EE 472 Lab 2
Learning the Development Environment

Jonathan Ellington
Patrick Ma
Jarrett Gaddy

Contents

List of Tables

List of Figures

1 ABSTRACT

In this lab the students are to take on the role of an embedded system design team. They will design an exciting new medical instrument to monitor various patient metrics. When the device finds metrics are out of the acceptable range, the user will be notified, thus saving them from potential health risks. The students must first layout the design for their system using various design tools, then they must implement the system in software. Finally the students must test their system to make sure that it is ready to start saving lives.

2 INTRODUCTION

The students are to design an embedded system on the Texas Instruments Stellaris EKI-LM3S8962 and EE 472 embedded design testboard. The design must implement a medical monitoring device. This device must monitor a patient's temperature, heart rate, and blood pressure, as well as its own battery state. The design must indicate when a monitored value is outside of a specified range by flashing an LED on the test board. When a value deviates even further from the valid range an alarm will sound. This alarm will sound until the values return to the valid range or the user acknowledges the alarm with a button. The values of each measurement will also be printed to the oled screen.

The design will be tested to verify proper behavior on alarm and warning notifications. In addition the implementation will be tested by measuring the amount of time that each of the 5 program tasks running the instrument take to execute. These tasks are mini programs that each handle a part of the instruments purpose.

3 DISCUSSION OF THE LAB

3.1 Design Specification

3.1.1 Specification Overview

The entire system must satisfy several lofty objectives. The final product must be portable, lightweight, and Internet enabled. The system must also make measurements of vital bodily functions, perform simple computations, provide data logging functionality, and indicate when measured vitals exceed given ranges, or the user fails to comply with a prescribed logging regimen.

At the present time, only two subsystems must be produced: the display and alarm portions. Additionally, the system must demonstrate the ability to store basic measurements.

The initial functional requirements for the system are:

- Provide continuous sensor monitoring capability
- Produce a visual display of the sensor values
- Accept variety of input data types
- Provide visual indication of warning states
- Provide an audible indicator of alarm states

3.1.2 Detailed Specifications

For this project, these requirements have been further specified as follows:

The system must have the following inputs:

- Alarm acknowledgment capability using a pushbutton
- Sensor measurement input capability consisting of:
  - ∗ Body temperature measurement
  - ∗ Pulse rate measurement
  - ∗ Systolic blood pressure measurement
  - ∗ Diastolic blood pressure measurement

The system must have the following outputs:

- Visual display of the following data in human-readable formats:
  - ∗ Body temperature
  - ∗ Pulse rate
  - ∗ Systolic blood pressure
  - ∗ Diastolic blood pressure
  - ∗ Battery status
- Visually indicate warning state with a flashing LED
- Visually indicate a low battery state with an LED
- Audibly indicate an alarm state using a speaker

The initialization values, normal measurement ranges, displayed units, and warning and alarm behaviors for each vital measurement are given in Table 1. The sensors must be sampled every five seconds.

Measurement	Units	Initial Value	Min. Value	Max. Value	Warning Flash Period
Body Temperature	C	75	36.1C	37.8C	1 sec
Systolic BP	mm Hg	80	-	120 mmHg	0.5 sec
Diastolic BP	mm Hg	80	-	80mmHg	0.5 sec
Pulse Rate	BPM	50	60 BPM	100 BPM	2 sec
Remaining Battery	%	200	40 %	-	Constant

Table 1: Specifications for measurement data

A measurement enters a warning state when its value falls outside the stated normal range by 5%. An alarm state occurs when any measurement falls outside its stated normal range by 10%.

Additionally, the system must be implemented using the Stellaris EKI-LM3S8962 ARM Cortex-M3 microcomputer board, The software for the system must be written in C using the IAR Systems Embedded Workbench/Assembler IDE.

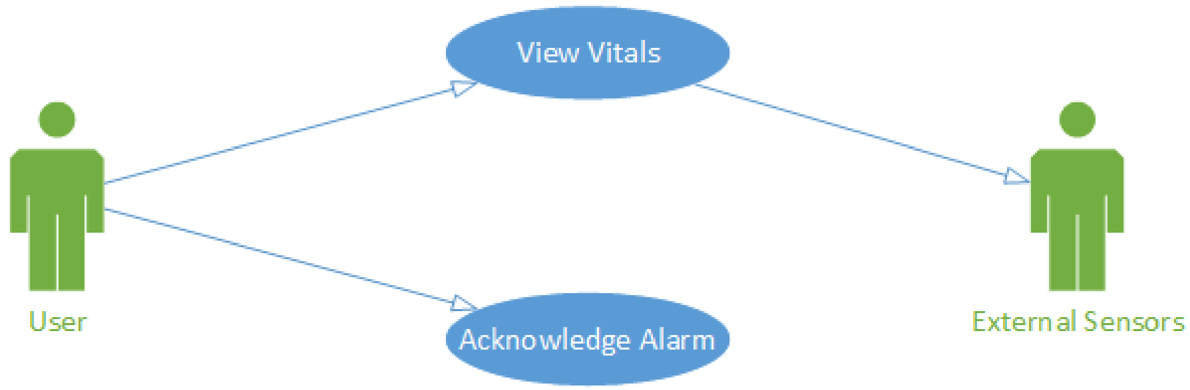

Figure 1: Use case diagram

3.1.3 Identified Use Cases

Taking the functional requirements listed above, several use cases were developed. A Use case diagram of these scenarios is given in Figure 1. Each use case is expanded and explained below.

Use Case #1: View Vital Measurements
In the first use case, the user views the basic measurements picked up by the sensors connected to the device.
During normal operation, once the device is turned on by the user, the system records the value output by each sensor. This raw value is linearized and converted into a human-readable form. Finally, this value is displayed on-screen.

Three exceptional conditions were identified for this use case:

- *One or more of the expected sensors is not connected* - If this occurs, the measurements taken by the device may be erratic. At the present moment, no action will be taken in such events. Later revisions may address the issue
- *A measured value is outside 5% of the specified normal range* - In this case, a warning signal will flash as an indication of the warning condition
- *A measured value falls outside 10% of a specified "normal" range* - In this case, an audible alarm will sound to indicate the alarm condition

Use Case #2: Acknowledge Alarm
In the second case, the system is in an alarm state. The user acknowledges the alarm condition by pressing a button.

Upon pressing the button, the system silences the audible alarm. Any visual warnings continue to flash during the silenced period. If a specified amount time passes and the sensor reading(s) continue to maintain an alarmed state, the audible alarm will recommence.

No exceptional conditions were identified for this use case.

3.2 Software Implementation

A top-down design approach was used to develop the system. First, a functional decomposition of the problem was carried out based on the identified use cases. Next, the system architecture was developed. After understanding the system architecture, the high-level project file structure in C was defined, followed by the low-level implementation of the tasks.

