Mobile Athlete Monitoring System
M.A.M.S.

Computer Engineering Senior Project
Final Project Documentation

Daniel Barton
Farhan Mohammed
Michael Pedersen
Michael Smith

I. Table of Contents

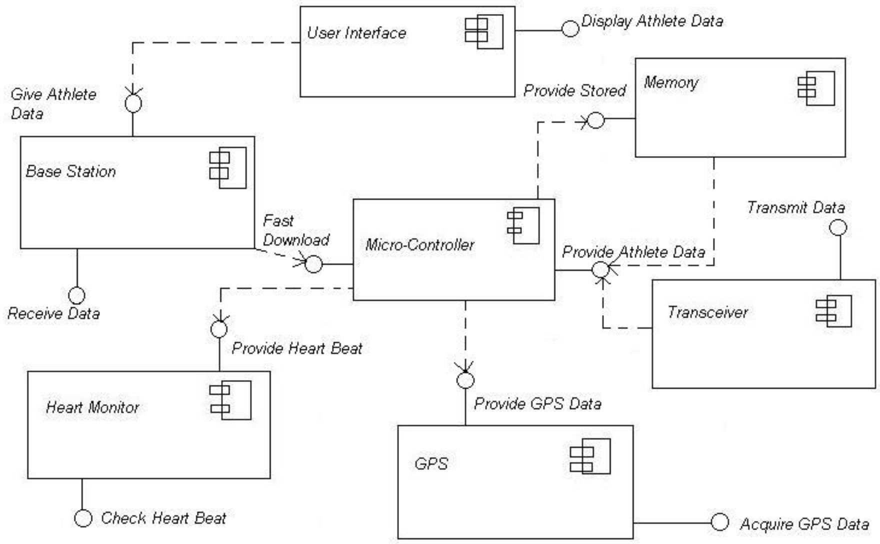
Motivation

This idea was initially identified as a need from the U.S. ski team. There is an incredible amount of data collected on the U.S. ski team athletes. However, most of this data is obtained from within the walls of controlled gyms and facilities. There is a need to obtain real-time information from the athlete while they are on the actual racing track.

A great deal of effort is spent in helping train athletes to perform better. Coaches and trainers analyze all the data they can get (e.g. heart rate, respiration rate, lactic acid levels) as an athlete performs in a training center to help determine their limits and where they may improve. Data collected in this controlled environment however, cannot accurately describe how the athlete will perform on an actual racecourse; say in a mountainous terrain with temperatures below zero. This project is to assist the coaches in the training of their athletes by supplying them with the best and most accurate information possible; by using our Mobile Athlete Monitoring System (MAMS) we will be able to obtain the actual data from the athlete for complete analysis in an entirely real training session, the actual course! With this realistic track data, both coach and athlete could work on obtaining the most optimal performance. Comparing this real-time data with already known performance maximums will allow an analysis of where an athlete is under-performing, and where the athlete is over-performing. This technology will definitely help to improve the athlete's performance in competition, and may also be modified to act as an aid for athletes in other disciplines.

Overview

This MAMS is designed to gather various vitals and positional data on an exercising Athlete. At conception, it was planned to provide the athlete's heart rate, respiration rate, global position, velocity, and altitude. However, in this final implementation the unit was only built to obtain heart rate, global position, velocity, and a fairly limited reading on altitude. As the MAMS system gathers this data on the athlete, it stores the obtained information in non-volatile memory for later retrieval and transmits real-time statistics to a base station for monitoring. The system is easily divided into components to accomplish its various tasks. Below is a U.M.L. block diagram to help in understanding how the system is implemented.



In the previous diagram, the microcontroller, heart monitor, G.P.S., memory, and a transceiver are all carried with the athlete. The base station has a transceiver as well to receive information from the athlete. The base station transceiver is connected to a laptop which displays the athlete's data through a graphical user interface. In our implementation of the MAMS, we needed to interface many pre-built parts. For the G.P.S. component we used a Garmin Geko G.P.S. unit. For the heart monitor we used a Polar chest strap Heart Monitor. For the athlete transmitter and base station receiver we used AeroComm's AC4490-1000 transceiver in each position. The memory component was implemented with Atmel's AT29BV020, a two megabit flash memory. Lastly, as our micro-controller we used the Freescale/Motorola MC9S08GB60. The user interfaces were implemented with C# software on a laptop running XP Pro and Sequel Server.

The MAMS system is started by turning on the power to the G.P.S. device, turning on the transceiver, then supplying power to the microcontroller and memory. However, you can start the components in any random order. The G.P.S. and transceiver work off their own power supplies; whereas, the microcontroller and memory work off the same power supply, provided by the development board for our microcontroller. Because of the different power supplies introduced in to the MAMS system, it is important to make sure that all the components that the athlete is carrying have the same common ground reference. We achieved this by utilizing the ground provided on the microcontroller's development board for all the devices.

At startup, control is given to the G.P.S. component. The G.P.S. device then provides the data through a serial interface cable to the microcontroller. After receiving this information, the microcontroller parses it and passes control to the heart monitor. Next, the heart monitor senses the heart beat, and sends this information to the microcontroller which waits for a second heart beat and calculates the difference between the two signals. This data is also parsed and the microcontroller checks on whether a write to memory is needed. If so the microcontroller writes this data to memory, these writes are done every third measurement. Approximately every third write, the MAMS transmits the data to the base station for real-time analysis. The process continues in this loop until it is switched off. At the end of the session the more complete athlete data can then be downloaded to a computer via a U.S.B. plug. Given all our memory space and power calculations the MAMS is capable of recording data for up to four hours, the length of most races.

