512-access.c
    |             +------------------------------------------+
    |             |            Hardware Abstraction          |   Msp430-
SPI.c/Platform.c
    +-------------+------------------------------------------+
  @endverbatim


  QF4A512-access.c and QF4A512-functional.c are the focus of this project.
QF4A512-access.c
  provides raw data transfer functions and mode control.  QF4A512-functional.c
provides a
  library of specific control features built on top of the API in QF4A512-access.c.
In Run
  mode, qf4a512_ReadSamples(), in QF4A512-access.c, is the go-to function for
reading the data
  stream from the device.

  QF4A512-access.c and QF4A512-functional.c are written without any SPI hardware
dependencies
  and are portable to any processor that uses little-endian byte order and provides
an
  underlying SPI API appropriate to the target device.  Msp430-SPI.c provides the
SPI API in
```

```
  this case.  Msp430-SPI.c and Platform.c are non-portable, and have to be modified
for each
  CPU variant and hardware platform, respectively.

  Note that this code does not demonstrate device calibration.  See
  “QFAN012 - Calibration of the QF4A512”, at quickfiltertech.com for calibration
information.


  @section  howto  How To Build

  @subsection  tools  Tools Used
  This project was compiled using the Evaluation Version of the IAR Embedded
Workbench, available
  at www.iar.de, and the Texas Instruments MSP-FET430UIF USB-to-JTAG debug
interface, available
  at www.ti.com ($99).

  The Windows help file (HTML) documentation is automated using Doxygen, an open-
source tool
  available at www.doxygen.org.  Doxygen can (optionally) use a open-source tool
called Dot to
  create call and inheritance graphs in the documentation. Dot is available at
www.graphviz.org.

  @subsection  procedure  Build Procedure

  To build this example, load the tools and unzip “QFAN003-Project Content.zip”.
Then double-
  click “IAR Workspace.eww” to open the workspace.  To build, select  Project-
>Rebuild All from
  the Main menu.  This demonstration project should build with no warnings or
errors.  Note that
  the stack size is set to 100 (64h) bytes.

  @image html IAR.png

  The inheritance graphs in the documentation are optional in Doxygen, but if Dot
is available
  and the HAVE_DOT option is set to 'yes' in Doxyfile, they will be produced.

  To build the HTML documentation, open Doxywizard from the Windows Start bar.
Then open
  "Document\Doxyfile" from the File->Open... dialog and select the Start button on
Doxywizard's
  main screen.  Doxygen should produce no warnings or errors for this demonstration
project.

  @image html Doxywizard.png

  This source code has also been built with two other tools after minor
modification.  The other
  tools are TI's Code Composer Essentials (www.ti.com) and Mspgcc
(mspgcc.sourceforge.net).  The
  latter is free, works on virtually any operating system, and appears to be of the
same quality
  as the commercial tools.
  @section  flow  Design Flow with PC Software

  This section describes how to integrate the output of the Quickfilter PRO PC
software with
  this example code.  It's important to point out that this flow is for designs
that have
  changing filter requirements during runtime.  Designs that need one configuration
should
  consider programming the QF4A512 internal EEPROM once during production.

  The first step is to get the PC software to produce a C header (.h) file that
contains a
  table of configuration entries for the QF4A512.  Be sure to use Version 3.1.1, or
later, of
  the PC software.  The Export tab on the QFControl dialog, shown below, produces
that file.

  @image html QfPro-Export.png

  The table of entries contains both the control register entries and the FIR
filter
  coefficients.  It is named "QFImageRegisterTable[]".  To load the table at
runtime, pass
  the array to the qf4a512_LoadImageRegisterTable() function.


  @subsection  build  Other Notes

  The assert() macro, and macros built upon it (AssertNonNull, etc.), are used to
instrument the
  code in the Debug build configuration.  These statements are removed in Release
builds, and
  have no effect on code size or bandwidth in that case.  However, they do consume
code space
  and runtime bandwidth in a Debug build.  The tradeoff is an improvement in the
ability to
  catch many types of bugs earlier in the debug process.  See assert.h in the C
library
  documentation.

  @par
  Please send any comments, bug reports, etc. relating to this example to
apps@quickfilter.com

*///*****************************************************************************
**************


//********************************FUNCTION
DECLARATIONS*****************************************//

void runtimerb(void);
void stoptimerb(void);
void send(void);
int currentTime(unsigned int);
void swDelay(unsigned int max_cnt);
void ZbSpi_Init(void);
```

```
//*****************************************************************************************
************//

__interrupt void  TimerB0Isr( void );

unsigned short int timer = 0;
unsigned int numSamples = 0;
unsigned int rectSample = 0;
unsigned int sumSample = 0;
unsigned int multSamples = 0;
unsigned int txSample = 0;
unsigned char txSample1 = 0;
unsigned char txSample2 = 0;


#include  "Msp430-SPI.h"
#include  "Qf4a512-access.h"
#include  "Qf4a512-functional.h"
#include  "Project.h"
#include  "Platform.h"
#include <msp430x44x.h>


//  Include table with device settings produced by the Quickfilter Pro software
//#include  "QFImageRegisterTable.h"


#define  NUM_FRAMES_PER_SAMPLE_REQUEST   1

int  main( void )
{
WDTCTL = WDTPW + WDTHOLD;
    //  Initialize the platform
    QfPlat_Init();

    //  Init QF4A512 and SPI hardware to which the QF4A512 is attached.
    qf4a512_Init();

    //Initialize the UART hardware to which the ZBEE is attached
    ZbSpi_Init();

    //  Load the table of settings and FIR coefficients from the Quickfilter Pro PC
    //  software.  (see header above for instructions on how to generate the table)

//    qf4a512_LoadImageRegisterTable(
//        SPI0_HANDLE,
//        (qf4a512_ConfigTableEntry *)QFImageRegisterTable,
//        QF_IMAGE_REGISTER_TABLE_DIMENSION);

   WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer
   _BIS_SR(GIE);   // Global Interrupt enable


    InfiniteLoop()
    {
      Byte  Frame;

        ///  Buffer to hold the samples currently being processed
        UInt16  Sample[ NUM_FRAMES_PER_SAMPLE_REQUEST ][
QF4A512_NUM_CHANNELS_ENABLED ];

        //  Get a sample of data
        Validate(
            qf4a512_ReadSamples(
                SPI0_HANDLE,                      //  Device handle
                Sample,                           //  Location to place the
samples
                NUM_FRAMES_PER_SAMPLE_REQUEST )    //  Number of frames to read
                 );

        // Start the timer
        runtimerb();

        //  Rectify sample and remove offset for each channel and then sum the
samples
        for( Frame = 0;
             Frame < NUM_FRAMES_PER_SAMPLE_REQUEST;
             Frame++ )
             {
                 Byte  Channel;

                 for( Channel = 0;
                      Channel < QF4A512_NUM_CHANNELS_ENABLED;
                      Channel++ )
                     {
                        rectSample  =  abs( Sample[ Frame ][ Channel ] - 32768);
                     sumSample   =  rectSample + sumSample;
                     }

                //Sum rectified&summed samples
                multSamples = multSamples + sumSample;
                numSamples++;

                  if(currentTime(10))                       //if 10 sec have passed
                   {
                     stoptimerb();                          //Stop timer B and reset
timer to 0
                     txSample = multSamples/numSamples;   //Average samples from 10
second period
                 txSample1 = txSample & (BIT7|BIT6|BIT5|BIT4|BIT3|BIT2|BIT1|BIT0);
                 txSample2 = txSample >> 8;

                     // Transmit Data
                     U1TXBUF = txSample1;
                     send();
                     swDelay(5);            // sw delay function
                     U1TXBUF = txSample2;
                     send();
                     swDelay(5);            // sw delay function

                     runtimerb();           //Restart timer B
                     numSamples = 0;
                     rectSample = 0;
                     sumSample = 0;
                     multSamples = 0;
                     txSample1 = 0;
                     txSample2= 0;
```

```
                    }
              }
     }


      //  If your code ever stops reading Run mode data at runtime,
      //  be sure to call  qf4a512_ExitRunMode( SPI0_HANDLE );
}


//*********************************************************************************
***********//

//  @summary  start TIMER B  Delay 10 seconds

//*********************************************************************************
**********//

void runtimerb(void)
{
   TBCTL= TBSSEL_1 + CNTL_0 + MC_1 + ID_1;   //ACLK, 16 Bit, up mode, div=2
   TBR = 0x5000;                             //32768 clk tics = 10 sec
   TBCCTL0 = CCIE;                            //TBR interrupt enabled
}


//*********************************************************************************
***********//

//  @summary  stop TIMER B

//*********************************************************************************
***********//

void stoptimerb(void)
{
   TBCTL= MC_0;       //stop timer
   TBCCTL0 &= ~CCIE;   //TBR interrupt disabled
   timer = 0;         //reset timer to 0
}


//*********************************************************************************
***********//

//  @summary  determines if the number of seconds specified by delay has been
reached

//*********************************************************************************
***********//

int currentTime(unsigned int delay)
{
   unsigned short int sec;           //elapsed seconds
   _DINT();                          //disable interrupts
   sec = (timer/32768)*10;           //sec gets the value of 10*timer/32768
   _EINT();                          //enable interrupts
   if (sec >= delay)                  //If the number of seconds is greater than the
timer delay...
      return 1;                       //return 1
   else
      return 0;                       //else return 0
}


//*********************************************************************************
***********//

//  @summary  Timer B0 interrupt service routine.

//*********************************************************************************
***********//

#pragma  vector = TIMERB0_VECTOR
__interrupt void TimerB0Isr(void)
{
   timer++;    //increment the timer
}


//*********************************************************************************
***********//

//  @summary  Waits till TX buffer is done

//*********************************************************************************
***********//

void send(void)
{
   char dmy = 0;

   while (!(IFG2 & UTXIFG1))                 // USART1 TX buffer done
      dmy=dmy;
}

//*********************************************************************************
***********//

//  @summary  Software delay

//*********************************************************************************
***********//

void swDelay(unsigned int max_cnt)
{
   unsigned int cnt1=0, cnt2;

   while (cnt1 < max_cnt)
   {
     cnt2 = 0;
     while (cnt2 < 65535)
       cnt2++;
     cnt1++;
   }
}
```

```
/*****************************************************************************************
********

*****************************************************************************************
*******
 ********
********
 ********                        S P I   F O R   Z B E E
********
 ********
********

*****************************************************************************************
*******

*****************************************************************************************
*******/

void ZbSpi_Init(void)
{
    //Configure UART1 for XBEE at 9600 baud Transmit -- Interrupts not enabled
    P3SEL |= 0x30;                            // P3.6,7 = USART1 TXD/RXD
    ME2 |= UTXE1 + URXE1;                     // Enable USART1 TXD/RXD
    U1CTL |= CHAR;                            // 8-bit character
    U1TCTL |= SSEL0;                          // BRCLK = SMCLK
    U1BR0 = 0x6D;                             // 1MHz 9600
    U1BR1 = 0x00;                             // 1MHz 9600
    U1MCTL = 0x03;                            // modulation
    U1CTL &= ~SWRST;                          // Initialize USART state machine
    P3DIR |= BIT6;                            // P3.6 output direction
    P3DIR &= ~BIT7;                           // P3.7 input direction
}
```

MSP430 – SPI.c

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Msp430-SPI.c

  Abstraction of MSP-430 SPI port for use with Quickfilter device drivers.

  $Id: Msp430-SPI.c 125 2006-07-24 20:40:36Z jhopson $


*****************************************************************************************
********

  @formats
      - C
      - Doxygen (comment markup to produce HTML docs)

  @dependencies
      - Operating System:  none
      - Toolset:           IAR MSP
      - Platform:          TI MSP-TS430PZ100  (any MSP430 should work)
      - CPU Architecture:  MSP-430
      - CPU Variant:       '449   (any MSP430 with the same USART should work)
      - Device:            USART0
      - CPU byte order:    little endian

  @notes
    -# This non-portable C file abstracts the SPI port of an MSP-430 for use by any 
SPI-
       based Quickfilter devices.  This code doesn't have any context about the 
meaning
       of the data it is sending or receiving over the SPI port.  It passes a raw 
payload
       of bytes to or from the SPI port.  Interrupts are used, DMA is not.

       @par
       @verbatim
         +-------------+-------------------------------------------+
         |             |             Application Code              |
         |  Standard   +-------------------------------------------+         This 
file
         |      C      |   SPI-based Quickfilter Device Driver     |        -------- 
---
         |  Libraries  +-------------------------------------------+ <--- provides 
this API
         |             |  SPI Hardware Abstraction Library (HAL)   |
         +-------------+-------------------------------------------+
       @endverbatim

       @par
       This library *only* manipulates SPI hardware and its interrupt control 
registers.
       It doesn't call any other libraries.

    -# SPI ports do not have to be opened or closed before use.  All ports are 
initialized
       at startup with one call to @ref QfSpi_Init.

    -# Low Power Mode 0 (LPM0) is used.  The CPU & MCLK are stopped, but the FLL+, 
SMCLK
       and ACLK are fully active.  All enabled interrupts are acknwledged while in 
LPM0.

    -# The Quickfilter software uses a device number to distinguish between 
multiple devices.
       Only Device 0 (USART0) is implemented so far, but the interface can 
accommodate
       adding more devices.

    -# SPI ports only emit the clock when actually clocking data.

    -# The SPI baud rate is a compile-time constant, SPI0_BAUD_DIVISOR, and so is 
not
       adjustable at runtime.

  @references
     -# "MSP430x43x, MSP430x44x Mixed Signal Microcontroller (Rev. D) ", SLAS344D,
            Version Aug. 2004, Texas Instruments, 
focus.ti.com/lit/ds/symlink/msp430f449.pdf
```

```
    -# "TI MSP430x4xx Family User's Guide", SLAU056, Version E, Texas Instruments,
           focus.ti.com/lit/ug/slau056e/slau056e.pdf
    -# IAR Embedded Workbench Help, Version 3.40.2.3, www.iar.de
    -# "doxygen", Version 1.4.6, Dimitri van Heesch, doxygen.org

*///*************************************************************************
**************


#include  "Project.h"
#include  "Platform.h"
#include  "Msp430-SPI.h"                            //  Self include


/****************************************************************************
**************
    L o c a l   C o n s t a n t s ,   M a c r o s   a n d   T y p e s

*****************************************************************************
*************/

///  Modulation control value for SPI0
#define  SPI0_MODULATION_CONTROL   1


///  Baud rate divisor value for SPI0
#define  SPI0_BAUD_DIVISOR   0x8


///  Current operation being performed by the SPI port.
typedef  enum  { Uninitialized, Idle, Write, BufferedRead, UnbufferedRead }
SpiState;




/****************************************************************************
**************
    L o c a l   V a r i a b l e s

*****************************************************************************
*************/

///  SPI0 port state
static  volatile  SpiState  Spi0State = Uninitialized;



//****   S P I   0    W r i t e - R e l a t e d   D e f i n i t i o n s   ****

///  Pointer to user buffer with data to send out SPI
static  volatile  Byte * Spi0WriteBuffer;

///  Index into user write buffer
static  volatile  Count  Spi0WriteBufferIndex;

///  Total number of bytes to send from user buffer
static  volatile  Count  Spi0WriteBufferLength;
```

```
//****   S P I   0    R e a d - R e l a t e d   D e f i n i t i o n s   ****


///  Function pointer to be called in receive ISR when a byte is received
static  ReceiveCallback  Spi0ReceiveCallback;

///  Buffer to which data will be placed for unbuffered reads
static  volatile  Byte * Spi0ReadBuffer;

///  Number of bytes in @ref Spi0ReadBuffer
static  volatile  Count  Spi0ReadDataCount;

///  Number of bytes to read while in low power mode.  For an UnbufferedRead,
///  is the total number of bytes.  For BufferedRead mode, this is the number
///  of bytes in the read cycle, but buffering continues even after power mode
///  has been exited.
static  volatile  Count  Spi0ReceivesUntilWakeup;




/*****************************************************************************
**************
    L o c a l   F u n c t i o n   P r o t o t y p e s

*****************************************************************************
*************/

static Bool  UnbufferedReadCallback( Byte NewByte );
__interrupt  void  Spi0ReadIsr( void );



/*****************************************************************************
********

*****************************************************************************
*******
 ********
********
 ********                   P u b l i c   F u n c t i o n s
********
 ********
********

*****************************************************************************
*******

*****************************************************************************
*******/





/*****************************************************************************
**********//*!
```

```
  @summary  Initialize the SPI devices before first use

  @notes
    -# Call this once before calling any other functions in this module, typically
early
       in system initialization.  Also call if the device needs to be re-initializd
after
       a call to QfSpi_DeInit().

    -# The User's Guide gives the *required* initialization/re-configuration
process for
       the USART. "Failure to follow this process may result in unpredictable USART
       behavior"
          - Set SWRST
          - Initialize all USART registers with SWRST=1 (including UxCTL)
          - Enable USART module via the MEx SFRs (USPIEx)
          - Clear SWRST via software
          - Enable interrupts (optional) via the IEx SFRs (URXIEx and/or UTXIEx)

*//*****************************************************************************
*************/

void  QfSpi_Init( void )
{
    //  Sanity check SPI0 state.
    Assert( Spi0State == Uninitialized );


    SetBit( UCTL0, SWRST );           //  Hold the USART in Reset while configuring

    ClearBit( IE1, URXIE0 );          //  Disable SPI0 Receive Interrupts

    U0ME  |= UTXE0 + URXE0;           //  Enable USART0 transmit and receive modules

    P3SEL |= 0x0E;                    //  Select the SPI option for USART0 pins
(P3.1-3)

    UCTL0 |= CHAR +                   //  Character length is 8 bits
             SYNC +                   //  Synchronous Mode (as opposed to UART)
             MM;                      //  8-bit SPI Master **SWRST**

    UTCTL0 |= CKPH  +                 //  UCLK delayed by 1/2 cycle
              SSEL1 + SSEL0 +         //  Clock source is SMCLK (implies Master
mode)
                                      //         00 UCLKI              10 SMCLK
                                      //         01 ACLK  (fastest)    11 SMCLK
              STC;                    //  3-pin SPI mode (STE disabled)

    URCTL0 = 0;                       //  Receive control register

    UMCTL0 = SPI0_MODULATION_CONTROL;  //  No modulation
    UBR10  = GetHiByte( SPI0_BAUD_DIVISOR );   //  Set baud rate
    UBR00  = GetLoByte( SPI0_BAUD_DIVISOR );


    URCTL0 = 0;                       //  Init receiver contol register
    ME1  |= UTXE0 + URXE0;           //  Enable USART0 SPI transmit and receive.
(note that
                                     //  URXE0 and USPIE0 are one in the same on
the '449)

    ClearBit( UCTL0, SWRST );        //  Release USART state machine (begin
operation).
                                     //  Doesn't do anything in SPI mode until a
write
                                     //  to TXBUF0 occurs.
    Spi0State = Idle;
}




/*****************************************************************************
**********//*!

  @summary  Deinitialize the specified SPI device after last use.

  @notes
    -# Do not call any functions after calling this function, except QfSpi_Init().
       It may be called later to re-initialize the SPI port.

*//****************************************************************************
*************/

void QfSpi_DeInit( void )
{
    Assert( Spi0State != Uninitialized );

    SetBit( UCTL0, SWRST );          //  Reset USART state machine

    //  Be sure StreamRead is off.
    QfSpi_Configure( 0, SetUnbufferedReadMode, NULL, 0 );

    Spi0State = Uninitialized;
}




/*****************************************************************************
**********//*!

  @summary  Write 'Length' bytes from 'Buffer' to SPI 0

  @param[in]  Device  Device number. 0 for USART0.
  @param[in]  Buffer  Buffer with input data
  @param[in]  Length  Number of bytes to write

  @notes
    -# Transmits are interrupt-based, but not buffered.  This function returns only
       after the 'Buffer' has been sent.  The CPU is put in Idle mode during the
wait
       to save power.
    -# Data is transmitted byte-for-byte as it appears in the buffer.  Be sure the
higher
       level code handles any endian conversion before calling this function.

    -# SPI data received during the write cycle is discarded.

    -# SPI ports only emit the clock when actually transferring data.

    -# The SPI interface timing for this call is as follows.   '^'=stable on
       rising edge, all fields are msb first

       @par
       @verbatim
                                   1 1 1 1 1 1 1 1 1 1 2 2 2 2
                 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^...

          SDO    [  Buffer[0]  ] [  Buffer[1]  ] [  Buffer[2]  ]...

          SDI    x x x x x x x x x x x x x x x x x x x x x x x x...
       @endverbatim

*//***************************************************************************
*************/

void  QfSpi_Write(
    const Handle  Device,
    const Byte  * Buffer,
    const Count   Length)
{
    //  Sanity check input parameters & mode.
    Assert( Device == SPI0_HANDLE);
    AssertNonNull( Buffer );
    Assert( Length > 0);
    Assert( Spi0State == Idle );


    //  Setup info used in transmit interrupt handler
    Spi0State = Write;
    Spi0WriteBuffer = (volatile Byte *)Buffer;
    Spi0WriteBufferIndex = 1;
    Spi0WriteBufferLength = Length;

    //  Send first byte to get engine started.  Write to TXBUF0 clears UTXIFG0.
    TXBUF0 = Buffer[ 0 ];


    //  Enable SPI0 receive interrupts (clear flag before enabling)
    ClearBit( IFG1, URXIFG0 );
    SetBit( IE1, URXIE0 );

    //  Idle in low power mode 0 while receiving.
    __low_power_mode_0();

    Spi0State = Idle;
}
```

```
/*******************************************************************************
**********//*!

  @summary  Read 'Length' bytes from the SPI port and return the data in 'Buffer'.

  @param[in]  Device     Device number. 0 for USART0.
  @param[out] Buffer     Buffer to hold read data
  @param[in]  Length     Number of bytes to read.  Must be less than
                         QF_SPI_SIZE_OF_SPI_READ_BUFFER - 3.

  @return     True if buffer overflowed, false otherwise.

  @notes
    -# This function waits until Buffer is full to return.  The CPU is put in Idle
       mode during the wait to save power.

    -# Data appears in Buffer byte-for-byte as it arrives at the SPI port.  Be sure
       higher level code handles any byte-order conversion of wide data.

    -# As SPI master, MSP-430 controls the clock.

    -# The SDO pin is always low (0) during a read cycle.

    -# The SPI interface timing for this call is as follows.  '^'=stable on
       rising edge, all fields are msb first

          @verbatim
                                      1 1 1 1 1 1 1 1 1 1 2 2 2 2
                 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^...

          SDO  xx_______________________________________________...

          SDI    [  Buffer[0]  ] [  Buffer[1]  ] [  Buffer[2]  ]...
          @endverbatim

*//*****************************************************************************
*************/

Result  QfSpi_Read(
    const Handle  Device,
          Byte  * Buffer,
    const Count   Length)
{

    //  Check input parameters.
    Assert( Device == SPI0_HANDLE);
    AssertNonNull( Buffer );
    Assert( Length  >  0   &&
            Length  <=  QF_SPI_SIZE_OF_SPI_READ_BUFFER);
    Assert( Spi0State == Idle );


    //  UnbufferedRead mode requires some setup first.

    Spi0State = UnbufferedRead;
    Spi0ReceivesUntilWakeup = Length;
    Spi0ReadBuffer = Buffer;
    Spi0ReadDataCount = 0;
    Spi0ReceiveCallback = UnbufferedReadCallback;


    //  Enable SPI0 receive interrupts (clear flag then enable)
    ClearBit( IFG1, URXIFG0 );
    SetBit( IE1, URXIE0 );

    QfSpi_ResumeReceive( Device );


    //  Use low power mode while waiting for
    //  data transfer to complete.
    __low_power_mode_0();


    Spi0State = Idle;

    return  Success;
}




/***********************************************************************************
**********//*!

  @summary  Configure aspects of SPI port operation.

  @param[in]     Device   Device number. 0 for USART0.
  @param[in]     Request  Type of configuration requested.
  @param[in,out] Buffer   Any input or outout data associated with the
configuration request.
  @param[in,out] Length   Size of input or output data appropraite to the Request.

  @return        True if configuration successful, otherwise false.

*//********************************************************************************
*************/

Result  QfSpi_Configure(
    const Handle  Device,
    const QfSpi_ConfigRequest  Request,
    void  * Buffer,
    Count * Length)
{
    //  Check input parameters.
    Assert( Device == SPI0_HANDLE);
    Assert( Spi0State == Idle  ||  Spi0State == BufferedRead );
    Assert( Length == 0 );

    ClearBit( IE1, URXIE0 );        //  Disable SPI0 Receive Interrupts


    switch( Request )
    {

      //  Start reading and continuously buffering data
      case SetBufferedReadMode:
        Spi0State = BufferedRead;

        Spi0ReceiveCallback = (ReceiveCallback)Buffer;

        //  Enable SPI0 receive interrupts (clear flag then enable)
        ClearBit( IFG1, URXIFG0 );
        SetBit( IE1, URXIE0 );

        break;


      //  Set device to read bytes only as requested.  Note
      //  that this is called to stop StreamRead mode.
      case SetUnbufferedReadMode:

        Spi0State = Idle;
        break;


      //  Crash if any other Request value is used during Debug
#     if defined(Debug)
        default:   Assert(0);
#     endif
    }

    return  Success;
}




/*************************************************************************************
********

*************************************************************************************
*******
 ********
********
 ********                  U t i l i t y   F u n c t i o n s
********
 ********
********

*************************************************************************************
*******

*************************************************************************************
*******/



/*************************************************************************************
**********//*!

  @summary  Read a single byte from the SPI port.

  @param[in]   Device    Device handle
  @return   The byte that was read from the SPI port.

  @notes
    -# This function doesn't handle any error conditions, including buffer
overflow.  Only
       use it in Unbuffered mode.

*//**************************************************************************
*************/

inline
Byte  QfSpi_ReadByte( const Handle  Device )
{
    Byte  ReturnValue;

    Assert( Spi0State  !=  BufferedRead );

    QfSpi_Read( Device, &ReturnValue, 1 );

    return  ReturnValue;
}




/***************************************************************************
**********//*!

  @summary  Write a single byte 'Value' to the QF4A512 Configuration space at
'Address'.

  @param[in]   Device    Device handle
  @param[in]   Value     Byte to write to Configuration register

*//**************************************************************************
*************/

inline
void  QfSpi_WriteByte(
    const  Handle  Device,
    const  Byte    Value)
{

    QfSpi_Write( Device,  &Value,  1 );
}




/***************************************************************************
**********//*!

  @summary  Write a two-byte 'Value' to the QF4A512 Configuration space at
'Address'.

  @param[in]   Device    Device handle
  @param[in]   Value     Byte to write to Configuration register
```

```
*//*****************************************************************************
*************/

inline
void  QfSpi_WriteUInt16(
    const  Handle  Device,
           UInt16  Value)
{
    //  Data is always sent most-significant byte first, so
    //  change byte order to put MSB at low address.
    Value  = SwapUInt16Bytes( Value );

    QfSpi_Write( Device,  (Byte *)&Value,  2 );
}




/*******************************************************************************
**********//*!

  @summary  Continue receiving SPI0 bytes.

  @param[in]   Device    Device handle

  -  This function only resumes an already-configured SPI port.  The SPI port must
be
     configured before calling this function.
*//*****************************************************************************
*************/

inline
void QfSpi_ResumeReceive( const  Handle  Device )
{
    //  Check input parameters.
    Assert( Device == SPI0_HANDLE);


    //  Receive clocking is started by loading the transmit register.
    TXBUF0 = 0;
}




/*******************************************************************************
********

********************************************************************************
*******
 ********
********
 ********           L o c a l   ( S t a t i c )   F u n c t i o n s
********
 ********
********

********************************************************************************
*******
```

```
********************************************************************************
*******/



/*******************************************************************************
********

********************************************************************************
*******
 ********
********
 ********          I n t e r r u p t   S e r v i c e   R o u t i n e s
********
 ********
********

********************************************************************************
*******

********************************************************************************
*******/


//  Suppress  "Warning[Pa082]: undefined behavior: the order of volatile accesses
is
//  undefined in this statement..." that occurs below.  The order is unimportant
here.

#pragma  diag_suppress = Pa082




/*******************************************************************************
**********//*!

  @summary  SPI0 receive interrupt service routine.

  @notes
    -  The Status Register (SR) is pushed on the stack on entry to the ISR,
capturing the
       Global Interrupt Enable (GIE) bit and the power management bits.  The SR is
restored
       from the stack when the Return from Interrupt (RETI) instruction executes at
the end
       of the ISR.

       @par
       The __low_power_mode_off_on_exit() call, used below, manipulates the copy of
SR on
       the stack.

*//*****************************************************************************
************/

#pragma  vector = USART0RX_VECTOR
```

```
__interrupt  void  Spi0ReadIsr( void )
{

    if( Spi0State == Write )
    {
        //  Means of measuring frequency and length of write ISRs during Debug
        SetTestPoint( Spi0WriteIsrTestPoint );


        //  Sanity checks
        AssertNonNull( Spi0WriteBuffer );
        Assert( Spi0WriteBufferIndex  <=  Spi0WriteBufferLength );
        Assert( Spi0State == Write );


        //  If there's more data to send, load TXBUF0 to start another transmit.
        if (Spi0WriteBufferIndex  <  Spi0WriteBufferLength)
        {
            TXBUF0 = Spi0WriteBuffer[ Spi0WriteBufferIndex++ ];
        }
        else
        {
            //  Disable transmit (technically 'receive') interrupts
            ClearBit( IE1, URXIE0 );

            //  Defensive measure.  Null detected if ISR entered again.
            Spi0WriteBuffer = NULL;


            //  Exit low power mode 0, and continue foreground operation
            //  on return from this ISR
            __low_power_mode_off_on_exit();
        }


        ClearTestPoint( Spi0WriteIsrTestPoint );
    }

    else
    {
        //  Means of measuring frequency and length of read ISRs during Debug
        SetTestPoint( Spi0ReadIsrTestPoint );


        //  Sanity checks
        Assert( Spi0State == BufferedRead  ||  Spi0State == UnbufferedRead );
        Assert( Spi0ReadDataCount  <=  QF_SPI_SIZE_OF_SPI_READ_BUFFER );
        AssertNonNull( Spi0ReceiveCallback );


        if ( (*Spi0ReceiveCallback)(RXBUF0)  ==  true)
        {
            //  Start another cycle by loading the transmit register.
            TXBUF0 = 0;
        }
        else
        {
            //  Exit low power mode 0, and continue foreground operation on return
from
```

```
            //  this ISR.  (note that this does not turn off SPI reads in
BufferedMode).
            __low_power_mode_off_on_exit();
        }

        ClearTestPoint( Spi0ReadIsrTestPoint );
    }
}




/***********************************************************************************
**********//*!

  @summary  Read ISR callback used for unbuffered reads.

  -  This function is called during an interrupt service routine.  Be sure to keep
     it as short as possible and only manipulate things that are appropriate to
this context.

*//*********************************************************************************
*************/

Bool  UnbufferedReadCallback( Byte NewByte )
{

    //  Store received byte in buffer
    Spi0ReadBuffer[ Spi0ReadDataCount ] = NewByte;


    //  Increase the count of bytes in Spi0ReadBuffer
    Spi0ReadDataCount++;


    //  If wakeup counter is active, decrease the count of bytes before
    //  low-power mode is turned off.
    if (--Spi0ReceivesUntilWakeup == 0)
    {
        return  false;
    }

    return  true;
}
```