3.2.1 Functional Decomposition

After understanding how the user would interact with the device, the system was functionally decomposed into high-level blocks as shown in Figure 2. The main system control is located in the CPU, which controls all data flow into and out of the peripheral devices. The OLED displays the user's current vitals including blood pressure (systolic and diastolic), temperature, and pulse rate. In the future external sensors will be added, but for now the values are simulated using the CPU. The CPU also controls three LEDs colored green, yellow, and red. These LEDs are used to inform the user on the current state of their vitals as well as the state of the device. Under normal circumstances, the green LED will be lit. If the users' vitals fall outside of a specified range, the red LED will flash at a specified rate, depending on which vital is out of range. If the battery is low, the yellow LED will be illuminated.

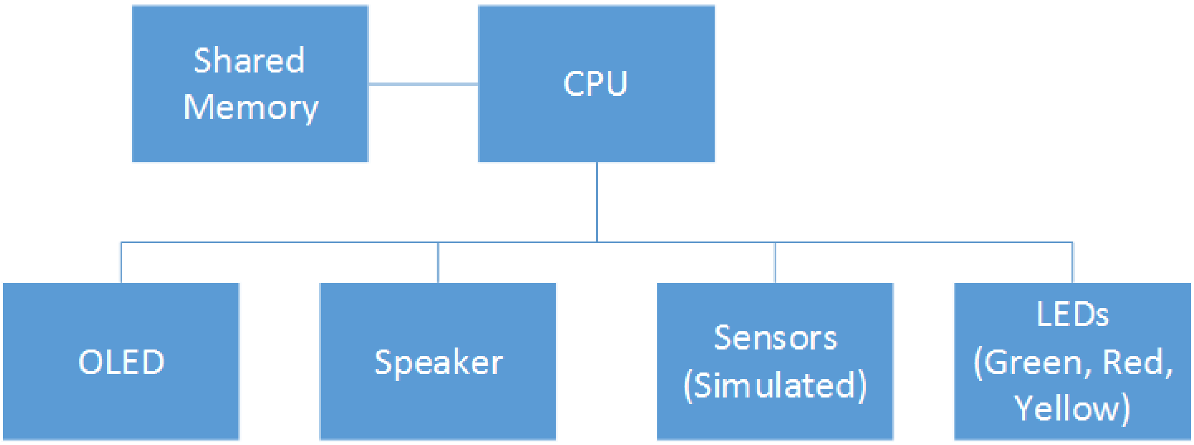

Figure 2: Functional Decomposition

3.2.2 System Architecture

Next, the system architecture was developed (Figure 3). At a high level the system works on two main concepts, the scheduler and tasks. Tasks embody some sort of work being done, and the scheduler is in charge of determining the speed and order in which the tasks execute. The system has several tasks, each with their own specific job. For modularity reasons, each task should have the same public interface and the scheduler should be able to run each task regardless of that specific tasks job or implementation. Thus the task concept is abstracted into a Task Control Block (TCB), and the scheduler maintains a queue of TCBs to run. The TCB abstraction is shown in Figure 3 using inheritance, and the fact that the scheduler has a queue of TCBs is shown with composition. The core functionality of the system was divided into the following five main tasks:

- **Measure Task** - In charge of interacting with the blood pressure, temperature, and pulse sensors (simulated)
- **Compute Task** - Converts sensor data into human readable format
- **Display Task** - Displays the measurements on the Stellaris OLED

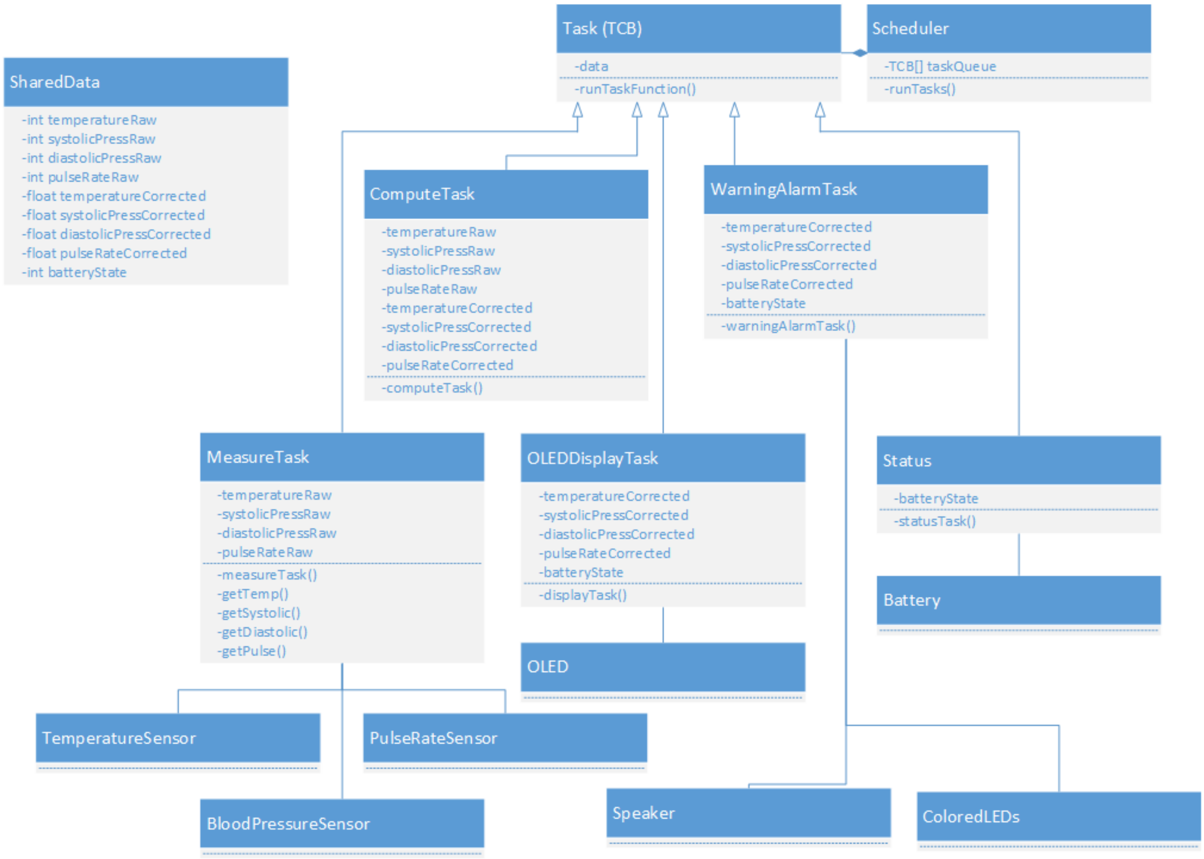

Figure 3: System Architecture Diagram

- **Warning/Alarm Task** - Interacts with the red, yellow, and green LEDs, as well as the speaker to annunciate warning and alarm information
- **Status Task** - Receives battery information from the device

Each of these tasks interact using the shared data shown in Figure 3.

3.2.3 High-level Implementation in C

After developing the system architecture, the design needed to be translated into the C programming language. The design manifested in a multi-file program consisting of the following source files:

- **globals.c/globals.h** - Used to define the Shared Data used among the tasks
- **schedule.c/schedule.h** - Defines the scheduler interface and it's implementation, as well as the TCB structure
- **timebase.h** - Defines the timebase used for the scheduler and tasks

Each task also has it's corresponding ".c" and ".h" file (for example, measure.c and measure.h).