The preceding has been a quick abstract as to how the entire process works. In the next few sections, we will discuss in greater detail the implementation, problems, and limitations of each device as well as how they interface together.

Microcontroller

In this project we used Freescale's MC9S08GB60. We chose this particular microcontroller because it had a large number of I/O ports, well implemented SCI functionality, low power consumption, good tolerance to cold temperatures, large code space, as well as a good set of tools to help in the development process. As we began to work with the chip and the tools that were associated with it, it became apparent that we needed to purchase the development board.

The development board provided a number of things that proved the success of our project; a diagram of the board is shown in appendix A. First, the board already had everything set up to use the Code Warrior Development Studio so we could download our code to the microcontroller. This quickly helped in the development process because we didn't have to spend much time figuring out how to load programs onto the microcontroller. The board also came equipped with two RS232 ports which contributed greatly to the ease in acquiring serial data from the G.P.S. component. The most important advantage that the development board provided was easy access to the pins. The microcontroller was only about ½" square. With 64 pins crowded in to that small of an area, it would have been very difficult connect wires correctly. The board spread these pins out to make them more accessible for soldering and testing purposes.

Besides the development board and the many I/O pins on the microcontroller, the chip had a very large amount of built in flash memory. This proved to be incredibly advantageous to us in our project. First, the large amount of memory enabled us to store large chunks of data to off-chip non-volatile memory all at once. The other major benefit of this on-chip memory is that it enabled us to write plenty of code in C instead of Assembly, and not have to worry about where to store it. Because of this space, we set aside a buffer on chip to construct the data packets for our athlete, and fill the 256k byte blocks for the off-chip memory. This is where the "provide athlete data" interface was made available. At any point in the program, we could index whatever data might be needed from this buffer that was set aside, and output the data serially or via the parallel port over any of the configured output pins. For information on how to configure the various input and output pins, and other setup options for the micro-controller pins, please refer to the data sheet for the microcontroller, MC9S08GB60.

We did make two critical mistakes in the set up of our microcontroller. The first problem was that we didn't provide an external clock source for the microcontroller. We used an internally generated clock that runs at 8 MHZ, providing a bus clock of 4 MHZ. This decreased our cost of the overall system. However, the internally generated clock on the microcontroller, as explained in the microcontroller data sheet, had a variance of +-25%. This proved to be difficult in our system because we used the SCI functions so heavily, and because we used some polling that was dependent on the system clock. If one was to fully implement this system, an external clock would be needed to eliminate some of the timing bugs currently found in the system. The timing causes some problems when it is both receiving and transmitting data. The timing can get a little off when reading the serial input from the G.P.S. component. The timing also gets off when sending a serial data stream to the transceiver for transmission to the base station. The last place where this causes a problem is in the heart monitor component. In the heart monitor component, the microcontroller counts the number of clock cycles past to determine the amount of time between heart beats. This clock variance also introduces error into that calculation, which brings us to the next problem in the implementation of our microcontroller.

The second problem is linked to the timing for the heart monitor. We made a mistake in the mapping of our pins. The external memory utilized the microcontroller's output compare and input capture pins. Because of this mapping an accurate count on the time could not be taken from the clock source itself, but rather the time had to be calculated based on instructions executed. This problem wasn't detected until late in the term, and was not able to be corrected due to time constraints.

There are a couple of known problems in the design of the MAMS system, in regards to the micro-controller. First, the internally generated clock has a definite variance. Because the sampling used in the MAMS system needs to be precise for the measurements and samples to be correct, an external clock would be beneficial to correct small variances introduced in the sampling of the various, or added components. The second problem was just a mistake on the decision of the pin assignments. The pins for output and input compare should have been left free to help in the sampling of current and future sensors.

External Memory Chip

The purpose of the external flash memory chip is to be able to record the data generated by the heart rate monitor and the G.P.S. device for later use. The Atmel memory chip AT29BV020 is a 2.7 volt in-system Flash Programmable and Erasable Read Only Memory (PEROM). The surface mount chip's availability in 2 megabit size with a PLCC (Plastic Leaded Chip Carrier) socket made it convenient for our senior project. The 2 megabits (256K x 8) of memory is organized as 262,144 words by 8 bits. Since it was very important for us to find a nonvolatile chip, we learned that our chip is manufactured with Atmel's advanced nonvolatile CMOS EEPROM technology, and the device offers access times to 120 ns, and a low 54 mW power dissipation. When the device is deselected, the CMOS standby current is less than 40 µA. The device endurance is such that any sector can typically be written to in excess of 10,000 times. The maximum sector program time is 20 msec and the Atmel chip does not require high input voltages for programming which met the requirement for our project. After reviewing all the component features we decided this chip would be the exceptional choice for our senior project.

Since we chose a microcontroller that has multiple Input/Output pins, it made it easy for us to interface the external memory chip to the microcontroller. However soldering the chip to our board was a challenging task, which needed to be planned and approached with caution since there was a lot of room for error. The main reason being that the P.L.C.C. socket for the memory chip was very small and all of the 31 pins needed to be soldered to the board under a magnifying lens. But to further decrease the chances of error, the pins were labeled so after soldering was done it was easy to verify if the pins were connected to the correct ports/pins on the microcontroller. We confirmed using the multi-meter that the soldering connections were done successfully. The pin connections from the microcontroller to the external memory chip are shown below in the table.