Platform.c

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Platform.c

  Abstracts a variety of operations that are specific to the way this platform
circuit
  is designed.

  $Id: Platform.c 121 2006-07-24 20:17:09Z jhopson $
```

```
*************************************************************************************
********

  @formats
      - C
      - Doxygen (comment markup to produce HTML docs)

  @dependencies
      - Operating System:  none
      - Toolset:           IAR MSP
      - Platform:          TI MSP-TS430PZ100 with QF4A512
      - CPU Architecture:  MSP-430
      - CPU Variant:       '449
      - Device:            GPIO config of this platform
      - CPU byte order:    n/a

  @notes
    -# Note that these are not MSP430 GPIO control functions per se.  These 
functions
       put the GPIO control in the context of their use on this MSP430/QF4A512 
platform.
          @verbatim
          +-------------+-------------------------------------------+
          |             |             Application Code              |         This 
file
          |  Standard   +-------------------------------------------+       --------
-----
          |      C      |   SPI-based Quickfilter Device Driver     |
          |  Libraries  +-------------------------------------------+ <--- provides 
part of
          |             |  SPI Hardware Abstraction Library (HAL)   |          this 
API
          +-------------+-------------------------------------------+
          @endverbatim

       This library *only* manipulates SPI hardware and its interrupt control 
registers, and
       calls a few Standard C library functions.  It doesn't call any other 
libraries.

  @references
     -# "QF4A512 4-Channel Programmable Signal Converter", Rev C3, Apr 06, 
Quickfilter
        Technologies, Inc., www.quickfiltertech.com/files/QF4A512revC3.pdf
     -# "TI MSP430x4xx Family User's Guide", SLAU056, Version E, Texas Instruments,
            focus.ti.com/lit/ug/slau056e/slau056e.pdf
     -# IAR Embedded Workbench Help, Version 3.40.2.3, www.iar.de
     -# "doxygen", Version 1.4.6, Dimitri van Heesch, doxygen.org

*///*********************************************************************************
**************


#include  "Project.h"
#include  "Platform.h"
```

```
/*******************************************************************************
**************
    L o c a l   C o n s t a n t s ,   M a c r o s   a n d   T y p e s

*******************************************************************************
*************/


/*******************************************************************************
**************
    L o c a l   V a r i a b l e s

*******************************************************************************
*************/

///  Callback function for the DRDY active ISR
static FuncPtr  DrdyCallbackFunc = NULL;


/*******************************************************************************
**************
    L o c a l   F u n c t i o n   P r o t o t y p e s

*******************************************************************************
*************/

void  InitClocking( void );
__interrupt void  TimerA0Isr( void );
__interrupt void  DrdyIsr( void );



/*******************************************************************************
********

*******************************************************************************
*******
 ********
********
 ********                   P u b l i c   F u n c t i o n s
********
 ********
********

*******************************************************************************
*******

*******************************************************************************
*******/




/*******************************************************************************
**********//*!

  @summary  Initialize platform-specific elements
```

```
*//*****************************************************************************
*************/

void  QfPlat_Init( void )
{
    WDTCTL = WDTPW + WDTHOLD;             //  Disable watchdog timer


    //  Verify the byte order of the platform and
    //  give a warning if it isn't little endian.
#if  defined(Debug)
    UInt16  ByteOrderTestValue  = 1;
    Byte  * ByteOrderTestArray  = (Byte *)&ByteOrderTestValue;
    Assert( ByteOrderTestArray[0]  ==  1 );
#endif


    InitClocking();                //  Setup clocks

    InitTestPointPort();

    ClearBit( P5OUT, BIT2 );              //  Clear Valid Activity Indicator
    SetBit( P5DIR, BIT2);                 //  pin and set to an output

    SetBit( P3OUT, BIT0 );                //  Initialize /CS for QF4A512 on SPI0
    SetBit( P3DIR, BIT0 );

    ClearBit( P1DIR, DRDY_BIT );          //  Set DRDY to an input


    //  Global interrupt enable
    __enable_interrupt();
}



/*****************************************************************************
**********//*!

  @summary  Initialize the device clocking.  ACLK runs off of the 32.768 KHz
crystal.  MCLK
            runs off the DCO and supplies the CPU with 8MHz.  SMCLK runs the
peripherals
            (SPI and timer) at 8MHz.

  - The DCO clock is generated internally and calibrated from the 32kHz crystal.

  - ACLK is brought out on pin P1.5 (82), MCLK is brought out out on pin P1.1 (86),
SMCLK
    is brought out on P1.4 (83).

  - The 32kHz crystal connects between pins XIN and XOUT.  Nothing is connected to
XT2IN/XT2OUT.

  - Be sure not to let the DCO go above the max frequency of the device (8MHz, in
this case).
```

```
  - The Modulator is off (SCFQ_M in SCFQCTL), so the frequency will be as constant
as possible.

  - The value written to the SCFQCTL register is automatically incremented by one.
In other
    words, to achieve a frequency-multiplication factor of 32, 31 should be written
into the
    lower seven bits of SCFQCTL.

*//*****************************************************************************
*************/

void  InitClocking( void )
{

    FLL_CTL0 = _1010_0000;
            //  |||| ||||
            //  |||| |||+----- DCOF    - clear any existing fault condition
            //  |||| ||+------ LFOF    -           "
            //  |||| |+------- XT1OF   -           "
            //  |||| +-------- XT2OF   -           "
            //  ||++---------- XCAPxPF - 8pF capacitance on XIN/XOUT
            //  |+------------ XTS_FLL - low-frequency mode
            //  +------------- DCOPLUS - use divided fDCO output


    FLL_CTL1 = _0010_0000;
            //  |||| ||||
            //  |||| ||++----- FLL_DIV  - No ACLK division on P1.5
            //  |||| |+------- SELS     - SMCLK sourced by FLL
            //  |||+-+-------- SELM     - MCLK sourced by FLL
            //  ||+----------- XT2OFF   - no secondary clock input. turn oscillator
off
            //  |+------------ SMCLKOFF - turn (leave) SMCLK on
            //  +------------- (unused)


    SCFI0   = _0101_0000;
           //  |||| ||||
           //  |||| ||++----- MODx (LSBs) - TI guy doesn't know what these are for!
           //  |||| ||                      seems to work at 0.
           //  ||++-++------- FN_x  - DCO range from 2.8 to 26.6 MHz
           //  ++------------ FLLDx - add additional x2


    SCFQCTL = _1111_1000;
           //  |||| ||||
           //  |+++-++++----- N     - Feedback loop divisor (120d)
           //  +------------- SCFQ_M  - turn modulator off


    SetBitsUsingMask( P1DIR, _0011_0010 );    //  Set P1.1, P1.4 and P1.5 as
outputs
    SetBitsUsingMask( P1SEL, _0011_0010 );    //  Set P1.1, P1.4 and P1.5 as clock
out
}
```

```
/*****************************************************************************
**********//*!

  @summary  Activate chip select for the QF4A512 attached to SPI0.

  @notes
    -# Warning!  QF4A512 logic requires DRDY to be high before taking /CS low in
Run mode.
                 DRDY will remain low in Configure mode.

*//***************************************************************************
*************/

void  QfPlat_ActivateQF4A512ChipSelect( void )
{
    ClearBit( P3OUT, BIT0 );
}



/*****************************************************************************
**********//*!

  @summary  Deactivate chip select for the QF4A512 attached to SPI0.

  @notes
    -# /CS has to be low for at least four SYS_CLK (not SCLK) during each SPI bus
cycle.
       If /CS is de-asserted too soon, DRDY might not be cleared in time for the
next
       cycle.  Although this would rarely be an issue, this code checks to be sure
DRDY
       is high before de-selecting /CS.

*//***************************************************************************
*************/

void  QfPlat_DeactivateQF4A512ChipSelect( void )
{
    SetBit( P3OUT, BIT0 );
}



/*****************************************************************************
**********//*!

  @summary  Set the direction of the DRDY pin for the QF4A512 connected to USART0.

*//***************************************************************************
*************/

void  QfPlat_SetSpi0DrdyDirection( Bool Direction )
{
    if (Direction == Input)
    {
        ClearBit( P1DIR, BIT0 );
    }
    else
    {
        SetBit( P1DIR, BIT0 );
    }
}



/*****************************************************************************
**********//*!

  @summary  Set the state of the DRDY pin for the QF4A512 connected to USART0.

*//***************************************************************************
*************/

void  QfPlat_SetSpi0DrdyState( Bool State )
{
    if (State == Active)
    {
        SetBit( P1OUT, BIT0 );
    }
    else
    {
        ClearBit( P1OUT, BIT0 );
    }
}



/*****************************************************************************
**********//*!

  @summary  Gets the state of the DRDY pin.

  @notes
    -# DRDY is on pin 1.0.

*//***************************************************************************
*************/

Bool  QfPlat_IsDrdyPinActive( void )
{
    return   IsBitSet( P1IN, BIT0 )  ?  true : false;
}



/*****************************************************************************
**********//*!

  @summary  Configure interrupts from the DRDY pin.
```

```
*//*****************************************************************************
*************/

inline
void  QfPlat_ConfigureDrdyInterrupt( FuncPtr Handler )
{

    DrdyCallbackFunc = Handler;


    ClearBit( P1SEL, DRDY_BIT );    //  DRDY pin is a GPIO only (no special
functionality)
    ClearBit( P1DIR, DRDY_BIT );    //  Set DRDY to an input
    ClearBit( P1IES, DRDY_BIT );    //  DRDY interrupt occurs on rising edge
    ClearBit( P1IFG, DRDY_BIT );    //  Reset the DRDY bit interrupt flag
}




/*****************************************************************************
**********//*!

  @summary  Enable interrupts from the DRDY pin.

*//*****************************************************************************
*************/

inline
void  QfPlat_EnableDrdyInterrupts( void )
{
    //  Only transitions, not static levels, cause interrupts.  Be sure to set flag
    //  if DRDY is already active.

    if (QfPlat_IsDrdyPinActive())
    {
        SetBit( P1IFG, DRDY_BIT );
    }

    SetBit( P1IE, DRDY_BIT );      //  Enable interrupts on DRDY pin
}




/*****************************************************************************
**********//*!

  @summary  Disable interrupts from the DRDY pin.

*//*****************************************************************************
*************/

inline
void  QfPlat_DisableDrdyInterrupts( void )
{
    ClearBit( P1IE, DRDY_BIT );      //  Disable interrupts on DRDY pin
}
```

```
/*******************************************************************************
**********//*!

  @summary  Toggle the state of the Activity LED on the platform.

*//****************************************************************************
*************/

#pragma  inline
void  QfPlat_ToggleActivityLED( void )
{
    ToggleBit( P5OUT, BIT2 );
}




///  The number of TimerA ticks in one millisecond using SMCLK
#define  TimerA0TicksPerMillisecond   0x1F98

///  Count of milliseconds until wakeup from low power mode.  Used by QfPlat_Delay
static volatile UInt16  MillisecondsUntilWakeup;


/*******************************************************************************
**********//*!

  @summary  Delay the specified number of milliseconds

  @param[in]  Milliseconds   Number of milliseconds to delay

  @notes
    -# The accuracy of the delay isn't very important, but try to err on the side
of waiting
       too long.
    -# Be sure no other interrupts will occur during the wait time.

*//****************************************************************************
*************/

void  QfPlat_DelayMs( const UInt16  Milliseconds )
{

    MillisecondsUntilWakeup = Milliseconds;

    CCTL0   =  CCIE;                          //  CCR0 interrupt enabled
    CCR0   +=  TimerA0TicksPerMillisecond;    //  Add Offset to CCR0

    TACTL = TASSEL_2 +                        //  Use SMCLK,
            MC_2;                             //  Count up countinuously.  Match with
compare
                                              //  latch (TBCL0)

    __low_power_mode_0();                     //  Conserve power while waiting
}
```

```
/************************************************************************************
********

*************************************************************************************
*******
 ********
********
 ********                    U t i l i t y   F u n c t i o n s
********
 ********
********

*************************************************************************************
*******

*************************************************************************************
*******/


/************************************************************************************
********

*************************************************************************************
*******
 ********
********
 ********               L o c a l   ( S t a t i c )   F u n c t i o n s
********
 ********
********

*************************************************************************************
*******

*************************************************************************************
*******/



/************************************************************************************
********

*************************************************************************************
*******
 ********
********
 ********              I n t e r r u p t   S e r v i c e   R o u t i n e s
********
 ********
********

*************************************************************************************
*******
```

```
*******************************************************************************
*******/


//  Suppress  "Warning[Pa082]: undefined behavior: the order of volatile accesses
is
//  undefined in this statement..." that occurs below.  The order is unimportant
here.

#pragma  diag_suppress = Pa082



/*******************************************************************************
**********//*!

  @summary  Timer A0 interrupt service routine.

  @notes
    - Services QfPlat_DelayMs()

*//*****************************************************************************
*************/

#pragma  vector = TIMERA0_VECTOR

__interrupt void  TimerA0Isr( void )
{
    SetTestPoint( TimerA0IsrTestPoint );


    if (--MillisecondsUntilWakeup == 0)
    {
        TACTL = 0;                              //  Disable TimerA0 interrupt, and
turn
                                                //  timer off to save power.
        __low_power_mode_off_on_exit();
    }
    else
    {
        CCR0  +=  TimerA0TicksPerMillisecond;   //  Add delay for next millisecond
    }


    ClearTestPoint( TimerA0IsrTestPoint );
}




/*******************************************************************************
**********//*!

  @summary  Interrupt service routine to handle transitions on the DRDY pin.

  @notes
    - The port interrupt flag has to be explicitly cleared.
    - This function is called during an interrupt service routine.  Be sure to keep
      it as short as possible and only manipulate things that are appropriate to
this context.