The TCB structure that the scheduler uses must work for all tasks, and must not contain any task-specific information. Instead, the TCB consists of only a void pointer to the tasks data, and a pointer to a function that returns void and takes a void pointer, as follows:

```c
struct TCB {
  void *taskDataPtr;
  void (*taskRunFn)(void *);
}
```

Leaving out the type information allows the scheduler to pass the task's data (*taskDataPtr) into the task's run function completely unaware of the kind of data the task uses or how the task works.

For increased modularity, the data structure used by each task was not put in the task's header file. Instead, the structure was declared within the task implementation file, and instantiated using a task initialization function. In the header file, a void pointer pointing to the initialized structure is exposed with global scope, as well as the task's run and initialization functions.

3.2.4 Task Implementation

The primary task of this project is to implement C code for a medical device on the Stellaris EKI-LM3S8962 and its ARM Coretex A3 processor. The project was started by creating a main file that initializes the variables used in each task then runs into an infinite while loop. Inside the while loop a run method is called. The run method is part of the scheduler. This method keeps track of whether the device is on a minor cycle or a major cycle and runs the preform task method of each task. The tasks included in this project are Compute, Measure, Warning, oledDisplay, and status. Each task has a public interface of 2 void pointers. One that when initialized by the main method will point to the preform task function, and another that, when initialized, points to a struct containing pointers to the data required by that task. Each task has a task control block(TCB) in the scheduler. This TCB points contains pointers to the preform task function and the data for the task. The TCB is used by the scheduler to run the task. The scheduler contains an array of TCB, each element in the array corresponds to a new task. In this case there are 5 tasks so 5 elements in the array.The scheduler's run task contains a for loop that runs through the array of TCB and runs the function pointed to by the TCB with the argument of the data pointer stored in the TCB. After running all 5 tasks the TCB has a software defined wait of 250 milliseconds to implement the minor cycle delay. The software defined wait is simply a for loop that uses addition as a time consumption tool. It is known that system clock is 8 MHz so having a for loop go from 0 to 2 million would take about a quarter of a second. The control flow is shown in Figure 4.

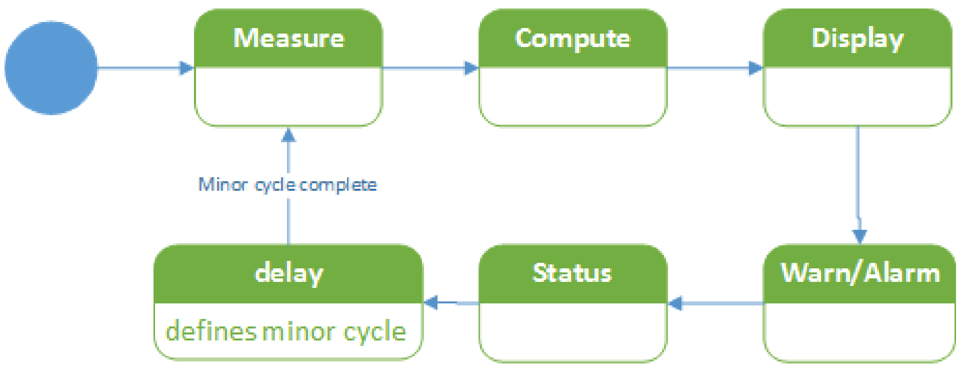

Figure 4: Control Flow Diagram

Each task has its own unique purpose in the system, and each uses a different part of the global data. The Measure task (Figure 5 in the appendix), deals only with the raw data from the instruments. This task is meant to act in place of the instruments that are unavailable. The task only runs if the scheduler has set the global value is Major Cycle to 1. On a major cycle the measure function either increments the data of each measurement by 1 or 2 or decrements the data by 1 or 2.

The compute task is very simple. Like the measure task it only runs on a minor cycle. It takes the raw data that has been set by the measure task, multiplies by a constant and adds a constant to each piece of raw data to get the corrected data. The compute task then puts the values for the corrected data in memory at the location of the global data pointer. Compute uses every data value except the battery.

After compute, the warning task begins checking for warning or alarm states. The warning task only deals with the corrected data from compute and the battery state. This task also must deal with the input and output signals used to display warning and sound an alarm. Unlike the other tasks, this task has more to its initialization than just initializing the data. In addition to setting up the pointers to the global data, during initialization, the task also enables peripheral banks C, E, F, and G. These are enabled using the SysCtlPeripheralEnable library call. Additionally the task set up pins C 5, 6, and 7 as outputs, pins F 0 and G 1 as PWM outputs, and pin E 0 as a pull up resistor input. Additionally the PWM outputs are set to use a 65 Hz clock to play a sound at this frequency whenever enabled. The activity diagram is shown in Figure 7 in the appendix. There are 3 subsystems in the warning task. These subsystems each handle a different part of user notification. The first subsystem deals with the alarm. The subsystem checks to see if any of the given corrected data is outside of the range given to us as acceptable by more than 10%. If any value is then the PWM output is enabled using PWMGenEnable. If the values fall back within the acceptable range, or the user hits the acknowledge button, then the PWM is disabled with PWMGenDisable and the sound stops. The next subsystem checks the corrected data for being 5% out of range of the accepted values. If any value is more than 5% out of its range then a warning will be displayed on the red led connected to pin C 5 using the GPIOPinWrite function. Depending on the value that is out of range the period the led flashes at will vary. The final subsystem is the battery check. This system checks if there is more than 30% of battery left on the device. This is taken from the battery state data field. If there is less than 20% battery left then a yellow led connected to pin C7 is illuminated, if there is more then 20% battery, and the device is not in a warning or alarm state then the green led on pin C 6 is illuminated.

To show a user their current medical measurements, the system also has an oledDisplay task. This task uses the corrected data from measurements, and the battery state. The display task has an activity diagram shown in Figure 8 in the appendix. This task uses the sprintf() function in C to convert the data types that the corrected data is stored in, and properly format these data values into a string which is stored in a buffer. The string contained in the buffer is then printed to the OLED screen using the driver library rit128x96x4 functions.

The last task is the status task. This task only deals with one piece of data which is the battery state data. The only thing the status task does is that on a major cycle, it decrements the battery state by 1. This is shown in the activity diagram in Figure 9 in the appendix.

4 PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS

4.1 Results

The project was completed and demonstrated on January 29, 2014.

Demonstration of the system to the interested parties showed that the system met the requirements initially presented at the onset of the lab project. Testing of the system prior to demonstration also verified that the system met the specifications listed in Section 3.1.

Additionally, during the demonstration, to show that several warning features worked as expected, source code was temporarily modified to speed up the progression of the system through various warning states. These changes were simple to execute and caused the desired effect on the system without causing unwanted aberrant behavior. Reverting the source code was similarly intuitive.

During the actual coding and implementation of the design, remarks were made several times about the ease of execution during that phase of the project. After the initial high level Design phase, very few changes were made, or required, to the functional design or system architecture.

Using an oscilloscope, the run times of each task were empirically determined. The results are given in Table 2.