Atmel Memory Chip Pins	**Microcontroller Pins**
A0 – A17	Port B, Port F, PTG6 & PTG7
CE	Pin PTG3
OE	Pin PTG4

WE	Pin PTG5
I/O0 – I/O7	Port D
VCC	VDD
GND	GND
NC	Not Applicable

Table 2: Microcontroller & Atmel interfaced pins

To program a sector of 256 bytes of data the Atmel Flash memory provides a feature called Software Data Protection. This feature helps avoid the possibility of inadvertent write operations that can be caused by noise or by power-up and power-down sequences. Once this protection feature has been enabled, the chip requires that all subsequent write operations perform a series of "dummy" write operations before loading the chosen sector with data. The "dummy" writes consist of loading three known data values into three predefined addresses. This 3-byte sequence preceding a write operation virtually eliminates the chance of inadvertent write operations. The sequence is described below.

1. Load Data AA (hex) into Address 05555 (hex)
2. Load Data 55 into Address 02AAA
3. Load Data A0 into Address 05555
4. Load desired sector with data
5. Pause twc (device write cycle time)
6. Continue with next operation.

If this protection feature is enabled, any attempt to write to the device without the 3-byte command sequence will start a write cycle. However, no data will actually be written to the device and during this "write" cycle time (twc), valid data cannot be read from the Flash memory chip. The other main thing to keep in mind with the memory chip is that the timing for the read and write cycles needs to be followed exactly or errors can be introduced. There needs to be exact following of the timing diagrams of the memory chips data sheet. For the Read operation the AT29BV020 is accessed like an EPROM.

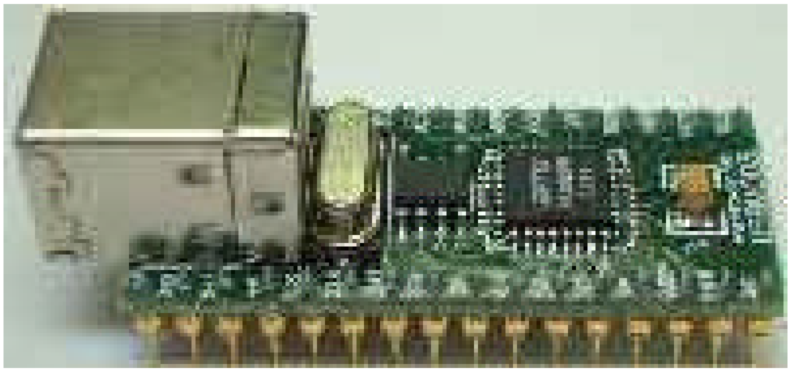
The main problem that we ran into was that once the program was written, it continued to generate invalid data on the last read. After additional testing we found that we needed to add a delay loop which gave the Flash memory chip enough time to write correct data. Another problem was that the debug interface also provided interference with the timing of the micro-controller.

Once we were successful in writing the desired data on the memory chip, now we needed some way to download the data from the memory chip onto a computer. One way to accomplish this task was by having an additional U.S.B. interface to the project to successfully download the recorded athlete data from the Atmel Flash memory chip to the computer.

There is one known problem in regards to the current configuration. The memory chip only comes in a surface mount chip. For our prototype, we placed the memory chip in a socket. This socket proved to be more than a little difficult to deal with. The J-clips that hold the memory chip in place have a tendency to bend and introduce errors in the data transferred to the memory chip. For the finished version of the MAMS system, the surface mount memory chip should be soldered directly to the board, and removed from the socket.

USB Serial Module Chip

The purpose of the USB chip interface to the project was to be able to successfully download the data from the external flash memory chip onto a computer. The USB interface was a challenging task in this project since the chip had to be interfaced with the microcontroller and the computer. After researching multiple USB module manufacturers, we found the USB MOD3 chip which is a second generation USB interface module using the FTDI FT232BM USB UART. The USB Serial Module was great for implementing a serial to USB interface. It handled many of the typical things in USB transfers, making it a fast solution for needed functionality; a built in oscillator for correct timing, port for connection, built in logic for interfacing with external circuits, and logic levels. It is almost a plug and play option. The only external hardware that was needed to make the chip functional was a bypass capacitor on the ground pin, and some pins wired together so that the computer would detect the device and provide the needed handshaking for transfer. The chip is shown below.



USB Interface With Microcontroller

The USB hardware interface is described in the following table.

USB MOD Chip Pins	Microcontroller Pins
VCCIO (+V) PCTL	VDD
RSTI	VCC
TXD	RXD
GND	GND

After connecting the pins to the microcontroller, the interface between the computer and the USB MOD chip needed to be designed.

USB Interface With Computer

There were a few things that needed to be taken under consideration for the interface between the USB MOD chip and the host PC. First, it made things simple when the drivers for the USB MOD chip were provided on the FDTI website for download. We downloaded and installed the virtual comport drivers in order for us to utilize simple comport commands. After the drivers were installed, the design decision needed to be made to figure out if the USB MOD chip needed to be self powered or bus powered. We opted to use the bus powered configuration, so we wouldn't have to figure out a way to provide a 5V power source. We also chose the bus powered configuration so that we wouldn't have to add more weight to the MAMS package, thus maintaining its non-invasive nature.