*//**************************************************************************************
*************/

#pragma  vector = PORT1_VECTOR

__interrupt  void  DrdyIsr( void )
{
    SetTestPoint( Spi0DrdyIsrTestPoint );


    //  Be sure DRDY bit is the one that trigerred the interrupt
    Assert( IsBitSet( P1IN, DRDY_BIT ) );
    AssertNonNull( DrdyCallbackFunc );

    //  Reset the DRDY bit interrupt flag
    ClearBit( P1IFG, DRDY_BIT );

    //  Call function using pointer supplied in call to
QfPlat_EnableDrdyInterrupts()
    DrdyCallbackFunc();


    ClearTestPoint( Spi0DrdyIsrTestPoint );
}
```

Qf4a512-access.c

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Qf4a512-access.c

  Raw data transfer functions and device mode control of QF4A512 (Small model).

  $Id: Qf4a512-access.c 121 2006-07-24 20:17:09Z jhopson $


**************************************************************************************
******

  @formats
    - C
    - Doxygen (comment markup to produce HTML docs)

  @dependencies
    - Operating System:  none
    - Toolset:           none
    - Platform:          none
    - CPU Architecture:  none
    - CPU Variant:       none
    - Device:            SPI port attached to a QF4A512
    - CPU byte order:    little endian
  @notes
    -# This portable C file implements the QF4A512 Library layer in the figure 
below.
       @par
       @verbatim
          +-------------+-------------------------------------------+          This 
file
          |             |             Application Code              |         ------
------
          |             |                                           |
          |  Standard   |            +------------------------------+
          |      C      |            |  QF4A512 Functional Driver   |
          |  Libraries  +------------+------------------------------+ <--- presents 
this API
          |             |            QF4A512 Access Driver          |
          |             +-------------------------------------------+ ---> & calls 
this API
          |             |  SPI Hardware Abstraction Library (HAL)   |     (& some 
standard C
          +-------------+-------------------------------------------+      library 
functions)
       @endverbatim

       @par
       The API supports reading and writing configuration data, reading and writing 
the on-
       board EEPROM, and reading the Run mode data stream.  The QF4A512 Functional 
Driver
       uses this module to provide more context-specific functionality, such as 
power
       management and reset management.

       @par
       This code does not directly access any hardware.  It calls the layer below 
(HAL)
       to send and receive data to and from the QF4A512.  The HAL is non-portable.

    -# This module does not control which channels are enabled.

    -# This module's interface supports multiple devices, but only Device 0 
(SPI0_HANDLE)
       is implemented at this time.

    -# The QF4A512 has three modes, Configure, Run and Eeprom.  This module 
attempts to make
       mode transitions transparent.  For example, calling @ref 
qf4a512_ReadConfigRegisters
       puts the device in Configure mode, if it's not there already.  However, 
explicit
       mode control is needed in one case, the transition out of Run mode.  If Run 
mode
       isn't explicitly exited, data will continue to be backup in the SPI buffer 
until it
       overruns.  When done with Run mode (i.e. making consecutive calls to
       qf4a512_ReadSamples), call @ref qf4a512_ExitRunMode.

    -# The EEPROM inside the QF4A512 is compatible with the ATMEL AT25320A.  The 
algorithms
```

```
       used to manage EEPROM are consistent with that device.  This module does not
support
       the write protection mechanisms of the AT25320A.

    -# The QF4A512 control registers that are more than 8-bits are stored little
endian
       (lowest byte first) when streamed in Autoincrement mode.  The Run mode data
from the
       device is big endian.

  @references
     -# "QF4A512 4-Channel Programmable Signal Converter", Rev C3, Apr 06,
Quickfilter
        Technologies, Inc., www.quickfiltertech.com/files/QF4A512revC3.pdf
     -# "SPI Serial EEPROMS", 3347J-SEEPR-10/05, Atmel Corp.,
        www.atmel.com/dyn/resources/prod_documents/doc3347.pdf
     -# "doxygen", Version 1.4.6, Dimitri van Heesch, doxygen.org

*///**********************************************************************************
**************


#include  "Project.h"
#include  "Platform.h"
#include  "Qf4a512-access.h"                      //  Self include


/***********************************************************************************
**************
    L o c a l   C o n s t a n t s ,   M a c r o s   a n d   T y p e s

***********************************************************************************
*************/


///  Mask for the Channel ID bits in the Flags byte of Run time data
#define  CHANNEL_NUM_MASK   _0110_0000

///  Number of bits to right shift the Flags byte to put
///  the Channel ID field in the bit 0 position.
#define  CHANNEL_NUM_OFFSET   5

///  Number of output frames the Device 0 read buffer can hold
#define  FRAME_CAPACITY_OF_DEVICE0_READ_BUFFER    8


////////////////////////////////////////////////////////////////////////////////////
//  EEPROM Management


///  Number of bytes in one page of EEPROM memory
#define  QF4A512_EEPROM_PAGE_SIZE   32


//  EEPROM Instruction Codes

///  Instruction to enable EEPROM writing (WREN)
#define  QF4A512_EEPROM_WRITE_ENABLE_INSTRUCTION    6
```

```
///  Instruction to disable EEPROM writing (WRDI)
#define  QF4A512_EEPROM_WRITE_DISABLE_INSTRUCTION   4

///  Instruction to read the EEPROM status register (RDSR)
#define  QF4A512_EEPROM_READ_STATUS_REGISTER_INSTRUCTION    5

///  Instruction to write the EEPROM status register (WRSR)
#define  QF4A512_EEPROM_WRITE_STATUS_REGISTER_INSTRUCTION   1

///  Instruction to read the EEPROM memory (READ)
#define  QF4A512_EEPROM_READ_INSTRUCTION    3

///  Instruction to write the EEPROM memory (WRITE)
#define  QF4A512_EEPROM_WRITE_INSTRUCTION   2



//  EEPROM Status Register Bits

///  Busy Indicator bit in the EEPROM Status register  (RDY).
#define  QF4A512_EEPROM_STATUS_BUSY_BIT     BIT0

///  Write enable bit in the EEPROM Status register  (WEN)
#define  QF4A512_EEPROM_STATUS_WRITE_ENABLE_BIT     BIT1

///  Block Protect Bit 0 in the EEPROM Status register  (BP0)
#define  QF4A512_EEPROM_STATUS_BLOCK_PROTECT_BIT0   BIT2

///  Busy Indicator Bit 1 in the EEPROM Status register  (BP1)
#define  QF4A512_EEPROM_STATUS_BLOCK_PROTECT_BIT1   BIT3

///  Write Protect Enable bit in the EEPROM Status register  (WPEN)
#define  QF4A512_EEPROM_STATUS_WRITE_PROTECT_BIT    BIT7




///  Operating modes for the QF4A512.
typedef enum  {

    Uninitialized = 0,
    Run,
    Configure,
    Eeprom

} qf4a512_Mode;




/*********************************************************************************
**************
    L o c a l   V a r i a b l e s

*********************************************************************************
*************/


///  Current operating mode of the QF4A512
```

```
static qf4a512_Mode  DeviceMode = Uninitialized;

///  Frame data buffer queue data in BufferedRead mode
static  volatile  UInt16  Device0ReadBuffer [ FRAME_CAPACITY_OF_DEVICE0_READ_BUFFER
]
                                            [ QF4A512_NUM_CHANNELS_ENABLED ];

///  Location in @ref Device0ReadBuffer where next byte will be inserted
static  volatile  Count  Device0FrameInsertIndex;

///  Location in @ref Device0ReadBuffer where next byte will be retrieved
static  volatile  Count  Device0FrameExtractIndex;

///  Number of bytes in @ref Device0ReadBuffer
static  volatile  Count  Device0FrameCount;

///  Channel number within a frame
static  volatile  Count  Device0Channel;

///  Byte number within a sample  (3 bytes per sample - Flags + 2-byte sample)
static  volatile  Count  Device0ByteNum;



/*********************************************************************************
*************
    L o c a l   F u n c t i o n   P r o t o t y p e s

*********************************************************************************
*************/

static void  PutQF4A512InConfigureMode( const Handle  Device );
static void  PutQF4A512InRunMode( const Handle  Device );
static void  PutQF4A512InEepromMode( const Handle  Device );
static Byte  GetEepromStatus( const Byte Device );
static Bool  IsEepromReady( const Byte Device );
static Bool  Device0BufferedReadCallback( Byte NewByte );
static void  Device0DrdyCallback( void );



/*********************************************************************************
********

*********************************************************************************
*******
 ********
********
 ********                  P u b l i c   F u n c t i o n s
********
 ********
********

*********************************************************************************
*******

*********************************************************************************
*******/
```

```
/*************************************************************************************
**********//*!

  @summary  Initialize all QF4A512 devices before first use

  - Call this first, before calling any other functions in this module, typcially
    during system initialization.  Also use if the devices need to be re-initializd
    after a call to @ref qf4a512_DeInit().

  - No other initizlization is required.  Devices don't need to be individually
opened.

  - This init leaves the QF4A512(s) in Configure mode, even if the AutoStart
feature
    is enabled on the device.

  - Before initializing a QF4A512, this code needs to know whether the device is
    currently in Configure or Run mode (the interface is different in each mode).
    This is done with the DRDY pin.

    @par
    First, be sure DRDY is an input to the microcontroller.  Be sure /CS is high
then
    wait at more than the longest Run mode frame time and check DRDY.  If it's low
    the device is in Configure mode.  If DRDY is high, the QF4A512 is in Run mode.

    @par
    Why do you have to wait if DRDY is low?  In Run mode DRDY is an output pin that
    will change state per the Run mode interface timing.  That timing dicates that
    DRDY go high when new data is ready (if /CS is high).  New data will always be
    ready every frame period, so DRDY will go high within that time if in Run mode.

*//**********************************************************************************
*************/

void  qf4a512_Init( void )
{
    //  Sanity check device mode.
    Assert( DeviceMode == Uninitialized );

    //  Initialize the SPI port(s) this module will use.
    QfSpi_Init();



    //  Detect existing device mode (See note above)

    QfPlat_SetSpi0DrdyDirection( Input );


    QfPlat_DelayMs(  200  /*ms*/  );


    if( QfPlat_IsDrdyPinActive()  ==  true)
    {
        DeviceMode = Run;
        PutQF4A512InConfigureMode( SPI0_HANDLE );
    }

    else  // Device is in Config mode
    {
        //  Config drives the DRDY pin as low output
        QfPlat_SetSpi0DrdyState( Low );
        QfPlat_SetSpi0DrdyDirection( Output );

        DeviceMode = Configure;


        //  To access coefficient memory in Configure mode, the SPI_CTRL[
ram_run_mode ]
        //  bit must be cleared so the coefficient area can synchronize to the SPI
clock,
        //  not the internal clock (clk_sys).
        //
        //  Warning!  Be sure 'DeviceMode = Configure' before calling any
qf4a512_...
        //            functions (as done below), or mayhem might ensue.

        Byte  CurrentSpiCtrl = qf4a512_ReadConfigByte( SPI0_HANDLE,
QF4A512_SPI_CTRL_ADDRESS );

        SetBit( CurrentSpiCtrl, BIT0 );
        qf4a512_WriteConfigByte(  SPI0_HANDLE,  QF4A512_SPI_CTRL_ADDRESS,
CurrentSpiCtrl );
    }


    //  Sanity check the target device type.
    Assert(qf4a512_ReadConfigByte( SPI0_HANDLE, QF4A512_GLBL_ID_ADDRESS )  ==
QF4A512_CHIP_ID_NUMBER   &&
           qf4a512_ReadConfigByte( SPI0_HANDLE, QF4A512_DIE_REV_ADDRESS )  >=
QF4A512_MINIMUM_DIE_REV_NUMBER);


    //  Set the autoincrement bit in SPI_CTRL so sequences > 1 byte can be
sent/received.
    qf4a512_WriteConfigByte( SPI0_HANDLE, QF4A512_SPI_CTRL_ADDRESS,  _0000_1000 );


    // Verify that autoincrement got turned on
    Assert( qf4a512_ReadConfigByte( SPI0_HANDLE, QF4A512_SPI_CTRL_ADDRESS )  &
_0000_1000 );

}




/*********************************************************************************
**********//*!

  @summary  Deinitialize all QF4A512 devices after last use.
  - Do not call any functions other than @ref qf4a512_Init() after calling this
function.

  - The device may be in any power state prior to entering this function.

  - Upon return, the device should consume the same or less power than the Sleep
power state.

  - Devices don't need to be individually closed.

*//*******************************************************************************
*************/

void  qf4a512_DeInit( void )
{
    //  Sanity check device mode.
    Assert( DeviceMode != Uninitialized );

    QfSpi_DeInit();

    DeviceMode = Uninitialized;
}




/*********************************************************************************
**********//*!

  @summary  Reads 'Length' Run mode samples from the QF4A512 into 'Buffer'.

  @param[in]  Device    Device handle
  @param[out] Buffer    Receives frames of 16-bit sample data
  @param[in]  Length    Number of frames of data to read

  @return   Always Success, for now.


  - Warning!  This function only works when all channels are active and sampling at
    the same frequency.  If the channels have different sampling frequencies, this
    function provides no means of knowing which samples are valid for a given
frame.
    (single-channel high-speed mode is not supported)

*//*******************************************************************************
*************/

Result  qf4a512_ReadSamples(
    const Handle   Device,
          UInt16   Buffer [][ QF4A512_NUM_CHANNELS_ENABLED ],
    const Count    Length)
{
    Count  Frame;

    //  Sanity check input parameters
    Assert( Device == SPI0_HANDLE);
    AssertNonNull( Buffer );
    Assert( Length > 0 );
    //  If not in Run mode, go there now
    if (DeviceMode != Run)
    {
        PutQF4A512InRunMode( Device );
    }



    for( Frame = 0;
         Frame < Length;
         Frame++ )
    {
        Count  Channel;

        if( Device0FrameCount == 0 )
        {
            //  Use low power mode while waiting for the next frame.
            __low_power_mode_0();
        }


        //  Store the received data in the Caller's buffer.

        for( Channel = 0;
             Channel < QF4A512_NUM_CHANNELS_ENABLED;
             Channel++ )
        {
            Buffer[ Frame ][ Channel ]  =  Device0ReadBuffer[
Device0FrameExtractIndex ][ Channel ];
        }


        //  Advance frame index to next position.  Wrap to the
        //  the beginning if it goes beyond the end.
        Device0FrameExtractIndex++;
        Device0FrameExtractIndex  %=  FRAME_CAPACITY_OF_DEVICE0_READ_BUFFER;


        //  Decrease the count of frames in Device0ReadBuffer
        Device0FrameCount--;
    }


    return  Success;
}




/*************************************************************************************
**********//*!

  @summary  Read 'Length' configuration registers into 'Buffer' starting at
'StartingAddr'.

  @param[in]  Device       Device handle
  @param[in]  StartingAddr Register or EEPROM address at which to start writing
data
```

```
  @param[out] Buffer       Pointer to data to be written
  @param[in]  Length       Number of bytes in WriteBuff to write

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - The QF4A512 interface timing for reading configuration registers is as follows.
    All fields are msb first and '^'=stable on rising edge.

      @verbatim
                                 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
3 3 3 3 3
             0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 9
      SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
^-^-^-^-^-^...

      SDO    1 [------ StartingAddr -----] x x x x x x x x x x x x x x x x x x x x
x x x x x...

      SDI    x x x x x x x x x x x x x x x x [  Buffer[0]  ] [  Buffer[1]  ] [
Buffer[2]  ]...
            _
      /CS
|________________________________________________________________________________..
.|

      /DRDY
________________________________________________________________________________...
      @endverbatim


*//*****************************************************************************
*************/

void  qf4a512_ReadConfigRegisters(
    const Handle  Device,
          Count   StartingAddr,
          Byte  * Buffer,
    const Count   Length)
{

    //  Sanity check input parameters
    Assert( Device   ==  SPI0_HANDLE);
    Assert( StartingAddr + Length  <=  QF4A512_MAX_REGISTER_ADDRESS );
    AssertNonNull( Buffer );
    Assert( Length  >  0 );


    //  Put device in Configure mode if not there already
    if (DeviceMode != Configure)
    {
        PutQF4A512InConfigureMode( Device );
    }


    //  Prepare address.
    StartingAddr <<= 1;                              //  Address is offset by 1
    SetBit( StartingAddr, BIT15 );                    //  1 in msb indicates read


    //  QF4A512 read configuration data frame
    QfPlat_ActivateQF4A512ChipSelect();               //  1. Activate /CS
    QfSpi_WriteUInt16( Device,  StartingAddr );       //  2. Send address
    QfSpi_Read( Device, Buffer, Length );             //  3. Read the data
    QfPlat_DeactivateQF4A512ChipSelect();             //  4. Deactivate /CS
}




/*************************************************************************************
**********//*!

  @summary  Write 'Length' configuration registers from 'Buffer' starting at
'StartingAddr'.

  @param[in]  Device       Device handle
  @param[in]  StartingAddr Register or EEPROM address at which to start writing
data
  @param[in]  Buffer       Pointer to data to be written
  @param[in]  Length       Number of bytes in WriteBuff to write

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - The QF4A512 interface timing for writing configuration registers is as follows.
    All fields are msb first and '^'=stable on rising edge.

     @verbatim
                                 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
3 3 3 3 3
             0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 9
      SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
^-^-^-^-^...

      SDO    0 [------ StartingAddr -----] x [  Buffer[0]  ] [  Buffer[1]  ] [
Buffer[2]  ]...

      SDI    x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x...
              _
      /CS
|________________________________________________________________________________..
.|

      /DRDY
________________________________________________________________________________...
      @endverbatim


*//***********************************************************************************
*************/

Result  qf4a512_WriteConfigRegisters(
    const Handle  Device,
          Count   StartingAddr,
    const Byte  * Buffer,
    const Count   Length)
{

    //  Sanity check input parameters
    Assert( Device  ==  SPI0_HANDLE);
    Assert( StartingAddr + Length  <=  QF4A512_MAX_REGISTER_ADDRESS );
    AssertNonNull( Buffer );
    Assert( Length  >  0 );

    //  Put device in Configure mode if not there already
    if (DeviceMode != Configure)
    {
        PutQF4A512InConfigureMode( Device );
    }


    //  Address to device is offset 1 bit to the left
    StartingAddr <<= 1;


    //  QF4A512 writeconfiguration data frame
    QfPlat_ActivateQF4A512ChipSelect();             //  1. Activate /CS
    QfSpi_WriteUInt16( Device,  StartingAddr );     //  2. Send address
    QfSpi_Write( Device, Buffer,  Length);          //  3. Write the data
    QfPlat_DeactivateQF4A512ChipSelect();           //  4. Deactivate /CS

    return  Success;
}





/*************************************************************************************
**********//*!

  @summary  Read 'Length' EEPROM bytes into 'Buffer' starting at 'StartingAddr'.

  @param[in]  Device       Device handle
  @param[in]  StartingAddr Register or EEPROM address at which to start writing
data
  @param[out] Buffer       Pointer to data to be written
  @param[in]  Length       Number of bytes in WriteBuff to write

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - The EEPROM read interface is as follows.  All fields are msb first and
'^'=stable
    on rising edge.

      @verbatim
                                 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
3 3 3 3 3
```

```
              0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 9
      SCLK    ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
^-^-^-^-^...

      SDO     0 0 0 0 x 0 1 1 [-------- StartingAddr -------] x x x x x x x x x x x
x x x x x...

      SDI     x x x x x x x x x x x x x x x x x x x x x x x x [  Buffer[0]  ] [
Buffer[1]  ]...
            _
      /CS
|________________________________________________________________________________..
.|

________________________________________________________________________________
      /DRDY
...
      @endverbatim

  - The Address field of the EEPROM is in little endian byte order, which is
opposite
    from the Configuration registers of the QF4A512.  However, this module handles
the
    reversal (e.g. there's no need to alter StartingAddr).

*//***********************************************************************************
*************/
/*
Result  qf4a512_ReadEeprom(
    const Handle  Device,
    const Count   StartingAddr,
          Byte  * Buffer,
    const Count   Length)
{

    //  Sanity check input parameters
    Assert( Device == SPI0_HANDLE);
    Assert( StartingAddr + Length  <=  QF4A512_MAX_EEPROM_ADDRESS );
    AssertNonNull( Buffer );
    Assert( Length > 0 );


    //  Put device in Configure mode if not there already
    if (DeviceMode != Eeprom)
    {
        PutQF4A512InEepromMode( Device );
    }


    //  Wait for EEPROM to be ready.
    WaitFor( IsEepromReady( Device )  ==  true );


    //  Atmel AT25320A EEPROM read cycle
    QfPlat_ActivateQF4A512ChipSelect();                             // 1. Activate
/CS
    QfSpi_WriteByte( Device,  QF4A512_EEPROM_READ_INSTRUCTION  );  // 2. Send the
Read instruction
```

```
    QfSpi_WriteUInt16( Device,  StartingAddr );                     // 3. Send the
address

    QfSpi_Read( Device,  Buffer,  Length);                          // 4. Read the
data
    QfPlat_DeactivateQF4A512ChipSelect();                           // 5. Deactivate
/CS

    return  Success;
}




/*************************************************************************************
**********//*!

  @summary  Write 'Length' EEPROM bytes from 'Buffer' starting at 'StartingAddr'.

  @param[in]  Device       Device handle
  @param[in]  StartingAddr Register or EEPROM address at which to start writing
data
  @param[in]  Buffer       Pointer to data to be written
  @param[in]  Length       Number of bytes in WriteBuff to write

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - The time between CS low and the first SCLK edge is not critical to the device.

  - Data received while performing this write is discarded.

  - EEPROM mode is distinct from Run mode.

  - This function is not aware of the means used by the lower-level HAL to access
    the QF4A512.  It may be interrupt-based, DMA-based or polled.

  - In the diagrams below all fields are msb first and '^'=stable on rising edge.

  - The interface timing for writing EEPROM is as follows.

      @verbatim
                                1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
3 3 3 3 3
             0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 9
      SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
^-^-^-^-^...

      SDO    0 0 0 0 x 0 1 0 [-------  StartingAddr  ------] [  Buffer[0]  ] [
Buffer[1]  ]...

      SDI    x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x...
          _
_
```

```
      /CS
|_______________________________________________________________________________..
.|

________________________________________________________________________________
      /DRDY
...
      @endverbatim

   - Although the time between CS low and the first SCLK edge is not critical, the
     low-to-high transition of the CS pin must occur during the SCK low-time
immediately
     after clocking in the D0 (LSB) data bit.

   - After each data byte is received, the EEPROM is received, the five low-order
     address bits are internally incremented by one; the hig-order bits of the
     address will remain constant.  So, to write 32 bytes at a time, start on a
     32-byte boundary.

   - The EEPROM is automatically returned to the write disable state at the
completion
     of a write cycle.

   - The Address field of the EEPROM is in little endian byte order, which is
opposite
     from the Configuration registers of the QF4A512.  However, this module handles
     the reversal (e.g. there's no need to alter StartingAddr).

*//*****************************************************************************
*************/
/*
Result  qf4a512_WriteEeprom(
    const Handle  Device,
    const Count   StartingAddr,
    const Byte  * Buffer,
    const Count   Length)
{

    ///  Offset from StartingAddress from which data will be written on current
cycle.
    Count  AddressOffset = 0;

    ///  EEPROM address to which data will be written on the current cycle
    Count  CurrentAddress;

    Count  NumRemainingBytes;         ///<  Number of bytes still to be written
    Count  BytesToWriteThisCycle;     ///<  Number of bytes to write on the current
cycle



    //  Sanity check input parameters
    Assert( Device == SPI0_HANDLE);
    Assert( StartingAddr + Length  <=  QF4A512_MAX_EEPROM_ADDRESS );
    AssertNonNull( Buffer );
    Assert( Length > 0 );


    //  Put device in Configure mode if not there already
    if (DeviceMode != Eeprom)
    {
        PutQF4A512InEepromMode( Device );
    }



    //  Wait for EEPROM to be ready.
    WaitFor( IsEepromReady( Device )  ==  true );



    //  The EEPROM works on 32-byte pages, starting at 32-byte boundaries (e.g. the
lower 5
    //  bits are zero).  Loop through as many pages as needed to write all the
data, taking
    //  into account the odd sizes that may occur in the first and/or last pages.

    for( NumRemainingBytes = Length;
         NumRemainingBytes > 0;
         NumRemainingBytes -= BytesToWriteThisCycle )
    {

        CurrentAddress = StartingAddr + AddressOffset;


        //  If address doesn't fall on a page boundary, calculate the number of
bytes from
        //  the address up to the next page boundary.  (note that this will only
occur on
        //  the first write cycle).

        if ( CurrentAddress % QF4A512_EEPROM_PAGE_SIZE   >  0)
        {
            BytesToWriteThisCycle  =  QF4A512_EEPROM_PAGE_SIZE -
                                      (CurrentAddress % QF4A512_EEPROM_PAGE_SIZE);


            //  Also look for the case where the data both starts
            //  and ends within the bounds of a single page.
            if (NumRemainingBytes  <  BytesToWriteThisCycle)
            {
                BytesToWriteThisCycle  =  NumRemainingBytes;
            }
        }


        //  If there is less than one full page of data remaining, the data count
is the
        //  size of the write.  (note that this will only occur on the last write
cycle).

        else if (NumRemainingBytes  <  QF4A512_EEPROM_PAGE_SIZE)
        {
            BytesToWriteThisCycle  =  NumRemainingBytes;
        }
        //  All writes other than the special cases above are one full page in 
length.
        //  (note that this can occur on any write cycle, including the first or 
last).
        else
        {
            BytesToWriteThisCycle  =  QF4A512_EEPROM_PAGE_SIZE;
        }



        //  Sanity check the parameters for this write cycle
        Assert( CurrentAddress >= StartingAddr  &&
                CurrentAddress <  StartingAddr + Length );
        Assert( BytesToWriteThisCycle  <=  QF4A512_EEPROM_PAGE_SIZE );
        Assert( (GetEepromStatus( Device ) &  0x8c)  == 0 );  // Be sure write 
protection is off



        //  Atmel AT25320A EEPROM write cycle.
        //
        //     1. Activate /CS
        //     2. Enable writes with the WREN instruction.
        //     3. Deactivate /CS

        QfPlat_ActivateQF4A512ChipSelect();
        QfSpi_WriteByte( Device,  QF4A512_EEPROM_WRITE_ENABLE_INSTRUCTION );
        QfPlat_DeactivateQF4A512ChipSelect();

        Assert( (GetEepromStatus( Device ) & 2) == 2 );   // Double-check write 
enable


        //     4. Activate /CS
        //     5. Send the Write instruction
        //     6. Send the address
        //     7. Send the data
        //     8. Deactivate /CS

        QfPlat_ActivateQF4A512ChipSelect();
        QfSpi_WriteByte( Device,  QF4A512_EEPROM_WRITE_INSTRUCTION );
        QfSpi_WriteUInt16( Device,  CurrentAddress );
        QfSpi_Write( Device,  Buffer + AddressOffset,  BytesToWriteThisCycle );
        QfPlat_DeactivateQF4A512ChipSelect();


        //     9. Wait for Status register to indicate 'Ready'.

        WaitFor( IsEepromReady( Device ) == true );


        //  Adjust the write address for the next cycle
        AddressOffset  +=  BytesToWriteThisCycle;
    }


    return  Success;
}
```

```
*/

/*******************************************************************************
********

*******************************************************************************
*******
 ********
********
 ********                  U t i l i t y   F u n c t i o n s
********
 ********
********

*******************************************************************************
*******

*******************************************************************************
*******/



/*******************************************************************************
**********//*!

  @summary  Read a single byte from the QF4A512 Configuration space at 'Address'.

  @param[in]   Device    Device handle
  @param[in]   Address   Device address from which byte will be read

  @return   The byte that was read from the QF4A512.

*//*****************************************************************************
*************/

inline
Byte  qf4a512_ReadConfigByte(
    const  Handle  Device,
    const  Count   Address )
{
    Byte    ConfigValue;

    //  Get the requested configuration byte
    qf4a512_ReadConfigRegisters(
       Device,
       Address,
       &ConfigValue,
       1);

    return ConfigValue;
}
```

```
/*************************************************************************************
**********//*!

  @summary  Write a single byte 'Value' to the QF4A512 Configuration space at
'Address'.

  @param[in]   Device    Device handle
  @param[in]   Address   Device address from which byte will be read
  @param[in]   Value     Byte to write to Configuration register

*//**********************************************************************************
*************/

inline
void  qf4a512_WriteConfigByte(
    const  Handle  Device,
    const  Count   Address,
    const  Byte    Value)
{
    qf4a512_WriteConfigRegisters(
        Device,
        Address,
        &Value,
        1);
}




/*************************************************************************************
**********//*!

  @summary  Read a single byte from the QF4A512 EEPROM at 'Address'.

  @param[in]   Device    Device handle
  @param[in]   Address   EEPROM address from which byte will be read

  @return   The byte that was read from the QF4A512 EEPROM.

*//**********************************************************************************
*************/
/*
inline
Byte  qf4a512_ReadEepromByte(
    const  Handle  Device,
    const  Count   Address )
{
    Byte    EepromValue;

    //  Get the requested EEPROM byte
    qf4a512_ReadEeprom(
       Device,
       Address,
       &EepromValue,
       1);

    return EepromValue;
}
```

```
*/

/*******************************************************************************
**********//*!

  @summary  Write a single byte to the QF4A512 EEPROM at 'Address'.

  @param[in]   Device    Device handle
  @param[in]   Address   EEPROM address to which byte will be written
  @param[in]   Value     Byte to write to Configuration register

*//*****************************************************************************
*************/
/*
inline
void  qf4a512_WriteEepromByte(
    const  Handle  Device,
    const  Count   Address,
    const  Byte    Value)
{
    qf4a512_WriteEeprom(
        Device,
        Address,
        &Value,
        1);
}


*/

/*******************************************************************************
**********//*!

  @summary  Stop the continuous sampling of data in Run mode.

  @param[in]  Device  Device handle

  - If this function isn't called after Run mode is finished, the SPI read buffer 
will
    overflow.

*//*****************************************************************************
*************/

inline
void  qf4a512_ExitRunMode( const Handle  Device )
{
    PutQF4A512InConfigureMode( Device );
}




/*******************************************************************************
********

********************************************************************************
*******
```

```
 ********
********
 ********            L o c a l   ( S t a t i c )   F u n c t i o n s
********
 ********
********

*****************************************************************************************
*******

*****************************************************************************************
*******/




/****************************************************************************************
**********//*!

  @summary  Put the specified QF4A512 in Run mode

  @param[in]  Device       Device handle

  - It's OK if the device is already in Run mode.

  - qf4a512_Init() calls this function before this driver is fully operational.  It
assumes
    things are done in the manner below.  Check that interaction before changing
this function.

*//**************************************************************************************
*************/

static void  PutQF4A512InRunMode( const Handle  Device )
{

    //  Sanity check Device handle
    Assert( Device == SPI0_HANDLE);

    if (DeviceMode == Run)
    {
        return;
    }

    //  Device has to be configured before Run mode.  If in
    //  Eeprom mode, setup for configuration first.
    else if (DeviceMode != Configure)
    {
        PutQF4A512InConfigureMode( Device );
    }


    //  The SPI_CTRL[ ram_run_mode ] bit must be cleared so the coefficient memory
    //  area can synchronize to the internal clock (clk_sys), not the SPI clock, in
    //  Run mode.
    //
    //  This also sets multi-channel mode, turns parity off, enables autoincrement,
    //  and sets Channel 1 as the fast channel (the one that drives DRDY)
    qf4a512_WriteConfigByte( Device,  QF4A512_SPI_CTRL_ADDRESS,  _0000_1001 );


    //  DRDY is an input in Run mode
    QfPlat_SetSpi0DrdyDirection( Input );


    //  Start Run mode
    qf4a512_WriteConfigByte( Device,  QF4A512_RUN_MODE_ADDRESS,  _0000_0001 );


    //  From here on, until the device is put back in Configure or Eeprom mode, the
    //  QF4A512 interface operates differently.  See the QF4A512 datasheet.


    //  Initialize circular frame buffer parameters
    Device0FrameInsertIndex = 0;
    Device0FrameExtractIndex = 0;
    Device0FrameCount = 0;


    //  Turn on buffering for the SPI port.  This sets up, but doesn't start data
    //  capture.
    QfSpi_Configure(
        Device,
        SetBufferedReadMode,                     //  Specify Run mode
        (void *)Device0BufferedReadCallback,    //  Call this when each byte arrives
        0 );


    //  Configure DRDY pin interrupts.  A low-to-high on DRDY will start data
capture.
    QfPlat_ConfigureDrdyInterrupt( Device0DrdyCallback );

    //  Catch the next high transition of DRDY
    QfPlat_EnableDrdyInterrupts();

    DeviceMode = Run;
}




/*************************************************************************************
**********//*!

  @summary  Put the specified QF4A512 in Configure mode

  @param[in]  Device   Device handle

  - It's OK if the device is already in Configure mode.

*//*************************************************************************************
*************/

static void  PutQF4A512InConfigureMode( const Handle  Device )
{

    //  Sanity check Device handle
    Assert( Device == SPI0_HANDLE);


    if (DeviceMode == Configure)
    {
        return;
    }

    else if (DeviceMode == Run)
    {
        ///  Send this to a device in Run mode to put it in Configure mode
        static const Byte  ExitRunMode[]  = { QF4A512_RUN_MODE_ADDRESS,  0,  0 };

        QfPlat_DisableDrdyInterrupts();

        QfSpi_Configure( Device, SetUnbufferedReadMode, NULL, 0 );

        //  Get device back to Configure mode

        QfPlat_ActivateQF4A512ChipSelect();             //  1. Activate /CS
        QfSpi_Write( Device, (Byte *)&ExitRunMode, 3 ); //  2. Send command data
        QfPlat_DeactivateQF4A512ChipSelect();           //  3. Deactivate /CS
    }


    //  This mode drives the DRDY pin as low output
    QfPlat_SetSpi0DrdyState( Low );
    QfPlat_SetSpi0DrdyDirection( Output );

    DeviceMode = Configure;


    //  To access coefficient memory in Configure mode, the SPI_CTRL[ ram_run_mode
]
    //  bit must be cleared so the coefficient area can synchronize to the SPI
clock,
    //  not the internal clock (clk_sys).
    //
    //  Warning!  Be sure 'DeviceMode = Configure' before calling any qf4a512_...
    //            functions (as done below), or mayhem might ensue.

    Byte  CurrentSpiCtrl = qf4a512_ReadConfigByte( Device, QF4A512_SPI_CTRL_ADDRESS
);

    SetBit( CurrentSpiCtrl, BIT0 );
    qf4a512_WriteConfigByte( Device, QF4A512_SPI_CTRL_ADDRESS,  CurrentSpiCtrl );

}




/***********************************************************************************
**********//*!

  @summary  Put the specified QF4A512 in Eeprom mode

  @param[in]  Device   Device handle
  - It's OK if the device is already in Eeprom mode.

*//*****************************************************************************
*************/
/*
static void  PutQF4A512InEepromMode( const Handle  Device )
{

    //  Sanity check Device handle
    Assert( Device == SPI0_HANDLE);


    if (DeviceMode == Eeprom)
    {
        return;
    }

    //  If device isn't in Configure mode, get it there first.
    //  From there, just set DRDY high to get to Eeprom mode.
    else if (DeviceMode != Configure)
    {
        PutQF4A512InConfigureMode( Device );
    }


    //  This mode drives the DRDY pin as low output
    QfPlat_SetSpi0DrdyState( High );


    DeviceMode = Eeprom;
}

*/


/*****************************************************************************
**********//*!

  @summary  Read the Status register of the specified QF4A512 device.

  @param[in]  Device   Device handle

  @return   Status byte of EEPROM device, verbatim

  - EEPROM read status register (RDSR) timing.

      @verbatim
                                   1 1 1 1 1 1
                0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
      SCLK(o)   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^

      SDO(o)    0 0 0 0 x 1 0 1 x x x x x x x x

      SDI(i)   x__hi_impedance__[-- data 1 ---]
             _                                 _
      /CS(o)  |________________________________|
               ________________________________
      /DRDY(o)
      @endverbatim
```

```
*//*****************************************************************************
*************/
/*
static Byte  GetEepromStatus( const Byte Device )
{
    Byte  EepromStatus;

    //  Sanity check Device handle
    Assert( Device == SPI0_HANDLE );


    //  EEPROM Read Status Byte frame

    QfPlat_ActivateQF4A512ChipSelect();                     // 1. Activate /CS
    QfSpi_WriteByte( Device,                                // 2. Send Read Status
Command
        QF4A512_EEPROM_READ_STATUS_REGISTER_INSTRUCTION );
    EepromStatus = QfSpi_ReadByte( Device );                // 3. Read the Status
byte
    QfPlat_DeactivateQF4A512ChipSelect();                   // 4. Deactivate /CS


    return  EepromStatus;
}

*/


/*****************************************************************************
**********//*!

  @summary  Checks the EEPROM Status register and returns true if the Ready bit is
set

  @param[in]  Device   Device handle

  @return  True of the Ready bit is set in the Status register, false if not

*//*****************************************************************************
*************/
/*
static Bool  IsEepromReady( const Byte Device )
{
    Byte EepromStatus;

    //  Sanity check Device handle
    Assert( Device == SPI0_HANDLE);


    EepromStatus = GetEepromStatus( Device );

    return  IsBitClear( EepromStatus, QF4A512_EEPROM_STATUS_BUSY_BIT );
}


*/
```

```
/***************************************************************************************
**********//*!

  @summary  Called when the DRDY pin goes high

  -  This function is called during an interrupt service routine.  Be sure to keep
     it short and only manipulate things that are appropriate to this context.
*//*************************************************************************************
*************/

static void  Device0DrdyCallback( void )
{
    //  Initialize frame buffer parameters
    Device0ByteNum = 1;
    Device0Channel = 0;

    //  Don't allow DRDY interrupts while reading SPI
    QfPlat_DisableDrdyInterrupts();

    //  Activate Chip Select  (/CS)
    QfPlat_ActivateQF4A512ChipSelect();

    //  SPI receive is already configured, it just needs to be resumed.
    QfSpi_ResumeReceive( SPI0_HANDLE );
}




//  Suppress  "Warning[Pa082]: undefined behavior: the order of volatile accesses
is
//  undefined in this statement..." that occurs below.  The order is unimportant
here.

#pragma  diag_suppress = Pa082




/***************************************************************************************
**********//*!

  @summary  Called by the Device0 read ISR every time a byte is received (in
Buffered mode).

  - This function is called during an interrupt service routine.  Be sure to keep
    it as short as possible and only manipulate things that are appropriate to this
context.

  - Since the 16-bit runtime sample data comes out msb first, and QfSpi_Read()
returns bytes
    in the order they are received, the samples in Data are in big endian byte
order.  The
    MakeUInt16 macro hides the byte swapping, but be aware of that it does occur.

  - Note that the New bit, in the Flags byte, isn't checked because this function
assumes
    all channels are sampling at the same frequency.  If DRDY goes high for the
'fastest'
```

```
    channel, the others must have new data too.

  - The time between CS low and the first SCLK edge is not critical to the device.

  - Run mode data comes out of the QF4A512 as a set of 3-byte frames, as shown
below,
    with one frame for each active channel.  Note that every active channel
produces
    a frame, even if the channels have dissimilar sampling rates and a channel has
no
    new data (see the New Data bit).

    @verbatim
                              1 1 1 1 1 1 1 1 1 1 2 2 2 2
             0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
      SCLK   ^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^...

      SDO    [-- address --] [--- data ----] x x x x x x x x

      SDI    0 1 2 3 [rsvd ] [-------- channel data -------]
             ^ ^ ^ ^
             | | | New Data
             | | Channel ID 0
             | Channel ID 1
             Odd Parity
            _                                                 _
      /CS    |_________________________________________________|

      /DRDY _________________________________________________...

    @endverbatim

*//*******************************************************************************
*************/

static Bool  Device0BufferedReadCallback( Byte NewByte )
{
    ///  High byte of the current sample
    static volatile Byte  HighByte;


    //  Handling varies depending on which byte is being processed

    if (Device0ByteNum  ==  1)         //  Flags byte (ignored)
    {
        //  Verify the channel number in Flags corresponds with our value
        Assert( (NewByte & CHANNEL_NUM_MASK) >> CHANNEL_NUM_OFFSET  ==
Device0Channel );

        //  Verify the New bit is set
        Assert( IsBitSet( NewByte, QF4A512_NEW_DATA_BIT ) );

        Device0ByteNum = 2;
    }

    else if (Device0ByteNum  ==  2)    //  High byte of sample (save for later)
    {
        HighByte = NewByte;
        Device0ByteNum = 3;
    }

    else if (Device0ByteNum  ==  3)    //  Low byte of sample
    {
        Device0ReadBuffer[ Device0FrameInsertIndex ][ Device0Channel ] =
MakeUInt16( HighByte, NewByte);
        Device0ByteNum = 1;


        //  Advance sample count
        Device0Channel++;


        //  Perform extra processing at the end of the frame
        if (Device0Channel  ==  QF4A512_NUM_CHANNELS_ENABLED)
        {

            //  Cycle /CS.  Note - /CS has to cycle between frames for
            //  new data to be latched into the output shift register.
            //  If not cycled, the existing samples will be repeated.
            QfPlat_DeactivateQF4A512ChipSelect();


            //  Advance frame index to next position.  Wrap to the
            //  the beginning if it goes beyond the end.
            Device0FrameInsertIndex++;
            Device0FrameInsertIndex  %=  FRAME_CAPACITY_OF_DEVICE0_READ_BUFFER;

            //  Increase the count of frames in Device0ReadBuffer
            Device0FrameCount++;

            //  Be sure buffer isn't being overflowed
            Assert( Device0FrameCount  <  FRAME_CAPACITY_OF_DEVICE0_READ_BUFFER );

            //  Catch the next high transition of DRDY.
            QfPlat_EnableDrdyInterrupts();

            return  false;
        }

    }

    return  true;
}
```