Task	Runtime (μs)
Measure	23.4
Compute	55.4
Display	22900.0
Warning	27.4
Status	5.6

Table 2: Empirically determined task runtimes

Answers to the last three questions in the list of items to include in the project report:
You don't find the stealth submarine. That's why they are so expensive; at that cost, you take great pains to never lose one.
A helium balloon always rises. It just rises upside-down.
If you really managed to lose the stealth submersible, you first have to tell the government, which will deny it has any stealth submersibles, then you have to comb the seven seas until your comb hits the sub.

4.2 Discussion of Results

The ease of change in the code is the result of a large amount of time spent on design. The design makes it easy to configure flash times, add new tasks, and to reason about tasks independently of the whole system. The solid high-level architectural design led to ease of implementation and change.

In terms of performance, the run times of each task appear to correspond with the number of instructions required for each task. Given the speed of the CPU, 8 MHz, we can calculate an estimated number of instructions for each task. This is given in Table 3. The majority of the cycles are likely spent waiting for memory. For example, the status task only has two comparisons and an arithmetic operation, but has to reference the data in global memory. The exception here is the display task, which was about three orders of magnitude more instructions

Task	Instructions
Measure	187
Compute	443
Display	183,200
Warning	219
Status	45

Table 3: Estimated instructions per task, rounded to the nearest instruction

than the other tasks. This was due to the sprintf() library call, included in the standard C library. While this could have been optimized, it was found that with a minor cycle delay of 250 ms, the display delay of 22.9 ms was not significant.

4.3 Analysis of Any Errors

The project was completed without any residual errors or unsolved problems. See the following section for analysis of issues encountered while working on the project itself.

4.4 Analysis of Implementation Issues and Workarounds

The medical instrument design in this project was completed and tested successfully to meet all the requirements, the designers did face a number of errors and difficulties in the process. Most of the challenges faced in this design were in implementing the inputs and outputs from the ARM micro processor. In this implementation, the students originally consulted the driver library documentation and found how to enable a general purpose input output (GPIO) pin, and how to read from a GPIO pin. However, when the code to enable a GPIO pin was executed, the execution of tasks within the scheduler froze completely until a reset. This problem was caused by the students failure to enable the peripheral bank that the GPIO pins existed on. After learning how to do this, code was added to enable the peripheral and the task execution no longer froze on enabling a pin.

Another problem encountered by the design team was that initially after setting up a GPIO pin as an input to take a button press from the Stellaris test board, pressing the button did not have the intended effect. The push button did not change the state of the input. The students were originally perplexed and tried a number of solutions to this problem. Testing indicated that the code for the GPIO input was correct as a 3.3 V signal directly to the pin could trigger the intended event on the input. Eventually it was found that to use the push button switch the GPIO pin's pad must be configured to accept a pull up resistor type of input.

All problems were solved before demonstrating the product to the interested parties. As such, the final design did not have any errors.

5 TEST PLAN

To ensure that this project meets the specifications listed in section 3.1, the following parts of the system must be tested:

- Vitals are measured and updated
- System properly displays corrected measurements and units properly

- System enters, indicates, and exits the proper warning state for blood pressure, temperature, pulse, and battery
- System enters and exits the alarm state correctly
- Alarm is silenced upon button push
- Alarm recommences sound after silencing if system remains in alarm state longer than silence period

Additional tests to determine the runtime of each specific task are also required.

5.1 Test Specification

Annotated description of what is to be tested and the test limits. This specification quantifies inputs, outputs, and constraints on the system. That is, it provides specific values for each.

Note, this does not specify test implementation...this is what to do, not how to do it.

5.1.1 Scheduler

The scheduler needs to be shown to correctly schedule and dispatch tasks. This means that task should execute in the right order, and at the right time. Given a minor cycle of 50 ms, every task should run roughly once every 50 ms.

5.1.2 Measure Task

For this design, no external sensors were used; instead they were simulated. Each simulated sensor should be tested and verified against the specification, as follow.

- **Temperature** The temperature should increase by two every even major cycle (5 seconds) and decrease by one ever odd major cycle until it exceeds 50, at which point the process should reverse (decrease by two every even major cycle and increase by one every odd major cycle), until it dips below 15, and the whole process should be started over again.
- **Pulse** The pulse requirements are very similar to measure, except the pulse should increase by three rather than two, and the range is between 15 and 40.
- **Systolic Pressure** The systolic pressure should increase by three every even major cycle and decreases by one every odd major cycle. If it exceeds 100, it should reset to an initial value.
- **Diastolic Pressure** The diastolic pressure should decrease by two on even major cycles and decrease by one on odd major cycles, until it drops below 40, when it should restart the process.

5.1.3 Compute Task

The compute task should be verified to convert raw simulated sensor data according to the following formulas.

- $CorrectedTemperature = 5 + 0.75 * RawTemperature$
- $CorrectedSystolicPressure = 9 + 2 * RawSystolicPressure$
- $CorrectedDiastolicPressure = 6 + 1.5 * RawTemperature$
- $CorrectedPulseRate = 8 + 3 * RawTemperature$

5.1.4 Display Task

The display task should be tested to print each corrected value properly on the screen, and update them as the computations are done.

5.1.5 Warning/Alarm Task

The warning/alarm system needs to be tested to do several things. When in a warning state, it should flash the red LED at the rate appropriate for the warning. When the battery is low, it should illuminate the yellow LED. If the system is in an alarm state, it should sound the speaker alarm. The following ranges in Table 4 are calculated from the specified minimum and maximums found in Table 1 on page 2.

Data	Warning Range	Alarm Range
Temperature	34.3 - 39.7 C	32.5 - 41.6 C
Systolic Pressure	$>$ 84 mmHg	$>$ 88 mmHg
Diastolic Pressure	$>$ 126 mmHg	$>$ 132 mmHg
Pulse	57 - 63 BPM	54 - 110 BPM

Table 4: Initial values and warning/alarm states

5.1.6 Status Task

Since the initial design does not use a battery, the status task simulates the battery state using the CPU. For now, it simply decrements the state of the battery. The test should show that the battery state is decremented by one every major cycle.

5.2 Test Cases

The students begin testing by examining if the alarm sounds at the proper time. This is initially tested by disabling the functions that simulate measurements being made on each of the data measurements, and setting their initial values to be either within the alarm range or outside of the alarm range. The warning states were also initially tested this way. The initial values for raw data given in Table 1 on page 2 were used to test the normal state of the machine because each falls within the acceptable range of measurements for corrected data (also given in Table 1) that does not require a warning.