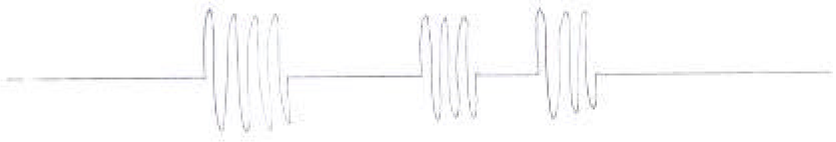
There were two programs that needed to be written in order to operate the USB MOD chip. The first program for the microcontroller would be simple, it would take data out of the external memory chip and put it on the transmit pin, SCI1 transmit pin of the microcontroller. The second program is embedded within the high level application. With our modified code using a virtual comport, it was very easy to access the port, and obtain the needed data.

Heart Monitor

The heart monitor component is comprised of the Polar heart monitor chest strap, antenna, and an amplifying circuit. The Polar heart monitor is a strap that goes around the athlete's chest. This Polar band normally transmits wirelessly to a watch that displays the heart rate; however, we removed the watch from the system since it would have been more difficult to extract the data from it than it would have been to capture the signal directly from the chest strap. To make this monitor work in our implementation for the MAMS system, we put a small wire wrapped antenna very close to the transmitter for the Polar heart monitor. We then ran long wires from the antenna back to an amplifier that cleaned up the signal so it could be processed by the microcontroller.

The antenna that we built was made of four short iron rods about two inches in length, wrapped with small gauge wire. This wasn't a great antenna, but by placing it very close to the Polar transmitter, we were able to pick up the transmitter's signal. In looking at the signal that we received from the Polar transmitter, it turned out to be a wave with 3 peaks for each heart beat, as shown below.

(Signal from the Polar Monitor Received by our Antenna)



After acquiring the heart rate signal we encountered two additional problems. First, every time there was a heart beat, the Polar monitor output a three peak sequence. To determine the heart rate of the athlete connected to the Polar monitor, we needed to determine the distance between the 3 peak sequences and then do some calculations on it. Since we didn't have access to the actual clock to count time, due to problems discussed in the microcontroller section, the distance between pulses was very difficult to determine. In the code for the heart monitor, found in appendix B we used a software hold to account for the wave fluctuations in each of the three peaks. We used polling to determine when we were in one of the peak sections. Because we were using polling it became critical for the clock to operate accurately so that we could skip over each of the peaks. The code created for the heart monitor accounts for about an 8.5 millisecond software hold, after sensing a high bit from our antenna. After obtaining the pulse width in milliseconds, we divided 60,000 by that time difference. The result from the equation would provide us with the athlete's heart beats/minute. The formula for the calculation was as follows: Athlete's Heart Rate (beats/minute) = 60,000/(distance between three pulse sequence in milliseconds).

The next problem was that the output signals from the heart rate monitor were only about +-200 mV. We needed at least at least 1.7 volts to register a high signal. To solve this problem, we used a negative feedback single rail op amp, the LM7131. We input the signal from the antenna in to the negative feedback op amp circuit. (Displayed in the MAMS schematic, at the end of this document) We also tied the op amp to only a single positive rail. As you can see from the wave diagram above, the peaks were both positive and negative. By tying the op amp to only a positive power supply, we amplified the negative part of the signal to achieve a peak of around 2 V. In other words, our negative feedback op amp circuit had a gain of 10, allowing our high signals to now register above the threshold on the microcontroller.

Tying the op amp to a single rail was an important part of making this component work. In our initial setup of the op amp circuit, we tried to amplify all of the peaks from the above wave diagram. However, with the negative and positive supplies, the circuit behaved erratically. One minute we would have a clean signal, and the next minute there would be so much noise in the system that the microcontroller couldn't read anything but highs on its input port for the heart monitor. This was the main problem that we had with the heart monitor. We solved this reoccurring problem by making sure that all grounds to all the different devices on our MAMS were connected to a common ground. The only other problem is that the time between peaks is determined by a fixed number of instructions. If the reassignment of pins on the micro-controller were to be taken care of, the actual clock cycles between the peaks could be more accurately calculated; thus increasing the accuracy of the heart rate.

G.P.S.

The G.P.S. component was implemented with a Garmin Geko 201. This is a fairly small, extremely light piece of equipment, which runs off its own power supply. Weighing around 3.1 ounces, with batteries installed the Geko made it an ideal candidate for allowing us to keep our ideal weight at a minimum. The Geko allowed us to obtain accurate satellite G.P.S. transmissions and also had a serial connector for simple interfacing to the microcontroller.

In our initial design, we wanted to use the NMEA output for G.P.S. data. However, this proved to have some problems with amount of space required and how quickly the Geko provided data over the interface to the microcontroller. To our good fortune, Garmin provides other interfaces for the G.P.S. data. They all have particular output sequences as shown in the data sheet for the Geko 201, which can be found on Garmin's web site at http://www.garmin.com. However, the main problem with these output sentences is that they were variable in length. There is one output sentence that Garmin provides that is called a simple text output. The simple text output sequence is a fixed length of 58 bytes. This output provided all the necessary information that we needed, global position, velocity, altitude (altitude is off by a significant amount, in the future an altimeter should be implemented to acquire more accurate data on this piece of data) time, and date. With the data being a fixed length, this proved to be the easiest output to parse the data.