Qf4a512-functional.c

```
/*****************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Qf4a512-functional.c

  Implementation of high-level Qf4a512 features (Small model).

  $Id: Qf4a512-functional.c 121 2006-07-24 20:17:09Z jhopson $

*******************************************************************************
******
```

```
  @formats
      - C
      - Doxygen (comment markup to produce HTML docs)

  @dependencies
      - Operating System:  none
      - Toolset:           none
      - Platform:          none
      - CPU Architecture:  none
      - CPU Variant:       none
      - Device:            QF4A512, SPI port + portable SPI API
      - CPU byte order:    little endian

    -# This portable C file implements the QF4A512 Library layer in the figure
below.
            @verbatim
              +-------------+------------------------------------------+
This file
              |             |             Application Code             |         --
----------
              |             |                                          |
              |  Standard   |           +------------------------------+ <---
presents this API
              |      C      |           |  QF4A512 Functional Driver  |
              |  Libraries  +-----------+------------------------------+ ---> &
calls this API
              |             |           QF4A512 Access Driver          |     (&
some standard C
              |             +------------------------------------------+
library functions)
              |             |  SPI Hardware Abstraction Library (HAL)  |
              +-------------+------------------------------------------+
            @endverbatim

       This file is the front-line API for an applications programmer controlling a
QF4A512.
       Although the lower level API (@ref Qf4a512-access.c) could be called
directly from an
       application, it is intended that the functions in this file be used where
possible.

       This code does not directly access any hardware.  It calls the access driver
to send
       and receive data to and from the QF4A512.

    -# This code does not support device calibration.


  @references
     -# "QF4A512 4-Channel Programmable Signal Converter", Rev C3, Apr 06,
Quickfilter
        Technologies, Inc., www.quickfiltertech.com/files/QF4A512revC3.pdf
     -# "SPI Serial EEPROMS", 3347J-SEEPR-10/05, Atmel Corp.,
        www.atmel.com/dyn/resources/prod_documents/doc3347.pdf
     -# "doxygen", Version 1.4.6, Dimitri van Heesch, doxygen.org

*///*******************************************************************************
**************
```

```
#include  "Project.h"
#include  "Platform.h"
#include  "Qf4a512-access.h"
#include  "Qf4a512-functional.h"                    //  Self include


/**********************************************************************************
**************
    L o c a l   C o n s t a n t s ,   M a c r o s   a n d   T y p e s

***********************************************************************************
*************/


/**********************************************************************************
**************
    L o c a l   V a r i a b l e s

***********************************************************************************
*************/


/**********************************************************************************
**************
    L o c a l   F u n c t i o n   P r o t o t y p e s

***********************************************************************************
*************/



/**********************************************************************************
********

***********************************************************************************
*******
 ********
********
 ********                    P u b l i c   F u n c t i o n s
********
 ********
********

***********************************************************************************
*******

***********************************************************************************
*******/




/**********************************************************************************
**********//*!

  @summary  Load a table of address/value pairs into the QF4A512 configuration
registers.
```

```
  @param[in]   Device       Device handle
  @param[in]   Table        Pointer to array of qf4a512_ConfigTableEntry structures
  @param[in]   SizeOfTable  Dimension of the 'Table' array

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - 'Table' is typically generated by the Quickfilter Pro PC design software.  Its
name is
    QFImageRegisterTable.  The number of entries in that table is held in
    QF_IMAGE_REGISTER_TABLE_DIMENSION, so a typical call to this function would be

    @code
        qf4a512_WriteImageRegisterTableToEeprom(
            SPI0_HANDLE,
            (qf4a512_ConfigTableEntry *)QFImageRegisterTable,
            QF_IMAGE_REGISTER_TABLE_DIMENSION);
    @endcode

  - Writing the table one byte at a time, as this function does, is not very
efficient, but
    the configuration table entries are address/data pairs, so optimzing the number
of writes
    would consume memory and code space.  Hopefully, this function won't be called
much, and
    when it is called, time will not be at a premium.

*//*****************************************************************************
*************/

inline
Result  qf4a512_LoadImageRegisterTable(
    const  Handle                     Device,
    const  qf4a512_ConfigTableEntry * Table,
    const  Count                      SizeOfTable )
{
    Count  TableIndex;


    //  Sanity check input parameters
    Assert( Device == SPI0_HANDLE);
    AssertNonNull( Table );
    Assert( SizeOfTable > 0 );


    //  Write each register specified in the table

    for( TableIndex = 0;
         TableIndex < SizeOfTable;
         TableIndex++ )
    {
        Assert( Table[ TableIndex ].Address  <  QF4A512_MAX_FILTER_COEFF_ADDRESS );

        qf4a512_WriteConfigByte(
            Device,
            Table[ TableIndex ].Address,
            Table[ TableIndex ].Value );
        //  Verify coefficient writes in a Debug build.  Control area writes (0x0
        //  to 0x100) aren't verified because read only bits that differ with
        //  written data cause false errors during readback.

#if  defined(Debug)
        if( TableIndex  >=  QF4A512_MIN_GH_FILTER_COEFF_ADDRESS    &&
            TableIndex  <=  QF4A512_MAX_FILTER_COEFF_ADDRESS )
        {
            Assert( Table[ TableIndex ].Value  ==
                    qf4a512_ReadConfigByte( Device, Table[ TableIndex ].Address ));
        }
#endif

    }


    return Success;
}




/*************************************************************************************
**********//*!

  @summary  Write 'Table' to the QF4A512 EEPROM.

  @param[in]   Device       Device handle
  @param[in]   Table        Pointer to array of qf4a512_ConfigTableEntry structures
  @param[in]   SizeOfTable  Dimension of the 'Table' array

  @return   'Success' if configuration successful, otherwise a negative value error
code.


  - 'Table' is typically generated by the Quickfilter Pro PC design software.  Its
name is
    QFImageRegisterTable.  The number of entries in that table is held in
    QF_IMAGE_REGISTER_TABLE_DIMENSION, so a typical call to this function would be

    @code
        qf4a512_WriteImageRegisterTableToEeprom(
            SPI0_HANDLE,
            (qf4a512_ConfigTableEntry *)QFImageRegisterTable,
            QF_IMAGE_REGISTER_TABLE_DIMENSION);
    @endcode

  - Writing the table one byte at a time, as this function does, is not very
efficient, but
    the configuration table entries are address/data pairs, so optimzing the number
of writes
    would consume memory and code space.  Hopefully, this function won't be called
much, and
    when it is called, time will not be at a premium.
```

```
*//****************************************************************************
*************/
/*
inline
Result  qf4a512_WriteImageRegisterTableToEeprom(
    const  Handle                     Device,
    const  qf4a512_ConfigTableEntry * Table,
    const  Count                      SizeOfTable )
{
    Count  TableIndex;


    //  Sanity check input parameters
    Assert( Device == SPI0_HANDLE);
    AssertNonNull( Table );
    Assert( SizeOfTable > 0 );


    //  Write each register specified in the table

    for( TableIndex = 0;
         TableIndex < SizeOfTable;
         TableIndex++ )
    {
        qf4a512_WriteEepromByte(
            Device,
            Table[ TableIndex ].Address,
            Table[ TableIndex ].Value);


        //  Verify writes in a Debug build.

        Assert( Table[ TableIndex ].Value  ==
                qf4a512_ReadEepromByte( Device, Table[ TableIndex ].Address ));
    }


    return Success;
}

*/


/****************************************************************************
**********//*!

  @summary  Software reset the specified QF4A512.

  @param[in]  Device       Device number.

  - After the device is reset, it may come up in either Configure or Run mode,
depending on
    the programming of the EEPROM in the device.  This function does not address
the
    possibility of coming up in Run mode.  It assumes the device will come up in
Confugre
    mode.  If needed, qf4a512_Init() shows how to detect the mode after reset.
```

```
*//*****************************************************************************
*************/
/*
inline
void  qf4a512_ResetDevice( const Handle Device )
{
    qf4a512_WriteConfigByte( Device,  QF4A512_FULL_SRST_ADDRESS,  BIT0 );
}



*/
#if  defined(NOT_IMPLEMENTED_YET)

/*******************************************************************************
**********//*!

  @summary  NOT IMPLEMENTED YET - Configure the QF4A512 PLL-based clock.

  @param[in]   clockConfig  PLL configuration structure

  @return   'Success' if configuration successful, otherwise a negative value error
code.

  - The master clock for the QF4A512 is produced by a crystal oscillator with a
nominal frequency
    of 20MHz.  Alternatively the device can be fed with an external clock signal
derived
    elsewhere.  The master clock is used as a reference for a phase-locked loop
(PLL), from which
    clocks are derived to drive the FIR filters, the ADC and the analog front end.
The master
    clock is also divided down to provide a clock to be used for transfers to the
on-chip EEPROM.

    @par
    @verbatim
                 . . . . . . . . . . . . . . . . . . . . .
                 .                          PLL          .
                 .               ____                    .
                 .  +-------+    /    \     +--------+     .      +--------+
      20 MHz --+--->|  /a   |--->|diff|---->|  VCO   |---+----+-->|   /x   |--->
SYS_CLK
               | .  +-------+    \____/     +--------+   | .  |   +--------+     up to
200MHz
               | . PLL_CTRL_0      ^                     | .  |  SYS_CLK_CTRL
               | .  (a=1-64)       |                     | .  |    (x=1-64)
               | .                 |                     | .  |
               | .                 |        +--------+   | .  |   +--------+
               | .                 +--------|   /b   |<--+ .  +-->|   /y   |--->
ADC_CLK
               | .                          +--------+     .      +--------+     up to
100MHz
               | .                          PLL_CTRL_1     .       ADC_CLK
               | .                          (b=1-64)       .       (y=2-16)
               | . . . . . . . . . . . . . . . . . . . . .
               |                                                  +--------+
               +------------------------------------------------->|   /z   |--->
EE_CLK
```

```
                                                                          +--------+     up to
1.25MHz
                                                                           STARTUP
                                                                           (z=1-32)
    @endverbatim

  - The PLL clock frequency is determined by the input clock frequency


   the pre-divider value (M) and the divider value (N): PLL_CLK = f0 * N / M The
default frequency
   for PLL_CLOCK is 200MHz. (f0 = 20MHz, M = 1, N= 10)  Operation of the PLL is
possible in two
   frequency ranges: 20-100MHz and 100-300MHz.

*//***************************************************************************
*************/

Result  QfConfigureClock(
    const ClockConfig  clockConfig)
{
    Assert(0);
}

#endif



/****************************************************************************
********

****************************************************************************
*******
 ********
********
 ********                    U t i l i t y   F u n c t i o n s
********
 ********
********

****************************************************************************
*******

****************************************************************************
*******/



/****************************************************************************
********

****************************************************************************
*******
 ********
********
 ********               L o c a l   ( S t a t i c )   F u n c t i o n s
********
 ********
********
```

```
*************************************************************************************
*******

*************************************************************************************
*******/
```