Using these initial values, the code was programmed onto the Stellaris board. Correct operation was verified by the alarm not sounding, and the red led being off, indicating that no warning state was in effect. In addition the green led was on indicating a normal state. Next the students varied one parameter at a time to be outside of the acceptable range by more than 10%. Starting with the temperature being set to an initial raw value of 50, the alarm was verified by hearing the aural annunciation coming from the system. In addition, the temperature warning stat was also in effect. This means that the green led was off and the red led was blinking. To verify correct operation we needed to make sure the led was blinking with a period of 1 second. The correct flashing pattern was verified by counting the number of times the led flashed in 6 seconds. In this case, for temperature, the led flashed 6 times in 6 seconds indicating a 1 second period, and correct operation. After this test, the temperature value was returned to 42 and the Pulse was instead set to 45. The same methods were used to verify that the alarm and warning states for pulse rate were working correctly, but this time the warning led turned on 3 times in 6 seconds indicating a 2 second period which is the intended period of flashing. The pulse rate was then returned to 25 and each pressure reading was checked for correct operation individually by being set to an initial raw value of 100. Once again, the green led started off because the system was not in a normal state. The alarm was sounding due to the extremely high blood pressure measurements, and the red warning led flashed 12 times in 6 seconds indicating the correct period of .5 seconds for a blood pressure warning. In addition to testing the validity of each warning state and alarm state, the acknowledgement of the alarm was also tested during each of these tests. This was tested by hitting the acknowledge button once during each measurements test. During each test, hitting the acknowledge button turned the alarm sound off for a short time, as intended.

Next the measurement simulation functions were tested. This was done by re-enabling each one that had been disabled from the previous test one at a time. The initial raw values were again set to the values in Figure 5. When each measurement was re-enabled, the students could watch the temperature change at each major cycle using the OLED display. Since the OLED display indicated that the corrected temperature went up .75 degrees on a major cycle then down 1.5 degrees on the next, the temperature measurement was working as intended. This situation also gave the students an opportunity to verify that the warning and alarm states initiated as the temperature fell out of the acceptable range. The Led began flashing with a 1 second period after a few major cycles, then the alarm began sounding, indicating correct operation. Since temperature was working correctly, the temperature measurement function was once again disabled and the pulse rate measurement function was re-enabled. The OLED display indicated that the pulse rate was increasing by 3 on a major cycle then falling by 6 on the following major cycle. This was consistent with the intended design. The alarm and warning being initiated as the pulse rate fell. The pulse rate measurement was then disabled and each blood pressure measurement was re-enabled individually for testing. The Systolic pressure began by rising 4 mm Hg on a major cycle then falling 2 mm Hg on the next, and the Diastolic pressure by rising 3 on a Major cycle and falling 1.5 on the next, this was consistent with our design. The warning and alarm states were activated as each passed its threshold and the red led was blinking with a period of .5 seconds. The warning led was also tested in the case that all warning states were active. To do this all initial values were set to 100. In this case, as designed by the students, the red warning led indicated the fastest blinking warning with a .5 second period.

The final bit of testing preformed on the system was timing each task within the system. This was done by adding a general purpose output pin in our scheduler code. This output was set high right before the execution of a task, and set low immediately after the execution of the task. An oscilloscope was then attached to this output pin and set to trigger on a positive edge. The cursors were then used to measure the amount of time the signal was high in each cycle.

Using these initial values, the code was programmed onto the Stellaris board. Correct operation was verified by the alarm not sounding, and the red led being off, indicating that no warning state was in effect. In addition the green led was on indicating a normal state. Next the students varied one parameter at a time to be outside of the acceptable range by more than 10%. Starting with the temperature being set to an initial raw value of 50, the alarm was verified by hearing the aural annunciation coming from the system. In addition, the temperature warning stat was also in effect. This means that the green led was off and the red led was blinking. To verify correct operation the students needed to make sure the led was blinking with a period of 1 second. The correct flashing pattern was verified by counting the number of times the led flashed in 6 seconds. In this case, for temperature, the led flashed 6 times in 6 seconds indicating a 1 second period, and correct operation. After this test, the temperature value was returned to 42 and the Pulse was instead set to 45. The same methods were used to verify that the alarm and warning states for pulse rate were working correctly, but this time the warning led turned on 3 times in 6 seconds indicating a 2 second period which is the intended period of flashing. The pulse rate was then returned to 25 and each pressure reading was checked for correct operation individually by being set to an initial raw value of 100. Once again, the green led started off because the system was not in a normal state. The alarm was sounding due to the extremely high blood pressure measurements, and the red warning led flashed 12 times in 6 seconds indicating the correct period of .5 seconds for a blood pressure warning. In addition to testing the validity of each warning state and alarm state, the acknowledgement of the alarm was also tested during each of these tests. This was tested by hitting the acknowledge button once during each measurements test. During each test, hitting the acknowledge button turned the alarm sound off for a short time, as intended.

Next the measurement simulation functions were tested. This was done by re-enabling each one that had been disabled from the previous test one at a time. The initial raw values were again set to the values in Figure 5. When each measurement was re-enabled, the students could watch the temperature change at each major cycle using the OLED display. Since the OLED display indicated that the corrected temperature went up .75 degrees on a major cycle then down 1.5 degrees on the next, the temperature measurement was working as intended. This situation also gave the students an opportunity to verify that the warning and alarm states initiated as the temperature fell out of the acceptable range. The Led began flashing with a 1 second period after a few major cycles, then the alarm began sounding, indicating correct operation. Since temperature was working correctly, the temperature measurement function was once again disabled and the pulse rate measurement function was re-enabled. The OLED display indicated that the pulse rate was increasing by 3 on a major cycle then falling by 6 on the following major cycle. This was consistent with the intended design. The alarm and warning being initiated as the pulse rate fell. The pulse rate measurement was then disabled and each blood pressure measurement was re-enabled individually for testing. The Systolic pressure began by rising 4 mm Hg on a major cycle then falling 2 mm Hg on the next, and the Diastolic pressure by rising 3 on a Major cycle and falling 1.5 on the next, this was consistent with the design. The warning and alarm states were activated as each passed its threshold and the red led was blinking with a period of .5 seconds. The warning led was also tested in the case that all warning states were active. To do this all initial values were set to 100. In this case, as designed by the students, the red warning led indicated the fastest blinking warning with a .5 second period.

The final bit of testing preformed on the system was timing each task within the system.

This was done by adding a general purpose output pin in the scheduler code. This output was set high right before the execution of a task, and set low immediately after the execution of the task. An oscilloscope was then attached to this output pin and set to trigger on a positive edge. The cursors were then used to measure the amount of time the signal was high in each cycle.

6 SUMMARY AND CONCLUSION

6.1 Final Summary

The students began creating their medical instrument through a rigorous design process at different levels of detail. The students then continued work by implementing their design in C code for the ARM Cortex A3 microprocessor. Next the code was tested and debugged using the IAR workbench debugging tool, as well as visual queues programmed into the design. Finally, after verifying that the system worked as specified, it was presented to the instructor.

6.2 Project Conclusions

This project contained 3 major phases, the design, implementation, and testing steps. The students were immediately introduced to using the unified modeling language(UML) to design embedded systems. This is the first time many students will have used UML for system design which caused some confusion and difficulty. In the end through the use of the UML guidelines for design, the students were able to implement their system in code for the Texas Instruments Stellaris EKI-LM3S8962 much more quickly and with far fewer errors than if they had spent less time in the design phase of this project.

Effective design tools allowed the students to quickly implement their embedded system in C code for an ARM Cortex A3 processor, and move onto the testing phase of the project quickly. Unfortunately, while testing the students encountered a number of problems in using the PWM and general purpose input and output signals. After consulting the documentation for the Stellaris kit and solving their input/output problems, they began testing their design using visual and audio queues, the IAR embedded workbench debugger, and a few specifically programmed debug features. After the results of the testing verified the design to be working correctly, the students proceeded to present their medical instrument to their instructor.