The other major thing that we needed to change for the Geko to work with our MAMS system was that the interface from the Geko was set up for a RS232 interface, with +-5V. Our microcontroller only works for 0V to +3V, TTL levels. Because of this difference in voltage, we had to add to our initial implementation of the "provide G.P.S. Data" interface in the block diagram from page 3. In the initial implementation we just attached an inverter on the signal coming from the Geko, and put the rails for the inverter at TTL levels. However, after getting this to work with our system, we realized that the microcontroller's development board had the exact same circuit, wired through one of its serial ports. To take advantage of this already being implemented on the board we just had to attach the cables from the Geko to the appropriate pins on the development board, gnd being pin 5, and rxd being pin 3.

These few changes to the initial implementation, proved to be very beneficial in the final implementation, as there was a lack of breadboard space to attach the necessary pieces of the initial circuit. In the final implementation of the G.P.S. component, it is important that a Garmin G.P.S. device be used, with simple text interface enabled, ground tied to pin5 of SCI2, and the Garmin's data out cable attached to pin3 of SCI2. The other difference in the final implementation of the G.P.S. component is that we needed to slightly modify the setting for the baud rate. How these changes are discussed in detail in chapter 11 of the MC9S08GB60 data sheet. This modification was needed due to the problems in the variance of the clock, as already discussed in the microcontroller section.

If the clock problem spoken of in the microcontroller section of this document is corrected, and a more accurate clock is obtained, then the settings for the SCI on the microcontroller will need to be changed in order for the G.P.S. component to work properly (adjusting the baud rate).

The only other problem with the final implementation of our G.P.S. component is that it is currently working only for Garmin's proprietary simple text interface. However, the SCI for the microcontroller is set up at the standard for NMEA baud rates of 4800. This would enable future versions of the component to only change what data is obtained by the code in the microcontroller. So this problem is easily fixed by knowing the output of the data obtained from any particular G.P.S. acquisition device. The idea was to avoid proprietary systems in this component, but with time constraints we limited it to only Garmin products.

There is one major problem to avoid when obtaining information from the Garmin units. If one configures the clock of the microcontroller to run at 8 MHZ, you DO NOT have enough time to parse the incoming sentences. The entire sentence must be read in before any parsing can begin. Another problem that has already been referenced in other components is that you must be certain to have a common ground from the G.P.S. component and the microcontroller.

Upon completion of this project, the G.P.S. component is what is providing most of the data on the Athlete. In order for this to be a true M.A.M.S. system, there needs to be more accurate information obtained from other sources, to be added later, and connected to the Athlete.

Transceiver

After collecting the appropriate data from the athlete, the microcontroller was ready to send the information back to the base station. This is actually sent about every nine seconds. The data is sent out the microcontroller's PTE0 serial pin which connects to the transceiver's RX pin. A small amount of code was developed in CodeWarrior to read the data stored in the micro-controller's memory and write it out to this port.

To relay information from the athlete to the base station we used Aerocomm's AC4490-1000 transceiver component. It has line-of-sight transmission up to 20 miles and a no line-of-sight transmission up to four miles. Since the athlete would not be heavily obstructed at most points during the race this would work well. The transceiver accepts TTL signals from the microcontroller and sends them via the 900 MHz license free bandwidth back to the base station. We set all our devices to communicate at the 4800 baud rate, since data was not being sent very often this allowed for a more accurate transmission. While the base station transceiver uses a Software Development Kit (S.D.K.) board , the athlete's transceiver was soldered to a small pin adapter so it could be placed on a breadboard. It's set to transmit 8 bit mini packets with a stop bit and no parity at half duplex. The transceiver is set to transmit only when its buffer reaches 60 bytes but if a packet times out it will split the packet up and the packet will be reconstructed later in the C# software on the laptop. The destination of the athlete's transceiver is set to the MAC address of the base station transceiver instead of broadcasting. At the base station the SDK board takes the TTL signal received from the transceiver on the athlete and converts it to RS-232, a logic the computer can understand.

The AC 4490-1000 transceiver requires 1300 mA to operate while sending a packet. Although sending a packet only requires 10 ms at this amperage every five seconds, a race may last up to four hours and a substantial power source is required to ensure accurate and viable information. A NTE 1929 voltage regulator with a 12 volt alkaline battery pack (eight 1.5 volt AA batteries) allows us this endurable power with ample mAH insurance.

The SDK board interfaces to the computer via the COM or serial port. C# code was developed to open and monitor this port on the computer for any incoming packets.

Each packet is a fixed length of 60 bytes comprised of 23 variables, 20 of which are used. Each variable is in a known sequence within this packet. The packet is just a byte array with 60 indices, by using the known order and size of each variable we are able to separate them by using a large if/else block within a loop controlled by packet size i.e. 60. Once a packet has been divided into its appropriate variable destination it is then passed to the user interface database. The database, developed by Mike Pedersen, uses this information to display all relevant information of each athlete. This code runs in a thread in the background of the main C# user interface and is terminated by a closing packet.

Database

In order to properly store all the data that was being saved on the athlete's pack we needed to make a database. We decided to use Microsoft SQL Server 2000 to accomplish this. The database uses many different tables to store all of the needed data. Microsoft SQL Server 2000 provided us all the components that we needed to properly store all the data that was being collected. The data is saved using stored procedures that are part of Microsoft SQL Server 2000. The stored procedures are used to perform several different tasks that were required in order to complete this project, the code for each of the stored procedures can be found on the source code compact disc.