Common.h

```
/*****************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Common.h

  Common defines and macros that are generally applicable to C/C++ files.

  $Id: Common.h 121 2006-07-24 20:17:09Z jhopson $


*************************************************************************************
********

  @formats
      - C preprocessor directives
      - Doxygen (comment markup to produce HTML docs)

  @dependencies
      - Operating System:  none
      - Toolset:           none
      - CPU byte order:    none

  @notes
    -# Do not put anything non-portable in this file.

*///**********************************************************************************
**************


#if !defined(COMMON_H_INCLUDED)
#define      COMMON_H_INCLUDED


///  Type for pointers to functions with no input parameters or return types.

typedef  void(* FuncPtr )(void);



/////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Header Dependencies
//


///  Standardized, extensible type representing the result of a function call.
Positive
///  numbers and zero represent success, negative numbers represent failure.
```

```
typedef  enum  {

    Success                       =   0,           ///<  Zero and above is a
success

    InternalError                 =  -1,           ///<  Internal software error in
reporting module
    UnknownError                  =  -2,
    UnrecoverableError            =  -3,
    Timeout                       =  -4,

    DataOutOfRange                = -10,
    SizeOutOfRange                = -11,
    InvalidDataFormat             = -12,
    InvalidSyntax                 = -13,
    InvalidParameter              = -14,
    InvalidState                  = -15,

    BufferOverflow                = -20,
    BufferUnderflow               = -21,
    BufferEmpty                   = -22,

    InvalidAscii                  = -30,
    InvalidParity                 = -31,
    InvalidChecksum               = -32,
    InvalidFraming                = -33,
    InvalidProtocol               = -34,

    MediaFull                     = -40,
    MediaEmpty                    = -41,
    MediaReadError                = -42,
    MediaWriteError               = -43,
    MediaFormatError              = -44,
    InvalidMediaFormat            = -45,

    ResourceBusy                  = -50,
    ResourceReset                 = -51,
    ResourceNotFound              = -52,
    ResourceNotAvailable          = -53,
    ResourceExhausted             = -54,

    FeatureNotFound               = -60,
    FeatureNotInstalled           = -61,
    FeatureNotConfigured          = -62,
    FeatureNotSupported           = -63,
    FeatureNotImplemented         = -64,

    InvalidInterrupt              = -70,
    UnexpectedInterruptOccurred   = -71,
    UndefinedInterruptOccurred    = -72,

    InvalidUserAction             = -80,
    InvalidKeySelection           = -81

}  Result;
```

```
////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Common values for BOOL type.
//
//  Notes
//    1. Technically BOOL is true if > 0 and false if =0.  Tests for 'true' should
test
//       for non-zero, not 1.
//

#if  !defined(NO_COMMON_BOOL_MACROS)

#if !defined(false)
#define  false              0
#endif
#if !defined(true)
#define  true               1
#endif

#if !defined(False)
#define  False              0
#endif
#if !defined(True)
#define  True               1
#endif

#if !defined(No)
#define  No                 0
#endif
#if !defined(Yes)
#define  Yes                1
#endif

#define  Off                0
#define  On                 1

#define  Low                0
#define  High               1

#define  Clear              0
#define  Set                1

#define  Invalid            0
#define  Valid              1

#define  Inactive           0
#define  Active             1

#define  Disabled           0
#define  Enabled            1

#define  NotDone            0
#define  Done               1

#define  Short              0
#define  Long               1

#define  Input              0
```

```
#define  Output          1

#define  Inner           0
#define  Outer           1

#define  Minimize        0
#define  Maximize        1

#define  Unsuccessful    0
#define  Successful      1

#define  Incorrect       0
#define  Correct         1

#define  Bummer          0
#define  OkeeDokee       1

#endif



///////////////////////////////////////////////////////////////////////////////////
///////
//
//  Non-printable ASCII codes.
//

#if  !defined(NO_NON_PRINTABLE_ASCII_MACROS)

#define  nul         0          ///<  ^@   null character
#define  soh      0x01          ///<  ^A   start of heading
#define  stx      0x02          ///<  ^B   start of text
#define  etx      0x03          ///<  ^C   end of text
#define  eot      0x04          ///<  ^D   end of transmission
#define  enq      0x05          ///<  ^E   enquire
#define  ack      0x06          ///<  ^F   acknowledge
#define  bel      0x07          ///<  ^G   ring bell
#define  bs       0x08          ///<  ^H   backspace
#define  ht       0x09          ///<  ^I   horizontal tab
#define  lf       0x0A          ///<  ^J   line feed
#define  vt       0x0B          ///<  ^K   vertical tab
#define  ff       0x0C          ///<  ^L   form feed
#define  cr       0x0D          ///<  ^M   carriage return
#define  so       0x0E          ///<  ^N   shift out
#define  si       0x0F          ///<  ^O   shift in
#define  dle      0x10          ///<  ^P   data link escape
#define  dc1      0x11          ///<  ^Q   device control 1/x-on
#define  dc2      0x12          ///<  ^R   device control 2
#define  dc3      0x13          ///<  ^S   device control 3/x-off
#define  dc4      0x14          ///<  ^T   device control 4
#define  nak      0x15          ///<  ^U   negative acknowledgement
#define  syn      0x16          ///<  ^V   synchronous idle
#define  etb      0x17          ///<  ^W   end of transmission block
#define  can      0x18          ///<  ^X   cancel
#define  em       0x19          ///<  ^Y   end of medium
#define  sub      0x1A          ///<  ^Z   substitute
#define  esc      0x1B          ///<  ^[   escape
#define  fs       0x1C          ///<  ^\   file separator
#define  gs       0x1D          ///<  ^]   group separator
```

```
#define  rs       0x1E            ///<  ^^   record separator
#define  us       0x1F            ///<  ^_   unit separator
#define  del      0x7f            ///<       delete character

#endif




/////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Data extraction, combination and rotation macros.
//


///  Retun the lower 4 bits of a as a UInt8

#define  GetLoNibble(a)         (UInt8)((a)&0x0F)


///  Return second 4 bits a as a UInt8

#define  GetHiNibble(a)         (UInt8)((a)>>4)


///  Return lower byte of a as a UInt8

#define  GetLoByte(a)           (UInt8)((a)&0x00FF)


///  Return second byte of a as a UInt8

#define  GetHiByte(a)           (UInt8)((a)>>8)


///  Return lower word of a as a UInt16

#define  GetLoWord(a)           (UInt16)((a)&0xFFFF)


///  Return second word of a as a UInt16

#define  GetHiWord(a)           (UInt16)((a)>>16)


///  Set the lower byte of a UInt16, preserving the upper byte.
///  Existing contents in the lower word of a are lost.

#define  SetLoWord(a, b)        ((a)&0xff00)|(UInt16)(b))


///  Set the upper byte of a UInt16, preserving the lower byte
///  Existing contents in the upper word of a are lost.

#define  SetHiWord(a, b)        ((a)&0x00ff)|(UInt16)((b)<<8))
```

```
///  Rotate, not shift, a UInt8 to the left.  The MSB becomes the LSB.

#define  RotateByteLeft(a)      ((a)<<1)|(((a)&0x80)?1:0)


///  Rotate, not shift, a UInt8 to the right.  The LSB becomes the MSB.

#define  RotateByteRight(a)     ((a)>>1)|(((a)&0x01)?0x80:0)


///  Rotate, not shift, a UInt16 to the left.  The MSB becomes the LSB.

#define  RotateWordLeft(a)      ((a)<<1)|(((a)&0x8000)?1:0)


///  Rotate, not shift, a UInt16 to the right.  The LSB becomes the MSB.

#define  RotateWordRight(a)     ((a)>>1)|(((a)&1)?0x8000:0)


///  Create a UInt16 out of two UInt8 or INT8

#define  MakeUInt16(a,b)         (((UInt16)(a)<<8)|(UInt16)(b))


///  Create a UInt32 out of two UInt16 or INT16

#define  MakeUInt32(a,b)         (((UInt32)(a)<<16)|(UInt32)(b))


///  Add two pointers and create a void pointer

#define  AddPointers(a,b)        (void*)((Count)(a)+(Count)(b))


///  Subtract two pointers and create a void pointer

#define  SubtractPointers(a,b)   (void*)((Count)(a)-(Count)(b))


///  Swap upper and lower bytes (change endianness) in a 16-bit value

#define  SwapUInt16Bytes(a)     (((a)>>8) | ((a)<<8))


///  If the compiler doesn't define NULL, define it here

#if !defined(NULL)
#define  NULL  (void *)0
#endif



///////////////////////////////////////////////////////////////////////////////
///////
//
//  Hardware register manipulation macros
//
```

```
//  Notes
//    1.  These macros treat registers as volatile.
//    2.  It is sometimes easier to view bit registers in binary form.  The binary
//        representation macros (below) can be used as follows.
//        8-bit registers -
//            //                     7654 3210
//            SetByte( UartCtrl1,  _0001_0100 );
//
//         Word-wide registers -
//            //                                1111 11
//            //                                5432 1098   7654 3210
//            SetWordFromBytes( UartCtrl1,  _0001_0111, _1111_1111 );
//            SetWordFromBytes( UartCtrl2,  _0001_0100, _0101_1010 );
//            SetWordFromBytes( UartCtrl3,  _0101_0011, _0111_1001 );
//


///  Returns a byte (UInt8) from a volatile hardware register given a pointer to
that register.

#define  GetByte(ptr)          *((const volatile UInt8 *)(ptr))


///  Returns a word (UInt16) from a volatile hardware register given a pointer to
that register.

#define  GetWord(ptr)          *((const volatile UInt16*)(ptr))


///  Returns a word (UInt32) from a volatile hardware register given a pointer to
that register.

#define  GetLong(ptr)          *((const volatile UInt32*)(ptr))


///  Sets a byte (UInt8) volatile hardware register at the specified pointer
location.

#define  SetByte(ptr,val)      *((volatile UInt8 *)(ptr))=(val)


///  Sets a byte (UInt16) volatile hardware register at the specified pointer
location.

#define  SetWord(ptr,val)      *((volatile UInt16*)(ptr))=(val)


///  Sets a long (UInt32) volatile hardware register at the specified pointer
location.

#define  SetLong(ptr,val)      *((volatile UInt32*)(ptr))=(val)


///  Sets a word (UInt16) volatile hardware register from two bytes (UInt8) at the
specified pointer location.

#define  SetWordFromBytes(ptr,a,b)  *((volatile UInt16*)(ptr))=MakeUInt16(a,b)
```

```
////////////////////////////////////////////////////////////////////////////////
///////
//
//  Bit manipulation macros
//
//  Notes
//    1.  These macros treat all memory locations as volatile.
//    2.  Use IS_BIT_SET() in conditionals.  For instance -
//
//            if( IS_BIT_SET( UartCtrl1, BIT5 ) ) ...
//
//    3.  SET_BIT_x() and CLEAR_BIT_x() perform a NON-ATOMIC read-modify-write.  If
atomicity
//        is required, disable interrupts first.
//

#if  !defined(NO_COMMON_BIT_MACROS)

/*   Removed to eliminate conflict with same definitions in IAR header files -
#define  BIT0     0x01                  ///< Specifies first bit in the bit macros
#define  BIT1     0x02                  ///< Specifies second bit in the bit macros
#define  BIT2     0x04                  ///< Specifies third bit in the bit macros
#define  BIT3     0x08                  ///< Specifies fourth bit in the bit macros

#define  BIT4     0x10                  ///< Specifies fifth bit in the bit macros
#define  BIT5     0x20                  ///< Specifies sixth bit in the bit macros
#define  BIT6     0x40                  ///< Specifies seventh bit in the bit macros
#define  BIT7     0x80                  ///< Specifies eigth bit in the bit macros

#define  BIT8     0x0100                ///< Specifies ninth bit in the bit macros
#define  BIT9     0x0200                ///< Specifies tenth bit in the bit macros
*/
#define  BIT10    0x0400                ///< Specifies eleventh bit in the bit macros
#define  BIT11    0x0800                ///< Specifies twelth bit in the bit macros

#define  BIT12    0x1000                ///< Specifies thirteenth bit in the bit
macros
#define  BIT13    0x2000                ///< Specifies fourteenth bit in the bit
macros
#define  BIT14    0x4000                ///< Specifies fifteenth bit in the bit
macros
#define  BIT15    0x8000                ///< Specifies sixteenth bit in the bit
macros

#define  BIT16    0x00010000            ///< Specifies seventeenth bit in the bit
macros
#define  BIT17    0x00020000            ///< Specifies eightteenth bit in the bit
macros
#define  BIT18    0x00040000            ///< Specifies nineteenth bit in the bit
macros
#define  BIT19    0x00080000            ///< Specifies twentieth bit in the bit
macros

#define  BIT20    0x00100000            ///< Specifies twenty-first bit in the bit
macros
#define  BIT21    0x00200000            ///< Specifies twenty-second bit in the bit
macros
```

```
#define  BIT22    0x00400000          ///< Specifies twenty-third bit in the bit
macros
#define  BIT23    0x00800000          ///< Specifies twenty-fourth bit in the bit
macros

#define  BIT24    0x01000000          ///< Specifies twenty-fifth bit in the bit
macros
#define  BIT25    0x02000000          ///< Specifies twenty-sixth bit in the bit
macros
#define  BIT26    0x04000000          ///< Specifies twenty-seventh bit in the bit
macros
#define  BIT27    0x08000000          ///< Specifies twenty-eighth bit in the bit
macros

#define  BIT28    0x10000000          ///< Specifies twenty-ninth bit in the bit
macros
#define  BIT29    0x20000000          ///< Specifies thirtieth bit in the bit
macros
#define  BIT30    0x40000000          ///< Specifies thirty-first bit in the bit
macros
#define  BIT31    0x80000000          ///< Specifies thirty-second bit in the bit
macros


///  Returns true if bit is set, false otherwise.  Actually, true if any bits set
in b are set in a.

#define  IsBitSet(a,b)    ((a)&(b) ? true : false)


///  Returns true if bit is clear, false otherwise.  Actually, true if any bits
clear in b are set in a.

#define  IsBitClear(a,b)  ((a)&(b) ? false : true)


///  Sets the specified bit.  Actually, sets all the bits in a that are set in b .

#define  SetBit(a,b)      (a)|=(b)


///  Clears the specified bit.  Actually, clears all the bits in a that are set in
b .

#define  ClearBit(a,b)    (a)&= ~(b)


///  Toggles the specified bit.  Actually, toggles all the bits in a that are set
in b .

#define  ToggleBit(a,b)   (a)^=(b)



///  Applies the mask to value, then 'returns' true if any of those bits are set

#define  AreAnyMaskBitsSet(value, mask)    ((value)&(mask) ? true : false)
```

```
///  All bits that are set in mask are cleared in value

#define  ClearBitsUsingMask(value, mask)   (value)&= ~(mask)


///  All bits that are set in mask are set in value

#define  SetBitsUsingMask(value, mask)     (value)|=(mask)


///  All bits that are set in mask are toggled in value

#define  ToggleBitsUsingMask(value, mask)  (value)^=(mask)


#endif




////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Character and string macros.
//

#if  !defined(NO_COMMON_CHARACTER_MACROS)

#define  ASCII_MAX                  0x7F                ///<  Maximum ASCII VAlue
#define  WHITESPACE_CHARACTERS  " \t\r"                 ///<  Define string
containing the whitespace characters


///  Returns 0 if character is not ASCII, non-zero if ASCII

#define  IsAscii(chr)           ((chr)<= ASCII_MAX)


///  Returns 0 if character is not hexadecimal ASCII, otherwise non-zero

#define  IsHex(chr)             (((chr)>='0')&&((chr)<='9') ||\
                                 ((chr)>='a')&&((chr)<='f') ||\
                                 ((chr)>='A')&&((chr)<='F'))


///  Returns 0 if character is not ASCII whitespace, otherwise non-zero

#define  IsWhitespace(chr)      (((chr)==' ')||((chr)=='\t')||((chr)=='\n')) ?
true:false



///  Converts a character to its uppercase value only if it is a lowercase value.

#define  ToLower(chr)           (chr)+(((chr)>=0x61)&&((chr)<=0x7A))?0x20:0


///  Converts a character to its lowercase value only if it is an uppercase value.
```

```
#define  ToUpper(chr)           (chr)-(((chr)>=0x61)&&((chr)<=0x7A))?0x20:0


#endif




///////////////////////////////////////////////////////////////////////////////////
///////
//
//  Array macros
//
//
//  Notes
//    1.  Note that 'array' must be an array name and not an expression.
//

#if  !defined(NO_COMMON_ARRAY_MACROS)


///  Get size, in bytes (INT8), of an array

#define  GetArraySize(array)           (sizeof(array))


///  Get the number of elements in an array

#define  GetArrayDimension(array)      (sizeof(array)/sizeof(array[0]))


///  Get a pointer to the end of an array

#define  GetEndOfArray(array)          ((void *)((Count)&array + sizeof(array)))


///  Get the index number of an element given a pointer to the array and a pointer
to the item.

#define  GetIndexFromPtr(array,pitem)  (((pitem)-&array)/sizeof(array[0]))


#endif




///////////////////////////////////////////////////////////////////////////////////
///////
//
//  Structure macros.
//
//  Notes
//    1.  Note that 'str' and 'element' must be names and not expressions.
//

#if  !defined(NO_COMMON_STRUCTURE_MACROS)
```

```
///  Get the offset of a structure element, measured from the beginning of the
structure

#define  GetElementOffset(str,element)   ((Count) (&((str *)0)->element))


///  Get the pointer to a structure element, given the beginning of the structure
and the element name

#define  GetElementPtr(str,element)      AddPtr(&str,
GetElementOffset(str,element))


#endif




////////////////////////////////////////////////////////////////////////////////
///////
//
//  Assertion macros.
//

#if  !defined(NO_COMMON_ASSERTION_MACROS)

#if  !defined(DISABLE_COMMON_ASSERTION_MACROS)

///  Assert wrapper

#define  Assert(a)               assert(a)


///  Throw an assertion if a is non-NULL.

#define  AssertNull(a)           assert((a) == NULL)


///  Throw an assertion if a is NULL.

#define  AssertNonNull(a)        assert((a) != NULL)


///  Throw an assertion if a is below lo or abouve hi.

#define  AssertBounds(a,lo,hi)   assert(((a)<=hi) && ((a)>=lo))



///  'a' is a function call.  This asserts that the function returns a
///  non-negative value.  Note that this macro behaves differently from
///  other assertion macros because the function still exists, without
///  the return value check, in non-Debug builds.

#if  defined(Debug)
#    define  Validate(a)    Assert( (a) >= 0 )
#else
#    define  Validate(a)    a
```

```
#endif


#else

#    define  Assert(a)                ((void) 0)
#    define  AssertNull(a)            ((void) 0)
#    define  AssertNonNull(a)         ((void) 0)
#    define  AssertBounds(a,lo,hi)    ((void) 0)
#    define  Validate(a)             a

#endif

#endif




////////////////////////////////////////////////////////////////////////////////
///////
//
//  Miscellaneous macros.
//

#if  !defined(NO_COMMON_MISCELLANEOUS_MACROS)


///  Returns non-zero if val is even

#define  IsEven(val)                !((val)&0x01)


///  Returns non-zero if val is odd

#define  IsOdd(val)                 ((val)&0x01)


///  Returns the higher of a and b

#define  GetHigherOf(a,b)           ((a)>(b))?(a):(b)


///  Returns the lower of a and b

#define  GetLowerOf(a,b)            ((a)<(b))?(a):(b)


///  Limits a's range to within lo and hi

#define  GetWithinRange(a,lo,hi) ((a)>(hi)?(hi):((a)<(lo)?(lo):(a)))


///  Wait for the expression to become true.  Warning - this does not block the
task or
///  timeout.  Be sure expression will eventually become true.

#define  WaitFor(exp)               while(!(exp))
```

```
///  Create an infinite loop

#define  InfiniteLoop()            while(1)


///  Dummy value used to emphasize that the value doesn't matter

#define  DontCare      0x5a



#endif




/////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Time macros.
//

#define  MICROSECONDS_PER_MILLISECOND  1000
#define  MICROSECONDS_PER_SECOND       1000000
#define  MICROSECONDS_PER_MINUTE       60000000

#define  MILLISECONDS_PER_SECOND    1000
#define  MILLISECONDS_PER_MINUTE    60000
#define  MILLISECONDS_PER_HOUR      3600000

#define  SECONDS_PER_MINUTE         60
#define  SECONDS_PER_HOUR           3600
#define  SECONDS_PER_DAY            86400

#define  SECONDS_PER_NON_LEAP_YEAR  31536000
#define  SECONDS_PER_LEAP_YEAR      31622400


#define  HOURS_PER_DAY           24
#define  HOURS_PER_TWO_DAYS      48
#define  HOURS_PER_THREE_DAYS    72
#define  HOURS_PER_FOUR_DAYS     96
#define  HOURS_PER_FIVE_DAYS     120
#define  HOURS_PER_SIX_DAYS      144
#define  HOURS_PER_SEVEN_DAYS    168

#define  HOURS_PER_WEEK          168

#define  HOURS_PER_28_DAY_MONTH  672
#define  HOURS_PER_29_DAY_MONTH  696
#define  HOURS_PER_30_DAY_MONTH  720
#define  HOURS_PER_31_DAY_MONTH  744

#define  HOURS_PER_NON_LEAP_YEAR 8760
#define  HOURS_PER_LEAP_YEAR     8784


#define  DAYS_PER_WEEK             7
```

```
#define  DAY_OF_WEEK_MONDAY       0
#define  DAY_OF_WEEK_TUESDAY      1
#define  DAY_OF_WEEK_WEDNESDAY    2
#define  DAY_OF_WEEK_THURSDAY     3
#define  DAY_OF_WEEK_FRIDAY       4
#define  DAY_OF_WEEK_SATURDAY     5
#define  DAY_OF_WEEK_SUNDAY       6

#define  DAYS_IN_JANUARY           31
#define  DAYS_IN_NON_LEAP_FEBRUARY 28
#define  DAYS_IN_LEAP_FEBRUARY     29
#define  DAYS_IN_MARCH             31
#define  DAYS_IN_APRIL             30
#define  DAYS_IN_MAY               31
#define  DAYS_IN_JUNE              30
#define  DAYS_IN_JULY              31
#define  DAYS_IN_AUGUST            31
#define  DAYS_IN_SEPTEMBER         30
#define  DAYS_IN_OCTOBER           31
#define  DAYS_IN_NOVEMBER          30
#define  DAYS_IN_DECEMBER          31

#define  DAYS_PER_NON_LEAP_YEAR  365
#define  DAYS_PER_LEAP_YEAR      366


#define  MONTHS_PER_YEAR           12




//////////////////////////////////////////////////////////////////////////////////
///////
//
//  Spell out long numbers to clarify their meaning and avoid typos.

#if  !defined(NO_COMMON_LONG_NUMBER_DEFINES)


#define  TEN_THOUSAND                 10000
#define  HUNDRED_THOUSAND            100000
#define  ONE_MILLION                1000000
#define  TEN_MILLION               10000000
#define  HUNDRED_MILLION          100000000


#define  _2TO_THE_4TH                    16
#define  _2TO_THE_5TH                    32
#define  _2TO_THE_6TH                    64
#define  _2TO_THE_7TH                   128
#define  _2TO_THE_8TH                   256
#define  _2TO_THE_9TH                   512
#define  _2TO_THE_10TH                 1024
#define  _2TO_THE_11TH                 2048
#define  _2TO_THE_12TH                 4096
#define  _2TO_THE_13TH                 8192
#define  _2TO_THE_14TH                16384
#define  _2TO_THE_15TH                32768
#define  _2TO_THE_16TH                65536
```

```
#define   _2TO_THE_17TH           131072
#define   _2TO_THE_18TH           262144
#define   _2TO_THE_19TH           524288
#define   _2TO_THE_20TH          1048576
#define   _2TO_THE_21ST          2097152
#define   _2TO_THE_22ND          4194304
#define   _2TO_THE_23RD          8388608
#define   _2TO_THE_24TH         16777216
#define   _2TO_THE_25TH         33554432
#define   _2TO_THE_26TH         67108864
#define   _2TO_THE_27TH        134217728
#define   _2TO_THE_28TH        268435456
#define   _2TO_THE_29TH        536870912
#define   _2TO_THE_30TH       1073741824
#define   _2TO_THE_31TH       2147483648
#define   _2TO_THE_32ND       4294967296


#define   ONE_K                             _2TO_THE_10TH
#define   TWO_K                             _2TO_THE_11TH
#define   FOUR_K                            _2TO_THE_12TH
#define   EIGHT_K                           _2TO_THE_13TH
#define   SIXTEEN_K                         _2TO_THE_14TH
#define   THIRTY_TWO_K                      _2TO_THE_15TH
#define   SIXTY_FOUR_K                      _2TO_THE_16TH
#define   HUNDRED_TWENTY_EIGHT_K            _2TO_THE_17TH
#define   TWO_HUNDRED_FIFTY_SIX_K           _2TO_THE_18TH
#define   FIVE_HUNDRED_TWELVE_K             _2TO_THE_19TH


#define   HALF_MEGABYTE                     _2TO_THE_19TH
#define   ONE_MEGABYTE                      _2TO_THE_20TH
#define   TWO_MEGABYTES                     _2TO_THE_21ST
#define   FOUR_MEGABYTES                    _2TO_THE_22ND
#define   EIGHT_MEGABYTES                   _2TO_THE_23RD
#define   SIXTEEN_MEGABYTES                 _2TO_THE_24TH
#define   THIRTY_TWO_MEGABYTES              _2TO_THE_25TH
#define   SIXTY_FOUR_MEGABYTES              _2TO_THE_26TH
#define   HUNDRED_TWENTY_EIGHT_MEGABYTES    _2TO_THE_27TH
#define   TWO_HUNDRED_FIFTY_SIX_MEGABYTES   _2TO_THE_28TH

#define   HALF_GIGABYTE                     _2TO_THE_29TH
#define   ONE_GIGABYTES                     _2TO_THE_30TH
#define   FOUR_GIGABYTES                    _2TO_THE_32ND


#endif




////////////////////////////////////////////////////////////////////////////////////
///////
//
//  Binary representations of byte-wide numbers.
//

#if  !defined(NO_COMMON_BINARY_NUMBER_VALUES)
```

```
#define   _0000_0000    0x00                      //  0x00 to 0x1F
#define   _0000_0001    0x01
#define   _0000_0010    0x02
#define   _0000_0011    0x03
#define   _0000_0100    0x04
#define   _0000_0101    0x05
#define   _0000_0110    0x06
#define   _0000_0111    0x07
#define   _0000_1000    0x08
#define   _0000_1001    0x09
#define   _0000_1010    0x0A
#define   _0000_1011    0x0B
#define   _0000_1100    0x0C
#define   _0000_1101    0x0D
#define   _0000_1110    0x0E
#define   _0000_1111    0x0F

#define   _0001_0000    0x10                      //  0x10 to 0x1F
#define   _0001_0001    0x11
#define   _0001_0010    0x12
#define   _0001_0011    0x13
#define   _0001_0100    0x14
#define   _0001_0101    0x15
#define   _0001_0110    0x16
#define   _0001_0111    0x17
#define   _0001_1000    0x18
#define   _0001_1001    0x19
#define   _0001_1010    0x1A
#define   _0001_1011    0x1B
#define   _0001_1100    0x1C
#define   _0001_1101    0x1D
#define   _0001_1110    0x1E
#define   _0001_1111    0x1F

#define   _0010_0000    0x20                      //  0x20 to 0x2F
#define   _0010_0001    0x21
#define   _0010_0010    0x22
#define   _0010_0011    0x23
#define   _0010_0100    0x24
#define   _0010_0101    0x25
#define   _0010_0110    0x26
#define   _0010_0111    0x27
#define   _0010_1000    0x28
#define   _0010_1001    0x29
#define   _0010_1010    0x2A
#define   _0010_1011    0x2B
#define   _0010_1100    0x2C
#define   _0010_1101    0x2D
#define   _0010_1110    0x2E
#define   _0010_1111    0x2F

#define   _0011_0000    0x30                      //  0x30 to 0x3F
#define   _0011_0001    0x31
#define   _0011_0010    0x32
#define   _0011_0011    0x33
#define   _0011_0100    0x34
#define   _0011_0101    0x35
#define   _0011_0110    0x36
#define   _0011_0111    0x37
```

```
#define  _0011_1000    0x38
#define  _0011_1001    0x39
#define  _0011_1010    0x3A
#define  _0011_1011    0x3B
#define  _0011_1100    0x3C
#define  _0011_1101    0x3D
#define  _0011_1110    0x3E
#define  _0011_1111    0x3F

#define  _0100_0000    0x40                      //  0x40 to 0x4F
#define  _0100_0001    0x41
#define  _0100_0010    0x42
#define  _0100_0011    0x43
#define  _0100_0100    0x44
#define  _0100_0101    0x45
#define  _0100_0110    0x46
#define  _0100_0111    0x47
#define  _0100_1000    0x48
#define  _0100_1001    0x49
#define  _0100_1010    0x4A
#define  _0100_1011    0x4B
#define  _0100_1100    0x4C
#define  _0100_1101    0x4D
#define  _0100_1110    0x4E
#define  _0100_1111    0x4F

#define  _0101_0000    0x50                      //  0x50 to 0x5F
#define  _0101_0001    0x51
#define  _0101_0010    0x52
#define  _0101_0011    0x53
#define  _0101_0100    0x54
#define  _0101_0101    0x55
#define  _0101_0110    0x56
#define  _0101_0111    0x57
#define  _0101_1000    0x58
#define  _0101_1001    0x59
#define  _0101_1010    0x5A
#define  _0101_1011    0x5B
#define  _0101_1100    0x5C
#define  _0101_1101    0x5D
#define  _0101_1110    0x5E
#define  _0101_1111    0x5F

#define  _0110_0000    0x60                      //  0x60 to 0x6F
#define  _0110_0001    0x61
#define  _0110_0010    0x62
#define  _0110_0011    0x63
#define  _0110_0100    0x64
#define  _0110_0101    0x65
#define  _0110_0110    0x66
#define  _0110_0111    0x67
#define  _0110_1000    0x68
#define  _0110_1001    0x69
#define  _0110_1010    0x6A
#define  _0110_1011    0x6B
#define  _0110_1100    0x6C
#define  _0110_1101    0x6D
#define  _0110_1110    0x6E
#define  _0110_1111    0x6F
```

```
#define   _0111_0000    0x70                        //  0x70 to 0x7F
#define   _0111_0001    0x71
#define   _0111_0010    0x72
#define   _0111_0011    0x73
#define   _0111_0100    0x74
#define   _0111_0101    0x75
#define   _0111_0110    0x76
#define   _0111_0111    0x77
#define   _0111_1000    0x78
#define   _0111_1001    0x79
#define   _0111_1010    0x7A
#define   _0111_1011    0x7B
#define   _0111_1100    0x7C
#define   _0111_1101    0x7D
#define   _0111_1110    0x7E
#define   _0111_1111    0x7F

#define   _1000_0000    0x80                        //  0x80 to 0x8F
#define   _1000_0001    0x81
#define   _1000_0010    0x82
#define   _1000_0011    0x83
#define   _1000_0100    0x84
#define   _1000_0101    0x85
#define   _1000_0110    0x86
#define   _1000_0111    0x87
#define   _1000_1000    0x88
#define   _1000_1001    0x89
#define   _1000_1010    0x8A
#define   _1000_1011    0x8B
#define   _1000_1100    0x8C
#define   _1000_1101    0x8D
#define   _1000_1110    0x8E
#define   _1000_1111    0x8F

#define   _1001_0000    0x90                        //  0x90 to 0x9F
#define   _1001_0001    0x91
#define   _1001_0010    0x92
#define   _1001_0011    0x93
#define   _1001_0100    0x94
#define   _1001_0101    0x95
#define   _1001_0110    0x96
#define   _1001_0111    0x97
#define   _1001_1000    0x98
#define   _1001_1001    0x99
#define   _1001_1010    0x9A
#define   _1001_1011    0x9B
#define   _1001_1100    0x9C
#define   _1001_1101    0x9D
#define   _1001_1110    0x9E
#define   _1001_1111    0x9F

#define   _1010_0000    0xA0                        //  0xA0 to 0xAF
#define   _1010_0001    0xA1
#define   _1010_0010    0xA2
#define   _1010_0011    0xA3
#define   _1010_0100    0xA4
#define   _1010_0101    0xA5
#define   _1010_0110    0xA6
```

```
#define  _1010_0111    0xA7
#define  _1010_1000    0xA8
#define  _1010_1001    0xA9
#define  _1010_1010    0xAA
#define  _1010_1011    0xAB
#define  _1010_1100    0xAC
#define  _1010_1101    0xAD
#define  _1010_1110    0xAE
#define  _1010_1111    0xAF

#define  _1011_0000    0xB0                          //  0xB0 to 0xBF
#define  _1011_0001    0xB1
#define  _1011_0010    0xB2
#define  _1011_0011    0xB3
#define  _1011_0100    0xB4
#define  _1011_0101    0xB5
#define  _1011_0110    0xB6
#define  _1011_0111    0xB7
#define  _1011_1000    0xB8
#define  _1011_1001    0xB9
#define  _1011_1010    0xBA
#define  _1011_1011    0xBB
#define  _1011_1100    0xBC
#define  _1011_1101    0xBD
#define  _1011_1110    0xBE
#define  _1011_1111    0xBF

#define  _1100_0000    0xC0                          //  0xC0 to 0xCF
#define  _1100_0001    0xC1
#define  _1100_0010    0xC2
#define  _1100_0011    0xC3
#define  _1100_0100    0xC4
#define  _1100_0101    0xC5
#define  _1100_0110    0xC6
#define  _1100_0111    0xC7
#define  _1100_1000    0xC8
#define  _1100_1001    0xC9
#define  _1100_1010    0xCA
#define  _1100_1011    0xCB
#define  _1100_1100    0xCC
#define  _1100_1101    0xCD
#define  _1100_1110    0xCE
#define  _1100_1111    0xCF

#define  _1101_0000    0xD0                          //  0xD0 to 0xDF
#define  _1101_0001    0xD1
#define  _1101_0010    0xD2
#define  _1101_0011    0xD3
#define  _1101_0100    0xD4
#define  _1101_0101    0xD5
#define  _1101_0110    0xD6
#define  _1101_0111    0xD7
#define  _1101_1000    0xD8
#define  _1101_1001    0xD9
#define  _1101_1010    0xDA
#define  _1101_1011    0xDB
#define  _1101_1100    0xDC
#define  _1101_1101    0xDD
#define  _1101_1110    0xDE
```

```
#define  _1101_1111    0xDF

#define  _1110_0000    0xE0                        //  0xE0 to 0xEF
#define  _1110_0001    0xE1
#define  _1110_0010    0xE2
#define  _1110_0011    0xE3
#define  _1110_0100    0xE4
#define  _1110_0101    0xE5
#define  _1110_0110    0xE6
#define  _1110_0111    0xE7
#define  _1110_1000    0xE8
#define  _1110_1001    0xE9
#define  _1110_1010    0xEA
#define  _1110_1011    0xEB
#define  _1110_1100    0xEC
#define  _1110_1101    0xED
#define  _1110_1110    0xEE
#define  _1110_1111    0xEF

#define  _1111_0000    0xF0                        //  0xF0 to 0xFF
#define  _1111_0001    0xF1
#define  _1111_0010    0xF2
#define  _1111_0011    0xF3
#define  _1111_0100    0xF4
#define  _1111_0101    0xF5
#define  _1111_0110    0xF6
#define  _1111_0111    0xF7
#define  _1111_1000    0xF8
#define  _1111_1001    0xF9
#define  _1111_1010    0xFA
#define  _1111_1011    0xFB
#define  _1111_1100    0xFC
#define  _1111_1101    0xFD
#define  _1111_1110    0xFE
#define  _1111_1111    0xFF

#endif


#endif   //  COMMON_DEFINES_INCLUDED
```