A BREAKDOWN OF LAB PERSON-HOURS (ESTIMATED)

Person	Design Hrs	Code Hrs	Test/Debug Hrs	Documentation Hrs
Patrick	15	7	2	9
Jarret	3	6	6	7
Jonathan	15	5	4	8

By initializing/signing above, I attest that I did in fact work the estimated number of hours stated. I also attest, under penalty of shame, that the work produced during the lab and contained herein is actually my own (as far as I know to be true). If special considerations or dispensations are due others or myself, I have indicated them below.

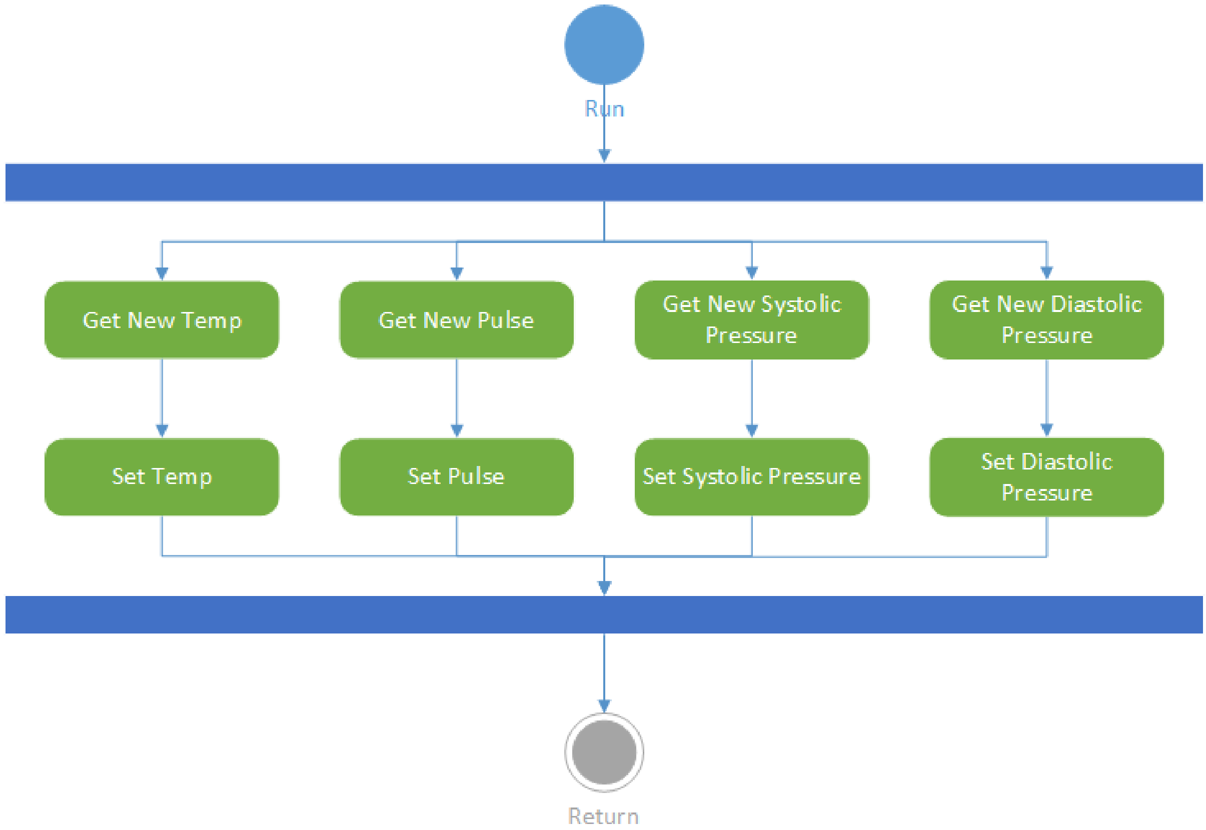

Figure 5: Measure Activity Diagram

B ACTIVITY DIAGRAMS

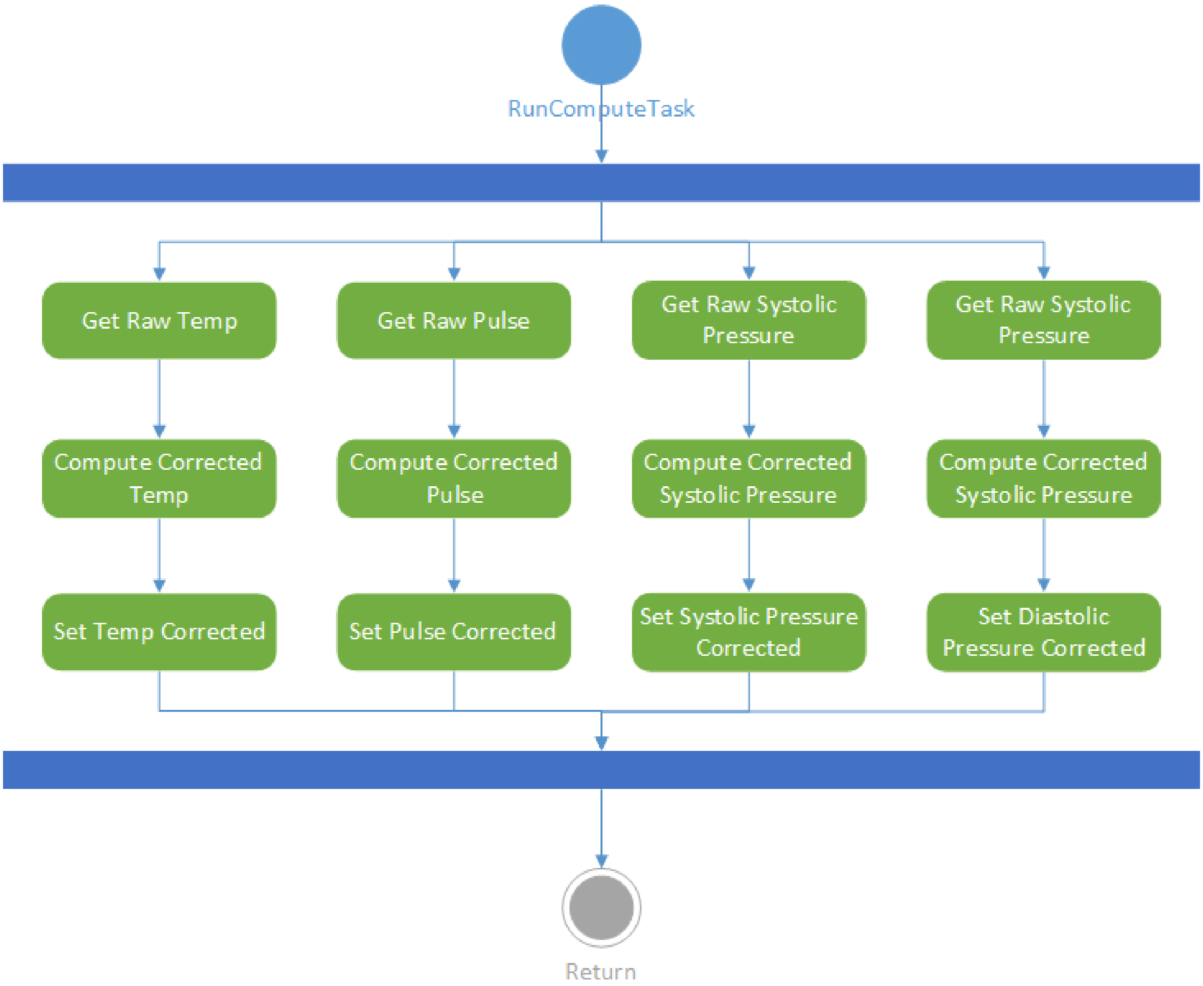

Figure 6: Compute Activity Diagram

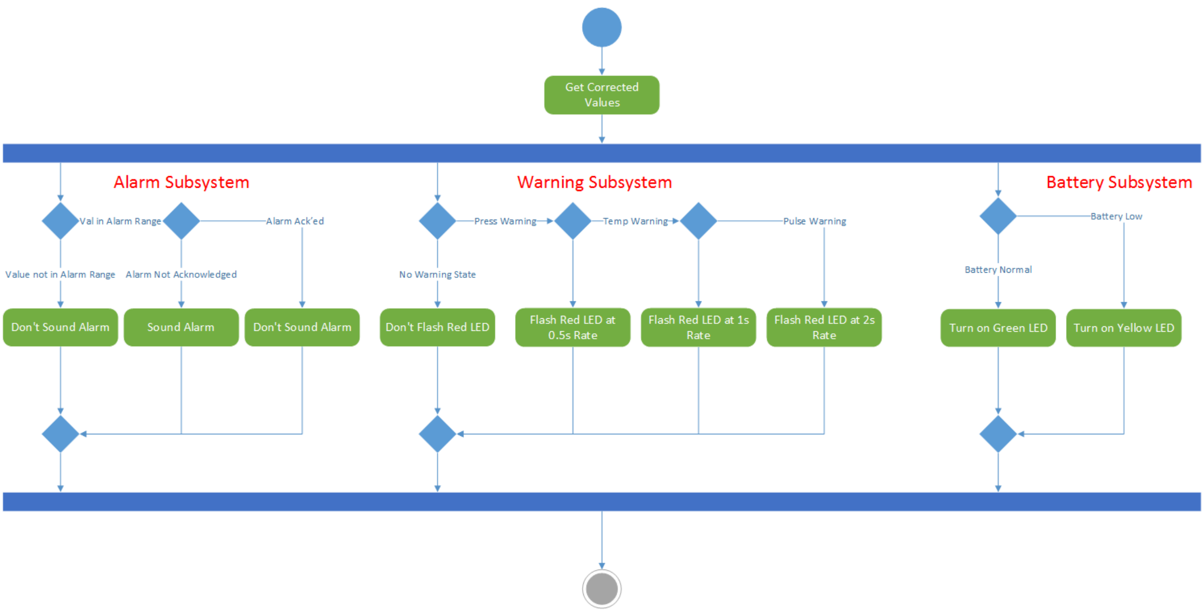

Figure 7: Warning Activity Diagram

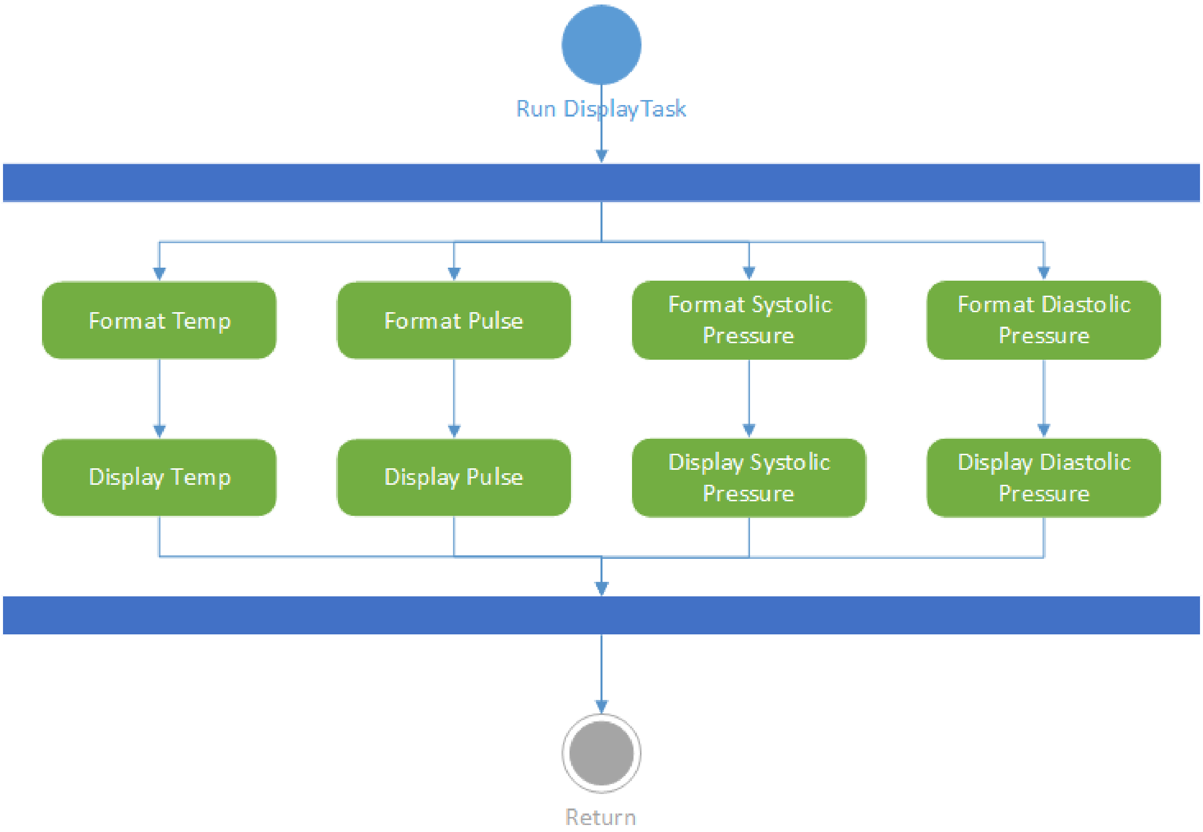

Figure 8: Display Activity Diagram

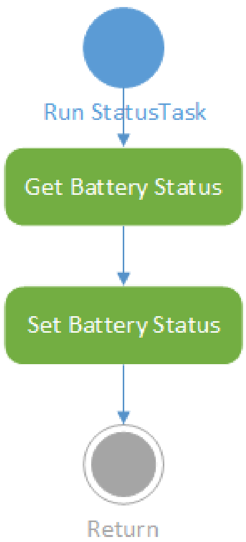

Figure 9: Status Activity Diagram

C SOURCE CODE

Source code for this project is provided below.