The database has a login table to provide access to the MAMS application. One table keeps track of all the athletes that are on a certain team and second table stores all of the tracks on which the team practices. Once the athlete has started a practice session, the data for that session will be kept in three different tables. The first one is an entry for the track, athlete, and course that the practice run is occurring. The second table keeps track of all real-time data as it is received from the base station receiver. The third and final table stores all of the data collected after the practice session has been completed, when it is downloaded from the external flash memory on the athlete unit.

There were really no problems with the database it was pretty straight forward from the beginning. In the future the only change that may be needed is to the data types of some of the tables' columns. Scalability should not be an issue for the database as it was designed to work with a large number of athletes in mind. Currently the only problem concerning the database is that it must be run on a windows machine since we are using Microsoft SQL Server 2000.

High Level Code

The high level portion of the project is where all the data collected from the athlete is displayed. The code was implemented using Microsoft Visual Studio .Net 2003, and was written using C#. We wanted to be able to display all the data that we collected in a way that would be understandable for the coach or trainer. We therefore have provided them with an interface that displays all of the real-time data in an easy to read format. It also has reporting capabilities to take full advantage of all of the data that is stored in the external flash memory. The program is updatable so new athletes, tracks/courses, and logins may be added.

In our initial design we intended to give the user (coach or trainer) a lot more capabilities with reporting on the athlete after the practice session was done. Because of timing problems with getting everything finished at the end of the semester we had to scale back much of this. The final product that we created provides slightly more options for the user upon downloading the full data after the race.

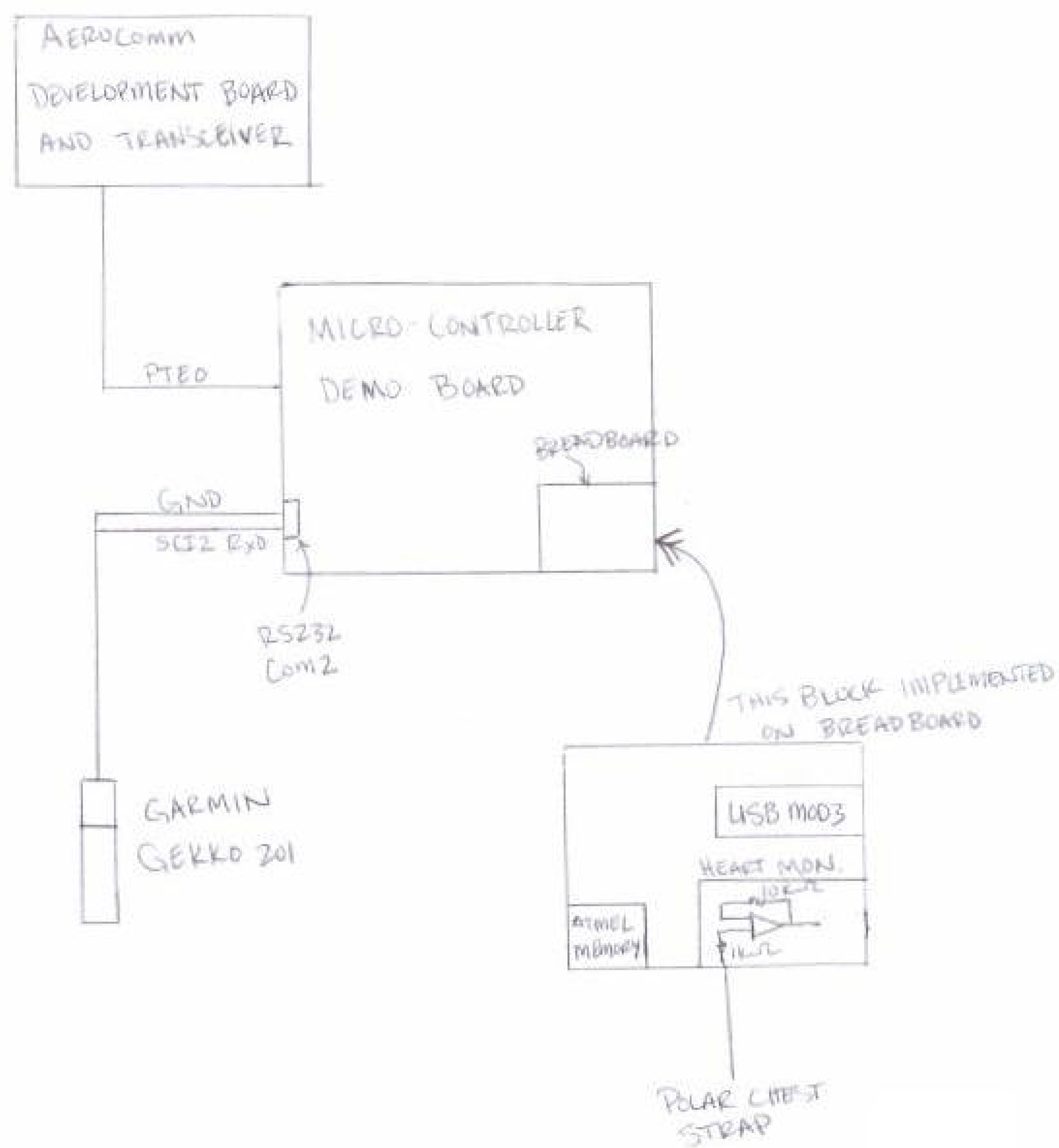
The real-time data was one of the major components of this project, we needed to be able to update the information for the coach as the practice session was happening quickly and accurately. We used a timer to access the database to see if new real-time data was available; if there was the program updates with the new information so that it can quickly be displayed to the screen, if there is no new data from the athlete then the current real-time data stays on the screen. Once the practice session has been completed the coach simply tells the program that the session has ended. Once the M.A.M.S. is retrieved from the athlete one may download the stored data which provides a more accurate and complete set of information collected during the practice session.