MSP430.h

```
/*****************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Msp430.h

  MSP-430-related definitions.

  $Id: Msp430.h 121 2006-07-24 20:17:09Z jhopson $

*///*****************************************************************************
**************


#if !defined(MSP430_44X_H_INCLUDED)
#define      MSP430_44X_H_INCLUDED
```

```
#include  <msp430x44x.h>
#include  "Msp430-Types.h"


#define  CPU_FAMILY              MSP430         //  CPU core compatibility (such as
                                                //  68HC11, x86_REAL).
#define  CPU_VAIRANT             449            //  Subtype of CPU
#define  CPU_FULL_PART_NUMBER    MSP430F449     //  Full CPU model number (w/o
package info)


#define  CPU_REGISTER_WIDTH      16             //  Data bus width in bits.
#define  CPU_DATA_BUS_WIDTH      16             //  Data bus width in bits.
#define  CPU_ADDRESS_BUS_WIDTH   16             //  Address bus width in bits.


#define  CPU_BYTE_ORDER          LITTLE_ENDIAN  //  Byte order of addressed data
(op-
                                                //  tions are
BIG/LITTLE/SWITCHABLE).

#define  CPU_ALIGNMENT           NO_ALIGN       //  Minimum storage boundary for
data
                                                //  (options are NO_ALIGN,
BYTE_ALIGN
                                                //  WORD_ALIGN, LWORD_ALIGN).

#define  CPU_VOLTAGE             3.3            //  Voltage of device.

#define  CPU_INPUT_CLOCK         8000000        //  Frequency of CPU clock in Hz


#endif
```

MSP430-SPI.h

```
/*****************************  (C) Quickfilter Technologies, Inc.
************************//*!

  @file  Msp430-SPI.h

  Abstraction of MSP-430 SPI port for use with Quickfilter device drivers.

  $Id: Msp430-SPI.h 121 2006-07-24 20:17:09Z jhopson $

*///**********************************************************************
**************


#if !defined(MSP_430_SPI_H_INCLUDED)                //        Inclusion control
#    define  MSP_430_SPI_H_INCLUDED


#include "Project.h"
```

```
///  Device handle for USART0.
#define  SPI0_HANDLE   0

///  Device handle for USART1.
#define  SPI1_HANDLE   1


///  Maximum allowable device number.  Device 0 is USART0 and device 1 is USART1.
#define  QF_SPI_MAX_DEVICE_NUMBER   0


///  Byte capacity of the local stream read buffers
#define  QF_SPI_SIZE_OF_SPI_READ_BUFFER   128


///  Standard type for function from receive ISR when a byte is received
typedef  Bool (* ReceiveCallback)(Byte);


///  Configuration options for the SPI port.
typedef enum  {
    ///  Reads are performed as requested.  No buffering
    ///  occurs between @ref QfSpi_Read read calls.
    SetUnbufferedReadMode,

    ///  SPI data is continuously buffered.  User must call @ref QfSpi_Read
    ///  at the rate at which data is filling up buffers.
    SetBufferedReadMode

}  QfSpi_ConfigRequest;



#ifdef __cplusplus
   extern "C"                              //  C++ source requires this linkage
specification
{
#endif



void  QfSpi_Init( void );

void  QfSpi_DeInit( void );

void  QfSpi_Write(
    const Handle  Device,
    const Byte   * Buffer,
    const Count   Length);

Result  QfSpi_Read(
    const Handle  Device,
          Byte   * Buffer,
    const Count   Length);

Result  QfSpi_Configure(
    const Handle  Device,
    const QfSpi_ConfigRequest  Request,
    void  * Buffer,
    Count * Length);

Byte  QfSpi_ReadByte(
    const Handle  Device );

void  QfSpi_WriteByte(
    const  Handle  Device,
    const  Byte    Value);

void  QfSpi_WriteUInt16(
    const  Handle  Device,
           UInt16  Value);

void QfSpi_ResumeReceive(
    const  Handle  Device );


#ifdef __cplusplus
}                                       //  End of C++ linkage specification
#endif


#endif
```

Platform.h

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Platform.h

  Abstracts a variety of operations that are specific to the way this platform
circuit
  is designed.

  $Id: Platform.h 121 2006-07-24 20:17:09Z jhopson $

*///*********************************************************************************
**************


#if !defined(PLATFORM_H_INCLUDED)                  //   Inclusion control
#    define  PLATFORM_H_INCLUDED


#include  "Project.h"
#include  "Msp430-SPI.h"


#define  DRDY_BIT   BIT0


//#if  defined(Debug)

//  Three instrumentation bits are used during debug.
                                          //   name   pin  (on '449)
                                          //   ----   ---
//#define  ActivityLED            BIT1      //   P5.1   12
```

```
//#define  TimerA0IsrTestPoint    BIT4      //   P5.4   55
//#define  Spi0DrdyIsrTestPoint   BIT5      //   P5.5   57
//#define  Spi0ReadIsrTestPoint   BIT6      //   P5.6   58
//#define  Spi0WriteIsrTestPoint  BIT7      //   P5.7   59


//#define  InitTestPointPort()   { P5DIR |=  BIT1 | BIT4 | BIT5 | BIT6 | BIT7;  }

//#define  SetTestPoint( a )     SetBit( P5OUT, a )
//#define  ClearTestPoint( a )   ClearBit( P5OUT, a )
//#define  ToggleTestPoint( a )  ToggleBit( P5OUT, a )
//#define  IsTestPointSet( a )   IsBitSet( P5OUT, a )


//#else


//#define  InitTestPointPort()   ((void)0)
//#define  SetTestPoint( a )     ((void)0)
//#define  ClearTestPoint( a )   ((void)0)
//#define  ToggleTestPoint( a )  ((void)0)
//#define  IsTestPointSet( a )   ((void)0)

//#endif



#ifdef __cplusplus
   extern "C" {                           //  C++ source requires this linkage
specification
#endif



void  QfPlat_Init( void );
void  QfPlat_ActivateQF4A512ChipSelect( void );
void  QfPlat_DeactivateQF4A512ChipSelect( void );
void  QfPlat_SetSpi0DrdyDirection( Bool Direction );
void  QfPlat_SetSpi0DrdyState( Bool State );
void  QfPlat_DelayMs( const UInt16  Milliseconds );
Bool  QfPlat_IsDrdyPinActive( void );
void  QfPlat_ConfigureDrdyInterrupt( FuncPtr Handler );
void  QfPlat_EnableDrdyInterrupts( void );
void  QfPlat_DisableDrdyInterrupts( void );
void  QfPlat_ToggleActivityLED( void );

#if  defined(Debug)
void QfAssertionFailed( const char *  Expression,   const char * FileName,   const
int LineNum );
#endif



#ifdef __cplusplus
   }                                      //  End of C++ linkage specification
#endif


#endif
```

Project.h

```
/*****************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Project.h

  Project-specific defines and types.

  $Id: Project.h 122 2006-07-24 20:25:01Z jhopson $


*************************************************************************************
******

  @formats
      - C declarations (header)
      - Doxygen (comment markup to produce HTML docs)

*///**********************************************************************************
*************


#if !defined(PROJECT_H_INCLUDED)                  //   Inclusion control
#    define  PROJECT_H_INCLUDED


//  Suppress  "Warning[Pa039]: use of address of unaligned structure member..."
occurs
//  because the structures in Qf4a512-access.h are (by necessity) misaligned.

#pragma  diag_suppress = Pa039


///  The inline keyword isn't supported in the IAR C compiler (it is in the C++
compiler)
#define  inline


#define  QF4A512_NUM_CHANNELS_ENABLED  3


#include  "Common.h"
#include  "Msp430.h"
#include  <stdlib.h>
#include  <assert.h>


#endif   //  PROJECT_H_INCLUDED
```

Qf4a512-access.h

```
/*****************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Qf4a512-access.h
  Raw data transfer functions and device mode control of QF4A512 (Small model).

  $Id: Qf4a512-access.h 120 2006-07-19 16:11:20Z jhopson $

*************************************************************************************
********

  @notes
    -#  The structures in this file must be packed (no space between elements).
The 'pack(1)'
        pragma is for the IAR compiler, but it shoul be benign with other
compilers.

*///*********************************************************************************
**************


#if !defined(QF4A512_ACCESS_H_INCLUDED)                // Inclusion control
#    define  QF4A512_ACCESS_H_INCLUDED



#include  "Project.h"


///  Number of channels in a QF4A512 device
#define  QF4A512_NUM_CHANNELS    3


///  Maximum number of FIR coefficients in each channel
#define  QF4A512_NUM_FIR_COEFFICIENTS_PER_CHANNEL    256


///  Number of bytes in each FIR filter coefficient.
#define  QF4A512_NUM_BYTES_PER_FIR_COEFFICIENT     3


///  Maximum of G & H coefficients in each channel
#define  QF4A512_NUM_GH_COEFFICIENTS_PER_CHANNEL    64



/************************************************************************************
**********//*!

  @summary  Channel and FIR configuration for one individual channel.

  - Below is the memory map for the channel configuration area.
       @par
       @verbatim
         +-----------------------------------------+ 00EA
         |             E2h   (reserved)    (8)    |    <-- note less reserved
space
         |  Channel    E1h   Maintenance   (1)    |        on Channel 4
         |     4       C2h   Configuration (31)   |
         |             C0h   Run & Status  (2)    |
         +-----------------------------------------+ 00C0
         |             B2h   (reserved)    (14)   |
         |  Channel    B1h   Maintenance   (1)    |
         |     3       92h   Configuration (31)   |
         |             90h   Run & Status  (2)    |
         +----------------------------------------+ 0090
         |             82h   (reserved)    (14)   |
         |  Channel    81h   Maintenance   (1)    |
         |     2       62h   Configuration (31)   |
         |             60h   Run & Status  (2)    |
         +----------------------------------------+ 0060
         |             52h   (reserved)    (14)   |
         |  Channel    51h   Maintenance   (1)    |
         |     1       32h   Configuration (31)   |
         |             30h   Run & Status  (2)    |
         +----------------------------------------+ 0030
       @endverbatim


  - This structure must be packed because some two-byte registers are at odd
addresses.

*//*********************************************************************************
*************/

#pragma pack(1)

typedef struct {                //   Offset          Description
                                //   ------          -----------
    Byte    CH1_PGA;            ///<  Offset +0h.     Control Register. Enable FIR
operation, set PGA gain.
    Byte    CH1_STAT;           ///<  Offset +1h.     Channel Status

    Byte    CH1_CFG;            ///<  Offset +2h.     Channel configuration
    UInt16  AREC_1_GAIN;        ///<  Offset +3/4h.   AREC gain control
    UInt16  CHPC_1_DIV;         ///<  Offset +5/6h.   Chopper period setting
    UInt16  CIC_1_R;            ///<  Offset +7/8h.   CIC decimation, R value
    Byte    CIC_1_R_H;          ///<  Offset +9h.     CIC decimation, R value
    Byte    CIC_1_SHIFT;        ///<  Offset +Ah.     CIC shift

    Byte    FIR_0_0_CTRL;       ///<  Offset +Bh.     FIR Control, filter 0
    Byte    FIR_0_0_NMIN_F1;    ///<  Offset +Ch.     Minimum storage address for f1
    Byte    FIR_0_0_NMAX_F1;    ///<  Offset +Dh.     Maximum storage address for f1
    Byte    FIR_0_0_CMIN_F1;    ///<  Offset +Eh.     Minimum coefficient storage
address for f1
    Byte    FIR_0_0_CMAX_F1;    ///<  Offset +Fh.     Maximum coefficient storage
address for f1
    Byte    FIR_0_0_NMIN_F2;    ///<  Offset +10h.    Minimum storage address for f2
    Byte    FIR_0_0_NMAX_F2;    ///<  Offset +11h.    Maximum storage address for f2
    Byte    FIR_0_0_CMIN_F2;    ///<  Offset +12h.    Minimum coefficient storage
address for f2
    Byte    FIR_0_0_CMAX_F2;    ///<  Offset +13h.    Maximum coefficient storage
address for f2

    Byte    FIR_0_1_CTRL;       ///<  Offset +14h.    FIR Control, filter 1
    UInt16  FIR_0_1_NMIN_F1;    ///<  Offset +15/16h. Minimum storage address for f1
    UInt16  FIR_0_1_NMAX_F1;    ///<  Offset +17/18h. Maximum storage address for f1
    Byte    FIR_0_1_CMIN_F1;    ///<  Offset +19h.    Minimum coefficient storage
address for f1
    Byte    FIR_0_1_CMAX_F1;    ///<  Offset +1Ah.    Maximum coefficient storage
address for f1
```

```
    UInt16  FIR_0_1_NMIN_F2;   ///<  Offset +1B/1Ch. Minimum storage address for f2
    UInt16  FIR_0_1_NMAX_F2;   ///<  Offset +1D/1Eh. Maximum storage address for f2
    Byte    FIR_0_1_CMIN_F2;   ///<  Offset +1Fh.    Minimum coefficient storage
address for f2
    Byte    FIR_0_1_CMAX_F2;   ///<  Offset +20h.    Maximum coefficient storage
address for f2

    Byte    CH1_SRST;          ///<  Offset +21h.  Channel soft reset

}  qf4a512_ChannelConfigRegs;

#pragma pack()




/************************************************************************************
**********//*!

  @summary  Represents all registers in the QF4A512 device.

  - Below is the memory map for the entire device.

       @par
       @verbatim
        +------------------------------------------+ 00F0
        |          Die Rev & Calibration           |
        +------------------------------------------+ 00EA
        |             E2h   (reserved)    (8)      |    <-- note less reserved space
        |  Channel    E1h   Maintenance   (1)      |        on Channel 4
        |     4       C2h   Configuration (31)     |
        |             C0h   Run & Status  (2)      |
        +------------------------------------------+ 00C0
        |             B2h   (reserved)    (14)     |
        |  Channel    B1h   Maintenance   (1)      |
        |     3       92h   Configuration (31)     |
        |             90h   Run & Status  (2)      |
        +------------------------------------------+ 0090
        |             82h   (reserved)    (14)     |
        |  Channel    81h   Maintenance   (1)      |
        |     2       62h   Configuration (31)     |
        |             60h   Run & Status  (2)      |
        +------------------------------------------+ 0060
        |             52h   (reserved)    (14)     |
        |  Channel    51h   Maintenance   (1)      |
        |     1       32h   Configuration (31)     |
        |             30h   Run & Status  (2)      |
        +------------------------------------------+ 0030
        |           Global Maintenance             |
        +------------------------------------------+ 001D
        |          Global Configuration            |
        +------------------------------------------+ 0011
        |              Run & Status                |
        +------------------------------------------+ 0009
        |             EEPROM Startup               |
        +------------------------------------------+ 0005
        |               High Level                 |
        +------------------------------------------+ 0000
       @endverbatim
  - Below is an alternate view of the memory map showing runtime visibility,
    by mode, and the EEPROM space.

        @par
        @verbatim
                                  Registers

                        ...    +-------------------+ 3FFF
                         .     |      Unused       |
                         .     +-------------------+ 2400
                         .     |   Filter Data     |                      EEPROM
                         .     |                   |
          Visibility     .     +-------------------+ 1000                +-----------
--+ FFF
          ----------     .     |      Unused       |                     | User
Space |
                         .     +-------------------+ 0F00  . . . . . . . +-----------
--+ F00
          Configure .....      |      Filter       |                     |
|
            Mode         .     |   Coefficients    |         One-to-One  |
|
                         .     |                   |          Mapping    |
|
                       __.__   +-------------------+ 0100                +-----------
--+
            Run    __|   .     | Control & Status  |         <========>  |
|
            Mode     |   .     |    Registers      |                     |
|
         (write only) |__.__   +-------------------+ 0000  . . . . . . . +-----------
--+ 000
        @endverbatim


  - This structure must be packed because some two-byte registers are at odd
addresses.

*//**************************************************************************
*************/


#pragma pack(1)
                             //   Addr       Description
typedef struct {             //   ----       -----------

    ///  High-level configuration registers

    Byte  GLBL_SW;           ///<  Addr 0h.  Dummy RAM register.  For testing
device reads & writes
    Byte  GLBL_ID;           ///<  Addr 1h.  Chip ID Including Revision Number
    Byte  FULL_SRST;         ///<  Addr 2h.  Activates all soft resets
    Byte  GLBL_CH_CTRL;      ///<  Addr 3h.  Reset, Enable or Power Down each
channel
    Byte  RUN_MODE;          ///<  Addr 4h.  Set chip in Run or Configure Mode
    ///  EEPROM Startup

    Byte  EE_TRANS;           ///<  Addr 5h.   Control data transfers to/from EEPROM
    Byte  EE_COPY;            ///<  Addr 6h.   Control full transfers to/from EEPROM
    Byte  STARTUP;            ///<  Addr 7h.   Set startup configuration, rate for
EEPROM clock
    Byte  INIT_COUNT;         ///<  Addr 8h.   Initialization delay counter


    ///  Run & Status

    Byte  ENABLE_0;           ///<  Addr 9h.   Enable ADC and system clock per
channel
    Byte  ENABLE_1;           ///<  Addr Ah.   Enable AAF per channel, ADC operation
mode
    Byte  ENABLE_2;           ///<  Addr Bh.   Designate active channels
    Byte  PLL_SIF_STAT;       ///<  Addr Ch.   PLL lock, SIF address out of range
    Byte  EE_VAL;             ///<  Addr Dh.   EEPROM status register value
    Byte  EE_STATUS;          ///<  Addr Eh.   EEPROM transfer status flags
    Byte  ADC_STATUS_0;       ///<  Addr Fh.   ADC out of range, per channel
    Byte  ADC_STATUS_1;       ///<  Addr 10h.  ADC out of range, high or low, per
channel


    ///  Global Configuration

    Byte    PLL_CTRL_0;       ///<  Addr 11h.     PLL Pre-divider, frequency range.
    Byte    PLL_CTRL_1;       ///<  Addr 12h.     PLL loop divider.
    Byte    ADC_CLK_RATE;     ///<  Addr 13h.     Clock rate for ADC, CRC and AREC.
    Byte    SYS_CLK_CTRL;     ///<  Addr 14h.     System Clock control.
    Byte    SPI_CTRL;         ///<  Addr 15h.     Set single-, multi-channel mode
    Byte    SPI_MON;          ///<  Addr 16h.     Monitor internal data transfers
    UInt16  EE_STADDR;        ///<  Addr 17/18h.  EE start address for block
transfers (byte0)
    UInt16  SIF_STADDR;       ///<  Addr 19/1Ah.  Chip start address for block
transfers (byte0)
    UInt16  END_ADDR;         ///<  Addr 1B/1Ch.  Ending address for block transfers
(byte0)


    ///  Global Maintenance

    Byte  SCRATCH[ 8 ];       ///<  Addr 1Dh-24h.  RAM registers.  Available for any
runtime use.
    Byte  SU_UNLOCK;          ///<  Addr 25h.      Lock bit for test/maintenance
(factory use only)
    Byte  GLBL_SRST;          ///<  Addr 26h.      Global soft resets
    Byte  ADC_CTRL;           ///<  Addr 27h.      ADC control.
    Byte  AREC_CTRL;          ///<  Addr 28h.      AREC control.
    Byte  pad1[ 7 ];


    //  Array of channel configuration information.  (Note - there's no padding
after Chan4

    qf4a512_ChannelConfigRegs  Channel1Config;    ///<  Addr 30h.   Channel 1
configuration
                          Byte  pad2[ 14 ];
    qf4a512_ChannelConfigRegs  Channel2Config;    ///<  Addr 60h.   Channel 2
configuration
                         Byte  pad3[ 14 ];
    qf4a512_ChannelConfigRegs  Channel3Config;    ///<  Addr 90h.   Channel 3
configuration
                         Byte  pad4[ 14 ];
    qf4a512_ChannelConfigRegs  Channel4Config;    ///<  Addr C0h.   Channel 4
configuration
                         Byte  pad5[ 8 ];


    //  A few registers are tucked in the 'dead' space after Channel 4
    //  configuration.  Most of them are factory test, so they aren't
    //  included here.

    Byte  DIE_REV;                                ///<  Addr EAh.  Revision of the
silicon die
    Byte  pad6[ 21 ];


    ///  Array of G & H filter coefficients for each channel.

    UInt16  GHCoefficients[ QF4A512_NUM_CHANNELS ]
                          [ QF4A512_NUM_GH_COEFFICIENTS_PER_CHANNEL ];


    ///  Array of FIR coefficients for each channel.  Since the coefficients are 3
bytes
    ///  in length, and there is no 3-byte type in most C compilers, coefficients
    ///  are treated as three byte arrays (hence the third dimension below).

    Byte  FirCoefficients[ QF4A512_NUM_CHANNELS ]
                         [ QF4A512_NUM_FIR_COEFFICIENTS_PER_CHANNEL ]
                         [ QF4A512_NUM_BYTES_PER_FIR_COEFFICIENT ];


}  qf4a512_GlobalRegisters;

#pragma pack()




///  Static value of the CHIP_ID register for a QF4A512
#define  QF4A512_CHIP_ID_NUMBER    0xA0

///  Minimum value of DIE_REV for compatibility with this module (may be higher)
#define  QF4A512_MINIMUM_DIE_REV_NUMBER    0xC1

///  Length of one full frame of data (from /CS low to /CS high again)
#define  QF4A512_LENGTH_OF_FULL_FRAME   (QF4A512_LENGTH_OF_SINGLE_CHAN_FRAME  * \
                                         QF4A512_NUM_CHANNELS_ENABLED)

///  Length of one frame of data from one channel.  A QF4A512 channel has a Flags
byte and two data bytes.
#define  QF4A512_LENGTH_OF_SINGLE_CHAN_FRAME   3

///  Bit position, in the Flags byte of the Run mode data stream, of the bit
indicating new data
```

```
#define  QF4A512_NEW_DATA_BIT   BIT4



///  Lowest address of the G & H filter coefficient memory
#define  QF4A512_MIN_GH_FILTER_COEFF_ADDRESS    \
            GetElementOffset( qf4a512_GlobalRegisters, GHCoefficients )


///  Lowest address of the User FIR filter coefficient memory
#define  QF4A512_MIN_FILTER_COEFF_ADDRESS    \
            GetElementOffset( qf4a512_GlobalRegisters, FirCoefficients )


///  Highest address of the filter coefficient memory
#define  QF4A512_MAX_FILTER_COEFF_ADDRESS     sizeof(qf4a512_GlobalRegisters)


///  Maximum address of EEPROM
#define  QF4A512_MAX_EEPROM_ADDRESS    0xfff


///  Maximum useful configuration address.  This address leaves out the filter
coefficients
///  between 0x1000 and 0x2400, but none of the functions in this module access
that area.
#define  QF4A512_MAX_REGISTER_ADDRESS     QF4A512_MAX_FILTER_COEFF_ADDRESS



////////////////////////////////////////////////////////////////////////////////////
//  Addresses of common registers within the QF4A512


///  Address of the Chip ID register
#define  QF4A512_GLBL_ID_ADDRESS     GetElementOffset( qf4a512_GlobalRegisters,
GLBL_ID )

///  Address of the Chip ID register
#define  QF4A512_FULL_SRST_ADDRESS   GetElementOffset( qf4a512_GlobalRegisters,
FULL_SRST )

///  Address of the Die Revision register
#define  QF4A512_RUN_MODE_ADDRESS    GetElementOffset( qf4a512_GlobalRegisters,
RUN_MODE )

///  Address of the Die Revision register
#define  QF4A512_DIE_REV_ADDRESS     GetElementOffset( qf4a512_GlobalRegisters,
DIE_REV )

///  Address of the SPI Control register
#define  QF4A512_SPI_CTRL_ADDRESS    GetElementOffset( qf4a512_GlobalRegisters,
SPI_CTRL )
```

```
///  Configuration options for the QF4A512 port.
typedef enum  {

    SetIdleMode,
    SetRunMode,
    WriteConfigRegisters,
    ReadConfigRegisters,
    ReadEeprom,
    WriteEeprom

} qf4a512_ConfigRequest;




#ifdef __cplusplus
   extern "C"
{                             //  C++ source requires this linkage specification
#endif


void  qf4a512_Init( void );

void  qf4a512_DeInit( void );

Result  qf4a512_ReadSamples(
    const Handle   Device,
          UInt16   Buffer [][ QF4A512_NUM_CHANNELS_ENABLED ],
    const Count    Length);

void  qf4a512_ReadConfigRegisters(
    const Handle  Device,
          Count   StartingAddr,
          Byte   * Buffer,
    const Count   Length);

Result  qf4a512_WriteConfigRegisters(
    const Handle  Device,
          Count   StartingAddr,
    const Byte   * Buffer,
    const Count   Length);

Result  qf4a512_ReadEeprom(
    const Handle  Device,
    const Count   StartingAddr,
          Byte   * Buffer,
    const Count   Length);

Result  qf4a512_WriteEeprom(
    const Handle  Device,
    const Count   StartingAddr,
    const Byte   * Buffer,
    const Count   Length);

Byte  qf4a512_ReadConfigByte(
    const  Handle  Device,
    const  Count   Address );

void  qf4a512_WriteConfigByte(
    const  Handle  Device,
    const  Count   Address,
    const  Byte    Value);

void  qf4a512_WriteEepromByte(
    const  Handle  Device,
    const  Count   Address,
    const  Byte    Value);

Byte  qf4a512_ReadEepromByte(
    const  Handle  Device,
    const  Count   Address);

void  qf4a512_ExitRunMode(
    const Handle  Device );


#ifdef __cplusplus
}                                      //  End of C++ linkage specification
#endif


#endif
```

Qf4a512-functional.h

```
/******************************  (C) Quickfilter Technologies, Inc.
*************************//*!

  @file  Qf4a512-functional.h

  Implementation of high-level Qf4a512 features (Small model).