Downloading the data from the external memory is done in a very similar fashion to how we get data for the real-time statistics. Once the practice session has ended the coach or trainer simply needs to ground pin PTC7 on the microcontroller board in order to start the download from the athlete's external flash memory. Once PTC7 has been grounded the high level software spawns off a thread to receive the data through the serial interface discussed earlier in this document. Just like receiving real-time data we receive final data downloads in 60 byte packets that we parse to get the data into the database. Using a stored procedure the data is inserted into the database so that final reporting can be done on the practice session.

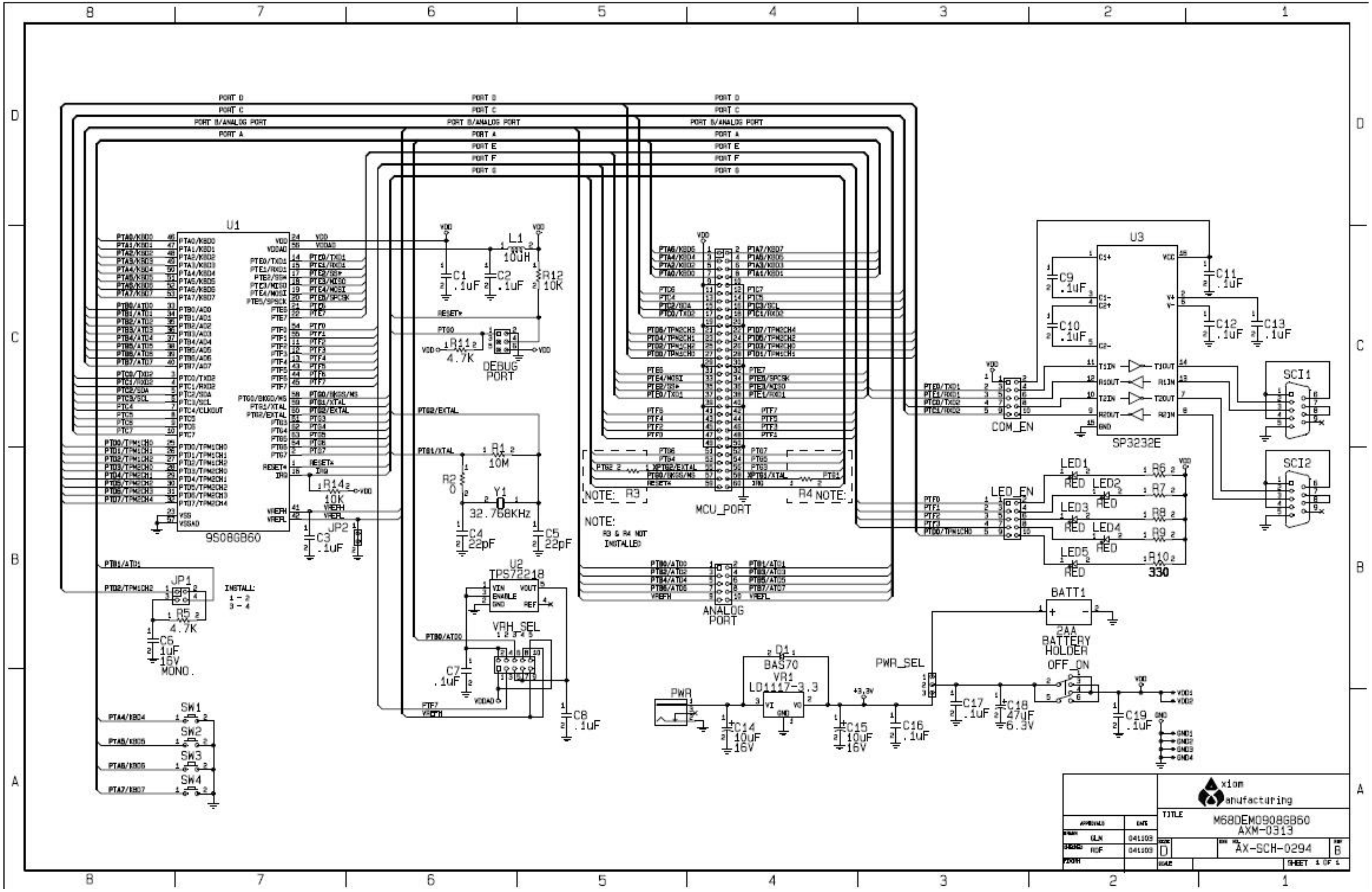
The biggest problem we encountered during the process of writing the high level interface was working with the serial interface. It is hard to debug the serial interface to make sure that everything is working the way that you intended it to. Although breakpoints are the best way to debug code, it distorts the timing for the serial interface and, from our experience, can produce unknown results. Once we were able to circumvent this anomaly we didn't encounter any other major problems while writing the high level interface.