  $Id: Qf4a512-functional.h 122 2006-07-24 20:25:01Z jhopson $

*///*****************************************************************************
**************


#if !defined(QF4A512_FUNCTIONAL_H_INCLUDED)           //   Inclusion control
#    define  QF4A512_FUNCTIONAL_H_INCLUDED



#include "Project.h"


///  Each entry in the configuration register table, produced by the PC software,
contains
///  an address/value pair.

typedef struct
{
    UInt16   Address;
    Byte     Value;

} qf4a512_ConfigTableEntry;


#ifdef __cplusplus
```

```
   extern "C"
{                             //  C++ source requires this linkage specification
#endif


Result  qf4a512_LoadImageRegisterTable(
    const  Handle                     Device,
    const  qf4a512_ConfigTableEntry * Table,
    const  Count                      SizeOfTable );

Result  qf4a512_WriteImageRegisterTableToEeprom(
    const  Handle                     Device,
    const  qf4a512_ConfigTableEntry * Table,
    const  Count                      SizeOfTable );

void  qf4a512_ResetDevice( const Byte Device );



#ifdef __cplusplus
}                                     //  End of C++ linkage specification
#endif


#endif
```

Display Module Code

```
#include "msp430x44x.h"
#include <in430.h>

#define a (0x80) // definitions for LCD seegments on the Olimex LCD. 4-Mux
operation is assumed
#define b (0x40) // For more details on 4-Mux operation, gather your LCD datasheet,
#define c (0x20) // TI's MSP430F449 User Guide (look for LCD Controller, then 4-
Mux),
#define d (0x01) // and MSP-449STK-2 schematic. You will need ALL these 3 when
defining
#define e (0x02) // each number or character. Remember, the Olimex LCD doesn't use
a LCD driver!
#define f (0x08) // You tell the LCD what characters to display. It's very time
consuming!!
#define g (0x04)
#define h (0x10)

// Variables for buttons
#define BTN1                (BIT4)
#define BTN2                (BIT5)
#define BTN3                (BIT6)
#define BTN4                (BIT7)
#define BTN_MASK            (BTN1 | BTN2 | BTN3 | BTN4)

// Variables for count
char *LCD = LCDMEM;                  //Pointer to LCD Memory Segments
float occurence[2] = {0, 0};         //Heart beat occurence time record array
```

```
unsigned char i = 0;
unsigned char next = 0;
unsigned char j;                      //Conversion factor
float time = 0;                       //Time
float t1 = 0;
float t2 = 0;
float dt = 0;

// Determines whether we're displaying the rotating information
// Or asking for input from the user
int rotating = 0;

unsigned char busy;
unsigned int ACC = 0;
unsigned int bufferACC = 0;
unsigned int volts = 0;
unsigned int restHR = 0;
unsigned int HR = 0;                  //Heart rate
unsigned int AC = 0;                  //Accelerometer
unsigned char switchCnt = 0;          //Display switch counter
unsigned char adc_in = 0;
unsigned int avgHR = 0;
unsigned char HRcnt = 0;
unsigned int bufferHR = 0;
unsigned int vectorHR[256] = {0};     //Store ADC12 measurements
unsigned char arrayHR[256] = {0};     //Store past HR calculations
unsigned char charHR[4];              //Store character version of heart rate
unsigned char beat = 0;
int counts = 0;
double rate = 0;
double carbs = 0;

// Button declarations
int variablevalue = 0;                //Variable value will be 3 digits, start at 0
int displayName = 1;                  // 1 if display name, 0 if button has been
pressed
char digits[3] = {0, 0, 0};
char buttonStatusPrev = 0;            // Previous status of buttons
char buttonStatus = 0;                // Current status of buttons
char buttonPressed = 0;               // Pressed buttons
char buttonReleased = 0;              // Released buttons
int inputtype = -1;

int sex = 0;
int age = 0;
int weight = 0;

int firstByte = 0;
int INbyte;                       //INbyte is the received number from
Accelerometer


// Function Headers
//void main();
//void setupTimerB(void);
//void setupEKG(void);
void init_sys();
void clearLCD();
```

```
double AEEaccell (int counts, int age, int sex);
double AEEhr (double HRaS, int age, int sex);
double AEEcombined (int age, int sex, int weight);

// Function Headers for Buttons
void buttonHandler();
void delayX (unsigned int x);
void rotationHandler();
void writeLetter(int position, char letter);
void writeWord(const char *word);

/******************** MAIN FUNCTION ********************/
void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;             // Stop watchdog timer
  init_sys();
 // configUART();

 /* unsigned int txSample = 144;
  unsigned char txSample1 = txSample & 0xff;
  unsigned char txSample2 = txSample >> 8;


 // unsigned int temp = txSample2 + 0x00;
//  temp = temp << 8;
 // temp |= txSample1;

 unsigned char temp = (1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1) & txSample2;
 temp = temp << 8;
 temp = temp | txSample1;

  unsigned char j=100;

  int ans = (int)temp;

  if (ans == txSample)
    writeWord("YES");
  else
    writeWord("NO");*/


  /*// Display the heart rate one number at a time
  for(char i=7; i>4;i--)
  {
    char y = (ans/j) + 48;
    writeLetter(i, y);
    if (y != 0) ans =(ans%j);
      j=j/10;
  }

  exit();*/

  while(1) {
    // Always call the button handler, even if we're displaying
    // the rotating information
    buttonHandler();

    // Only call the rotation handler if we're displaying
    // the rotating information
    if (rotating == 1)
      rotationHandler();
  }
}


/******************** DISPLAY HEARTRATE ********************/
void dispHR(unsigned int hR) {
  unsigned int j=100;

  // Display BPM
  writeWord("BPM");

  // Display the heart rate one number at a time
  for(char i = 7; i > 4; i--)
  {
    char y = (hR/j) + 48;
    writeLetter(i, y);
    if (y != 0) hR =(hR%j);
      j=j/10;
  }
}
/******************** DISPLAY ACCELEROMETRY ********************/
void dispACC(unsigned int aC) {
  unsigned int j=1000;

  // Display ACC
  writeWord("CT");

  // Display the Accelerometry one number at a time
  for(char i = 7; i > 3; i--)
  {
    char y = (aC/j) + 48;
    writeLetter(i, y);
    if (y != 0) aC =(aC%j);
      j=j/10;
  }
}
/******************** DISPLAY CARBS ********************/
void dispCarbs(unsigned int cR, char *str) {
  unsigned int j=100;

  // Display CARBS
  writeWord(str);

  // Display the carbs one number at a time
  for(char i = 7; i > 4; i--)
  {
    char y = (cR/j) + 48;
    writeLetter(i, y);
    if (y != 0) cR =(cR%j);
      j=j/10;
  }
}
/********** TimerB Interruup Service Routine ************/

#pragma vector = TIMERB0_VECTOR
```

```
__interrupt void Timer_B0(void)
{
  time += 0.01;                               //increment time
  t1 += 0.01;
  t2++;

  if ((int)t2 % 1000 == 0 && rotating == 1) {                // Then multiplied the
number back by 5,
                                              // so basically if the remainder is >0
then
                                              // it decides that it's not ready
    if (restHR == 0)
      restHR = bufferHR;

    restHR = 60;

    //Implement the combined algorithm
    double comb = AEEcombined (age, sex, weight);

    inputtype++;                              //Says that the input type is
incrementing

    if (inputtype > 4)                        //if the input type goes above 4 (ACC)
it should go back to HR
      inputtype = 3;                          //Input type for HR
  }


  // 15 second epoch, only if we're displaying the rotating information
  if((int)t2 % 500 == 0 && rotating == 1)
  {
   // if (sex !=0 && age != 0 && weight !=0)
     // AC = (int)(AEEaccell(age, sex));

    adc_in = 0;
    bufferHR = avgHR/(unsigned int)HRcnt;
    avgHR = 0;
    HRcnt = 0;
  }
}

/********** HEART RATE SIGNAL PROCESSING *************/
//interrupt triggered when a digital pulse is acquired
#pragma vector = PORT2_VECTOR

__interrupt void Port2_ISR(void)
{
//  buzzerOn();                             //Make sound for each heart rate
// swDelay(1);                             //Delay to make sure sound is heard

  if(next == 0)                          //If very first heart beat time
                                         //Is not recorded...
  {
    occurence[0] = time;                 //record time of very first heart beat
    next = 1;                            //Very first heart beat time is recorded
  }
  else
  {
    occurence[1] = time;                 //Record heart beat time
    dt = (occurence[1] - occurence[0]); //Calculate time difference between two
beats
    HR = (int)(60/dt);                  //Calculate heart rate
    occurence[0] = occurence[1];        //Second heart beat time to first heart
beat time
    avgHR += HR;                        //Actual average calculated later in the
timer interrupt
    HRcnt ++;

  }

  P2IFG &= ~(BIT0);
}

/****************** ROTATION HANDLER ****************/
void rotationHandler() {
  // First clear the screen
  clearLCD();

  // Inputtype used to determine what shows up on the LCD --> HR and Acceeleration
will cycle, carbs will be displayed upon request
  switch(inputtype)
  {
    case 3:  dispHR(bufferHR);  break;
    case 4:  dispACC(INbyte); break;
    case 5:  dispCarbs(carbs, "CG"); break;
    case 6:  dispCarbs(rate, "GPM"); break;
    //default: clearLCD(); break;
  }
}


/***************** BUTTONS ************************/
void buttonHandler(void)
{
  if (buttonPressed)
  {
    // software delay
    #ifdef SIZE_OPTIMIZED
      delayX (300);
    #else
      delayX (300);
    #endif
  }

  buttonStatusPrev = buttonStatus;
  buttonStatus = ~(P3IN & BTN_MASK);
  buttonPressed = (buttonStatusPrev ^ buttonStatus) & buttonStatus;
  buttonReleased = (buttonStatusPrev ^ buttonStatus) & buttonStatusPrev;

  // Display the correct input type
  if (displayName == 1 && rotating != 1) {
   switch(inputtype)
    {
      case 0:  writeWord("GENDER"); break;
      case 1:  writeWord("AGE"); break;
      case 2:  writeWord("WEIGHT"); break;
      case 3:  writeWord("RESTING"); break;
      default: clearLCD(); break;
    }
  }

  if (buttonPressed) {
    // Make sure we're not rotating
    if (rotating == 0) {
      // Make sure input type is between (inclusive) 0 and 2
      if (inputtype == -1)
        inputtype = 0;
      else {
        // Button One is Pressed
        if (buttonPressed & BTN1)   //this is saying if the button pressed is the
first button
        {
          variablevalue = digits[0] + 10*digits[1] + 100*digits[2];

          switch(inputtype)
          {
            case 0: sex = variablevalue; break;
            case 1: age =  variablevalue; break;
            case 2: weight = variablevalue / 2.2; break;
            case 3: restHR = variablevalue; restHR = bufferHR; rotating = 1; break;
          }

          inputtype++;                      //increment input type up to
inputtype==2

          digits[0]= 0;
          digits[1]=0;
          digits[2]=0;
          variablevalue=0;                //then set all buttons back to zero
          displayName = 1;
        }
        else
          displayName = 0;

        // Button Two is Pressed
        if (buttonPressed & BTN2 && inputtype != 0 && inputtype != 3)   //repeat
with button 2
        {
          // Make sure the number is within range for the AGE case
          if (inputtype == 1) // Make sure the digit does not exceed 1
            if (digits[2] + 1 > 1)
              digits[2] = -1;

          // Make sure the digit does not exceed 9
          if (digits[2] + 1 > 9)
            digits[2] = -1;

          // Button TWO does not work for the SEX or AGE case
          digits[2] += 1;
          variablevalue = digits[0] + 10*digits[1] + 100*digits[2];
        }

        // Button Three is Pressed
        if (buttonPressed & BTN3 && inputtype != 0 && inputtype != 3)  //repeat
with button 3
        {
          // Make sure the number does not exceed the range for the AGE case
          if (inputtype == 1) // Make sure it does not go above 120 years old
            if (digits[2] == 1 && digits[1] + 1 > 2)
              digits[1] = -1;

          // Make sure digit does not exceed 9
          if (digits[1] + 1 > 9)
            digits[1] = -1;

          // Button THREE does not work for the SEX case
          digits[1] += 1;
           variablevalue = digits[0] + 10*digits[1] + 100*digits[2];
        }

        // Button Four is Pressed
        if (buttonPressed & BTN4 && inputtype != 3)         //if the button pressed
is button 4
        {

            // Make sure the number is in range for SEX
            if (inputtype == 0) // Make sure the range does not go over 1
              if (digits[0] + 1 > 1)
                digits[0] = -1;

            // Make sure the digit does not exceed 9
            if (digits[0] + 1 > 9)
              digits[0] = -1;

             digits[0] += 1;
             variablevalue = digits[0] + 10*digits[1] + 100*digits[2];
        }

        if (rotating == 0) {
          char variable[4];

          variable[0]=digits[2]+0x30;
          variable[1]=digits[1]+0x30;
          variable[2]=digits[0]+0x30;
          variable[3]=0;

          // Clear the LCD before writing
          clearLCD();

          // Write the digits to the LCD
          writeWord(variable);
        }
      }
    }
    else if (rotating == 1) {    // If we're rotating
      // Reset the carbs
      if (buttonPressed & BTN1)
          carbs = 0;

      // Display the cummulative carbs
      if (buttonPressed & BTN2)
        inputtype = 5;

      // Display the rate of carbs burned
      if (buttonPressed & BTN3)
        inputtype = 6;
      // Display the heart rate
      if (buttonPressed & BTN4)
        inputtype = 3;
    }
  }
}

//Display numbers as user enters them
void writeNumber(int position, int letter)
{
  if (letter >= 0 && letter <= 9)
    writeLetter(position, (char)(letter + 48));
}

/************************  ALGORITHM ************************/
//HRaS = bufferHR - 0.83*bufferHR

//Calculate the Activity Algorithm
double AEEaccell (int counts, int age, int sex)
{
  if (counts < 133)
    return ((((0.203 * 133) - (0.75 * age) + (83 * sex) + 46) / 133) * counts) / 
4186.8;
  else
    return ((0.203 * counts) - (0.75 * age) + (83 * sex) + 46) / 4186.8;
}

//Calculate the Heart rate Algorithm
double AEEhr (double HRaS, int age, int sex)
{
  if (HRaS < 23)
    return ((((5.95 * HRaS) + (0.23 * age) + (84 * sex) - 134) / 23) * HRaS) / 
4186.8;
  else
    return ((5.95 * HRaS) + (0.23 * age) + (84 * sex) - 134) / 4186.8;
}

//Calculate Algorithms Combined
double AEEcombined (int age1, int sex1, int weight1)
{
  // Convert accelerometer data to per minute
  int counts = INbyte * 6;

  // Calculate the heart rate above sleep
  int HRaS = bufferHR - 0.83 * restHR;

  // Define the acceleration and heart rate algorithms
  double CAC = AEEaccell(counts, age1, sex1);                           //Where 
'CAC' is acclereation for the combined algorithm
  double CHR = AEEhr(HRaS, age1, sex1);                          //Where 'CHR' is 
heart rate for the combined algorithm

  double res = 0;                                             //Where 'res' is 
result

  if (counts < 25 && HRaS < 23)
    res = ((0.1 * CHR) + (0.9 * CAC)) * (weight1 * 0.45);
  else if (HRaS > 23 && HRaS < 80)
```

```
    res = ((0.5 * CHR) + (0.5 * CAC)) * (weight1 * 0.62);
  else if (counts > 25 && HRaS >= 80)
    res = ((0.9 * CHR) + (0.1 * CAC)) * (weight1 * 0.87);

  // Keep track of the rate (per minute) and the sum of carbs burned
  rate = res;
  carbs += res / 6;                                                 //Incrementing
value of carbohydrates is stored here

  return res;
}

/************************ 449UART ***********************************/

//******************************************************************************
//  MSP-FET430P440 Demo - USART0, 9600 UART Echo ISR, DCO SMCLK
//
//  Baud rate divider with 1048576hz = 1048576Hz/9600 = ~109.23 (06Dh|03h)
//  ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO = 32 x ACLK = 1048576Hz
//  //* An external watch crystal between XIN & XOUT is required for ACLK *//
//
//                MSP430F449
//             -----------------
//         /|\|              XIN|-
//          | |                 | 32kHz
//          --|RST          XOUT|-
//            |                 |
//            |             P2.4|------------>
//            |                 | 9600 - 8N1
//            |             P2.5|<------------
//
//
//  M. Buccini
//  Texas Instruments Inc.
//  Feb 2005
//  Built with IAR Embedded Workbench Version: 3.21A
//*****************************************************************************

void send()
{
  char dmy = 0;

  while (!(IFG1 & UTXIFG0))                // USART0 TX buffer done
    dmy=dmy;
}

void receive()
{
  char dmy = 0;

  while (!(IFG1 & URXIFG0))                // USART0 RX buffer FULL
    dmy=dmy;
}

void configUART(void)
{
  unsigned char  INbyte;//, OUTbyte=0;

  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  FLL_CTL0 |= XCAP18PF;                     // Configure load caps

  // Configs UART0 to work at 9600 baud Transmit -- Interrupts not enabled
  // Use this for the receive byte - it is 449
  P2SEL |= 0x30;                            // P2.4,5 = USART0 TXD/RXD
  ME1  |= UTXE0 + URXE0;                    // Enable USART0 TXD/RXD
  UCTL0 |= CHAR;                            // 8-bit character
  UTCTL0 |= SSEL1;                          // UCLK = SMCLK
  UBR00 = 0x6D;                             // 1MHz 9600
  UBR10 = 0x00;                             // 1MHz 9600
  UMCTL0 = 0x03;                            // modulation
  UCTL0 &= ~SWRST;                          // Initialize USART state machine
  P2DIR |= BIT4;                            // P2.4 output direction
  P2DIR &= ~BIT5;                           // P2.5 input direction

 /* Receive Acclerometer information*/

  int cd = 0;
  while(1)
  {
    //Receive byte
    receive();
    unsigned int tempByte = RXBUF0;  //tempByte stores it whenever it gets it
    /*
    unsigned int temp = txSample2 + 0x00;
  temp = temp << 8;
  temp |= txSample1;
    */
    if (cd % 2 == 0) { //if it's even then it does the logical or
      INbyte = tempByte + 0x00;
      INbyte = INbyte << 8;
      INbyte |= firstByte;
    }
    else { //if it's odd then it does the logical and and left shift
      firstByte = tempByte;
      cd = 0;
    }

    //display_byte(INbyte);

    cd++;
  }
}



/*********************** DELAY FOR BUTTONS *******************/
void delay (unsigned int x)                         //9+a*12 cycles
{
  unsigned int i;

  for (i = 0; i < x; i++);
}

void delayX (unsigned int x)
{
  unsigned int i;

  for (i = 0; i < x; i++)
```

```
    delay (255);
}
/************The code below this line is from the ECE2801 course that has been
editted to fit the project*************************/

/******************** initSys() *****************************/
void init_sys(void)
{
  // Setup LCD
  FLL_CTL0 = XCAP10PF;                    //Set load capacitance for 32k xtal
    // Initialize LCD driver (4Mux mode)
  LCDCTL = LCDSG0_7 + LCD4MUX + LCDON;  // 4mux LCD, segs16-23 = outputs
  BTCTL = BT_fLCD_DIV128;               // Set LCD frame freq = ACLK
  P5SEL = 0xFC;                         // Set Rxx and COM pins for LCD
  clearLCD();                // Clear LCD display

  // Setup Button ports
  P3SEL &= ~(BIT7|BIT6|BIT5|BIT4);  // P3.7-4 I/O Function [0000 xxxx]
  P3DIR &= ~(BIT7|BIT6|BIT5|BIT4);  // P3.7-4 Push buttons input [0000 xxxx]

  // Set up heart beat port
  P2SEL &= ~(BIT0);             // P2.0 I/O option (0)
  P2DIR &= ~(BIT0);             // Set P2.0 to input direction (0)
  P2IES |= (BIT0);              //Flag set with high to low transition (1)
  P2IFG &= ~(BIT0);
  P2IE |= (BIT0);               //Enable intterupt

  // Set up the timer B
  TBCTL = TBSSEL_2 + CNTL_0 + MC_1;
    //TBSSEL: sourc SMCLK
    //CNTL: 16 bit
    //MC: upmode to CCR0
    //ID: input CLK divided by 1
  TBCCR0 = 10485;               //10485 SMCLK ticks = 0.01 second
  TBCCTL0 = CCIE;               //TBCCR0 interrupt enabled


  _BIS_SR(GIE);               //Global interrupt enable
}

// **************************** clearLCD ***************************************
void clearLCD(void)                     // Makes the LCD blank
{                                       // Clear LCD memory to clear display
  unsigned int iLCD;

  for (iLCD =0; iLCD<20; iLCD++)        // Clears all 20 LCD memory segments
    LCD[iLCD] = 0;
}

// *************************** writeLetter ************************************
void writeLetter(int position,char letter) // writes single character on the LCD.
{                                          // User can specify position as well
  // DO NOT PLAY WITH THE CODE BELOW -----------------------------------------
  if (position == 1)                       // This is position adjustment for
compatibility
    position = position + 6;
  else
    if ((position > 1) & (position < 8))
      position = ((position * 2) - 1) + 6; // Adjust position
  // ------------------------------------------------------------------------

  switch(letter)
  {
    // Letter // LCDM7 // LCDM8 // End
    case 'A': LCD[position-1] = a + b + c + e; LCD[position] = b + c + g; break;
    case 'B': LCD[position-1] = c + h + e; LCD[position] = b + c + g; break;
    case 'C': LCD[position-1] = a + h; LCD[position] = b + c; break;
    case 'D': LCD[position-1] = b + c + h + e; LCD[position] = c + g; break;
    case 'E': LCD[position-1] = a + h + e; LCD[position] = b + c + g; break;
 //   case 'F': LCD[position-1] = a; LCD[position] = b + c + g; break;
    case 'G': LCD[position-1] = a + c + h + e; LCD[position] = b + c; break;
    case 'H': LCD[position-1] = b + c + e; LCD[position] = b + c + g; break;
    case 'I': LCD[position-1] = a + h + f; LCD[position] = d; break;
 //   case 'J': LCD[position-1] = b + h + c; LCD[position] = c; break;
 //   case 'K': LCD[position-1] = d + g; LCD[position] = b + c + g; break;
 //   case 'L': LCD[position-1] = h; LCD[position] = b + c ; break;
    case 'M': LCD[position-1] = b + c + g; LCD[position] = b + c + f; break;
    case 'N': LCD[position-1] = b + c + d; LCD[position] = b + c + f; break;
    case 'O': LCD[position-1] = a + b + c + h; LCD[position] = b + c; break;
    //file:///R|/MQP/HR_ACC_disp_2.txt (5 of 7) [4/23/2008 2:34:10 AM
    //file:///R|/MQP/HR_ACC_disp_2.txt
    case 'P': LCD[position-1] = a + b + e; LCD[position] = b + c + g; break;
//    case 'Q': LCD[position-1] = a + b + c + h + d; LCD[position] = b + c; break;
    case 'R': LCD[position-1] = a + b + d + e; LCD[position] = b + c + g; break;
    case 'S': LCD[position-1] = a + c + h + e; LCD[position] = b + g; break;
    case 'T': LCD[position-1] = a + f + b; LCD[position] = d + b; break;
//    case 'U': LCD[position-1] = b + c + h; LCD[position] = b + c; break;
//   case 'V': LCD[position-1] = g; LCD[position] = b + c + e; break;
    case 'W': LCD[position-1] = b + c + d; LCD[position] = b + c + e; break;
    case 'X': LCD[position-1] = d + g; LCD[position] = e + f; break;
//    case 'Y': LCD[position-1] = b + c + h + e; LCD[position] = f; break;
//    case 'Z': LCD[position-1] = a + h + g; LCD[position] = e; break;
    // number // LCDM7 // LCDM8 // END
    case '0': LCD[position-1] = a + b + c + h; LCD[position] = b + c; break;
    case '1': LCD[position-1] = b + c; LCD[position] = d & a; break;
    case '2': LCD[position-1] = a + b + e + h; LCD[position] = c + g; break;
    case '3': LCD[position-1] = a + b + c + e + h; LCD[position] = g; break;
    case '4': LCD[position-1] = b + c + e; LCD[position] = b + g; break;
    case '5': LCD[position-1] = a + c + h + e; LCD[position] = b + g; break;
    case '6': LCD[position-1] = a + c + h + e; LCD[position] = b + c + g; break;
    case '7': LCD[position-1] = a + b + c; LCD[position] = d & a; break;
    case '8': LCD[position-1] = a + b + c + e + h; LCD[position] = b + c + g;
break;
    case '9': LCD[position-1] = a + b + c + e ; LCD[position] = b + g; break;
    // others
//    case '.': LCD[position] = h; break; // decimal point
//    case '^': LCDM2 = c; break; // top arrow
//    case '!': LCDM2 = a; break; // bottom arrow
//    case '>': LCDM2 = b; break; // right arrow
//   case '<': LCDM2 = h; break; // left arrow
//    case '+': LCDM20= a; break; // plus sign
//    case '-': LCDM20= h; break; // minus sign
//    case '&': LCDM2 = d; break; // zero battery
//    case '*': LCDM2 = d + f; break; // low battery
//    case '(': LCDM2 = d + f + g; break; // medium battery
//    case ')': LCDM2 = d + e + f + g; break; // full battery */
  }
}
```

```c
// ****************************** writeWord ***********************************

void writeWord(const char *word)    // Displays a word upto 7 characters -- why 7?
                                    // Words must be in upper case (why?)
{
  unsigned int strLength = 0;       // Variable to store length of word
  unsigned int i;                   // Dummy variable

  strLength = strlen(word);         // Get the length of word now

  for (i = 1; i <= strLength; i++)  // Display word
    writeLetter(strLength - i + 1,word[i-1]); // Displays each letter in the word
}
```