This project was supposed to be designed to work with up to ten athletes without any problems. However, due to time constraints we were unable to develop an application that will work with up to ten athletes. The code currently works for one athlete and would work with as many as are desired if the code is expanded. All code for the high level application can be found on the source code compact disc.

M.A.M.S. Block Diagram/Schematic



Appendix A


PORT D
PORT C
PORT B/ANALOG PORT
PORT A
PORT E
PORT F
PORT G
U1
9S08GB60
VDD
L1
10UH
C1
.1uF
C2
.1uF
R12
10K
RESET*
R11
4.7K
DEBUG PORT
PTG2/EXTAL
PTG1/XTAL
R1
10M
R2
0
Y1
32.768KHz
C4
22pF
C5
22pF
R14
10K
C3
.1uF
JP2
MCU_PORT
NOTE:
R3
R4
R3 & R4 NOT INSTALLED
ANALOG PORT
U2
TPS72218
VIN
ENABLE
GND
VOUT
REF
VRH_SEL
C7
.1uF
C8
.1uF
VDDAD
JP1
INSTALL:
1 - 2
3 - 4
R5
4.7K
C6
1uF
16V
MONO.
SW1
SW2
SW3
SW4
PTA4/KBD4
PTA5/KBD5
PTA6/KBD6
PTA7/KBD7
U3
SP3232E
C9
.1uF
C10
.1uF
C11
.1uF
C12
.1uF
C13
.1uF
T1IN
T1OUT
R1OUT
R1IN
T2IN
T2OUT
R2OUT
R2IN
COM_EN
SCI1
SCI2
LED_EN
LED1
LED2
LED3
LED4
LED5
RED
R6
R7
R8
R9
R10
330
BATT1
2AA
BATTERY
HOLDER
OFF_ON
PWR_SEL
D1
BAS70
VR1
LD1117-3.3
PWR
+3.3V
C14
10uF
16V
C15
10uF
16V
C16
.1uF
C17
.1uF
C18
47uF
6.3V
C19
.1uF
VDD1
VDD2
GND
GND1
GND2
GND3
GND4
Axiom Manufacturing
TITLE
M68DEMO908GB60
AXM-0313
APPROVALS
DATE
DRAWN
GLN
041103
CHECKED
RDF
041103
FINISH
SIZE
D
DWG NO.
AX-SCH-0294
REV
B
SCALE
SHEET 1 OF 1
8
7
6
5
4
3
2
1
D
C
B
A

Appendix B (Micro-Controller Code, Please refer to source code for complete implementations)

****

// Initialization routine that takes care of the storage buffer, SCI baud rates, pull up pins, and all other needed initializations for the various components.
It takes no parameters.

void initAll()

****

// Handles the writing sequence for interfacing with the Atmel memory chip. Also writes 240 byte chunks of data. Does all sequencing etc. to handle a successful write to memory.
Function takes no parameters.

void memoryHandler()

****

// Helper function to the WriteMemory function. Sets up the writing sequence for the Atmel memory chip, or a read sequence depending on passed parameter.
Function takes as an input an int, for writes pass a 0, for reads pass a 1.

void SetSectorAddressPins (int ReadWrite)

****

// Another helper function to the WriteMemory function. This function sets everything up for a write sequence, including the three byte sequence code. It also configures the control pins for initial entry in to a write sequence.
Function takes no parameters.

void WriteMemoryProgrammingSequence()

****

// The main function to handle the writing of blocks of memory for the Atmel chip. It utilizes the the SetSectorAddressPins, and WriteMemoryProgrammingSequence functions to handle this task. The memory accepts writes in 256 byte chunks. As data from the athlete is obtained the loopVariable just needs to change to transfer more 8 bit chunks of data. The function loops through the input number of times to transfer the input number of bytes to memory.
Function takes loopVariable as an input, the int determines how many of the bytes of data to transfer to the memory.

void WriteMemory(int loopVariable)

****

// Function that reads data back from the Atmel Memory chip. Also utilizes the SetSectorAddressPins function.
Function takes an int as a parameter, that int determines how many bytes of data to be read from the current block of memory.

void ReadMemory(int loopVariable)

****

// Function that takes current athlete data and provides that data to the transceiver for real time transmission, via the SCI1 txd pin.
Function takes no parameters.

void Transmit()
****

// Function that handles the capture of GPS data from a Garmin enabled with simple text output. The function listens for incoming data on the SCI2 rxd pin. It stores this data for real time transmission, and storage in the Atmel memory.
Function takes no parameters.

void GPSHandler()
****

// Function that counts the the peaks detected from the Antenna on pin PTE7. It counts the time between four consecutive pulses from the polar chest strap. Saves the number of clock cycles spent; providing for the milliseconds for the heart rate calculation at the MAMS base station.
Function takes no parameters.

void HeartHandler()
****

// Function to handle the final download of all stored data from the MAMS system. This function waits for the grounding of PTCD7 to start the retrieval of data from the Atmel memory, and transfer of the data out the SCI1 txd pin. This SCI1 pin is attached to the USBMod3 chip that provides the high speed download. The MAMS system waits in this function after the completion of an Athlete's session. The end of session is determined by grounding PTC6, thus entering this function to await final download.
Function takes no parameters.

void Download()