C.1 Main Function

../code/main.c

```
#include "schedule.h"

#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"

#define NUM_TASKS 5

#ifdef DEBUG
  void
__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void) {
  // Set the clocking to run directly from the crystal.
  SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
      SYSCTL_XTAL_8MHZ);

  initialize();  // from schedule.h

  while (1) {
    runTasks();  // from schedule.h
  }
}
```

C.2 Global Data

../code/globals.h

```
/*
 * globals.h
 * Author(s): Jonathan Ellington, Patrick Ma
 * 1/28/2014
 *
 * Defines global data for tasks to access
 * MUST be initialized before using
 */

#define DEBUG 0

#define TEMP_RAW_INIT 80        // initial 80
#define SYS_RAW_INIT 50         // initial 50
```

```c
#define DIA_RAW_INIT 50          // initial 50
#define PULSE_RAW_INIT 30        // initial 30

#define TEMP_CORR_INIT 0.0
#define SYS_CORR_INIT 0.0
#define DIA_CORR_INIT 0.0
#define PULSE_CORR_INIT 0.0

#define BATT_INIT 200

typedef struct correctedData {
  int temperatureRaw;
  int systolicPressRaw;
  int diastolicPressRaw;
  int pulseRateRaw;
  float temperatureCorrected;
  float systolicPressCorrected;
  float diastolicPressCorrected;
  float pulseRateCorrected;
  int batteryState;
} GlobalData;

extern GlobalData globalDataMem;

void initializeGlobalData();
```

../code/globals.c

```c
/*
 * globals.c
 * Author(s): Jonathan Ellington, Patrick Ma
 * 1/28/2014
 *
 * Defines global data for tasks to access
 */
#include "globals.h"

GlobalData globalDataMem;

void initializeGlobalData() {
  globalDataMem.temperatureRaw = TEMP_RAW_INIT;
  globalDataMem.systolicPressRaw = SYS_RAW_INIT;
  globalDataMem.diastolicPressRaw = DIA_RAW_INIT;
  globalDataMem.pulseRateRaw = PULSE_RAW_INIT;

  globalDataMem.temperatureCorrected = TEMP_CORR_INIT;
  globalDataMem.systolicPressCorrected = SYS_CORR_INIT;
  globalDataMem.diastolicPressCorrected = DIA_CORR_INIT;
  globalDataMem.pulseRateCorrected = PULSE_RAW_INIT;

  globalDataMem.batteryState = BATT_INIT;
}
```

C.3 Timebase

../code/timebase.h

```
/*
 * timebase.h
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Defines the major and minor cycles the system runs on
 */

#define MINOR_CYCLE 250     // minor cycle, in milliseconds
#define MAJOR_CYCLE  20     // major cycle, in number of minor cycles

#define IS_MAJOR_CYCLE (minor_cycle_ctr % MAJOR_CYCLE == 0)

extern unsigned int minor_cycle_ctr;    // counts number of minor cycles
```

C.4 Scheduler

../code/schedule.h

```
/*
 * schedule.h
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Defines the scheduler interface.  The scheduler
 * is responsible for running tasks on a specified
 * schedule.
 */

/*
 * Must be called before runTasks()
 * Initializes schedule required data structures
 */
void initialize();


/* Run all the tasks in the queue, delay minor cycle */
void runTasks();
```

../code/schedule.c

```
/*
 * schedule.c
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Implements schedule.h
 */

```

```c
#include "schedule.h"
#include "timebase.h"
#include "globals.h"

// Each task include
#include "measure.h"
#include "compute.h"
#include "oleddisplay.h"
#include "warning.h"
#include "status.h"

#define NUM_TASKS 5


// TCB
typedef struct tcb_struct {
  void (*runTaskFunction) (void*);
  void *taskDataPtr;
} TCB;

static TCB taskQueue[NUM_TASKS];     // The taskQueue holding TCB for each task
unsigned int minor_cycle_ctr = 0;    // minor cycle counter

// Private functions
TCB *getNextTask();
void initializeQueue();
void delay_in_ms(int ms);

// Must initialize before running this function!
void runTasks() {
  for (int i = 0; i < NUM_TASKS; i++) {
    TCB *task = &taskQueue[i];
    task->runTaskFunction(task->taskDataPtr);
  }
  delay_in_ms(MINOR_CYCLE);
  minor_cycle_ctr = minor_cycle_ctr+1;
}

// Initialize datastructures
void initialize() {
  initializeGlobalData();    // from globals.h

  // Initialize each task data
  initializeMeasureTask(measureData);   // from measure.h
  initializeComputeTask(computeData);   // from compute.h
  initializeDisplayTask(oledDisplayData);    // from oleddisplay.h
  initializeWarningTask(warningData);    // from oleddisplay.h
  initializeStatusTask(statusData);    // from oleddisplay.h

  // schedule each task
  initializeQueue();
}
```

```

// Initialize the taskQueue with each task
void initializeQueue() {
  // Measure Task
  taskQueue[0].runTaskFunction = measureTask; // from measure.h
  taskQueue[0].taskDataPtr = measureData;     // from measure.h

  // Compute Task (not yet implemented)
  taskQueue[1].runTaskFunction = computeTask; // from compute.h
  taskQueue[1].taskDataPtr = computeData;     // from compute.h

   // Compute Task (not yet implemented)
  taskQueue[2].runTaskFunction = oledDisplayTask; // from compute.h
  taskQueue[2].taskDataPtr = oledDisplayData;     // from compute.h

  // Compute Task (not yet implemented)
  taskQueue[3].runTaskFunction = warningTask; // from compute.h
  taskQueue[3].taskDataPtr = warningData;     // from compute.h

      // Compute Task (not yet implemented)
  taskQueue[4].runTaskFunction = statusTask; // from compute.h
  taskQueue[4].taskDataPtr = statusData;     // from compute.h
}

// Software delay
void delay_in_ms(int ms) {
  for (volatile int i = 0; i < ms; i++)
    for (volatile int j = 0; j < 800; j++);
}
```

C.5 Tasks

C.5.1 Measure Task

../code/measure.h

```
/*
 * measure.h
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Defines the interface for the measureTask.
 * initializeMeasureData() should be called before running measureTask()
 */

/* Points to the data used by measure */
extern void *measureData;

/* Initialize MeasureData, must be done before running measureTask() */
void initializeMeasureTask(void *measureData);

/* Perform the measure task */
```

```
void measureTask(void *dataptr);
```

../code/measure.c

```
/*
 * measure.h
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Implements measure.c
 */

#include "globals.h"
#include "timebase.h"
#include "measure.h"
#include "inc/hw_types.h"
#include "drivers/rit128x96x4.h"

// Used for debug display
#if DEBUG
#include "drivers/rit128x96x4.h"
#include <stdlib.h>
#include <stdio.h>
#endif

// Internal data structure
typedef struct measureData {
  int *temperatureRaw;
  int *systolicPressRaw;
  int *diastolicPressRaw;
  int *pulseRateRaw;
} MeasureData;

static MeasureData data;              // internal data
void *measureData = (void *)&data;   // external pointer to internal data

void initializeMeasureTask(void *data) {
#if DEBUG
  RIT128x96x4Init(1000000);
#endif
  MeasureData *mdata = (MeasureData *)data;
  mdata->temperatureRaw = &(globalDataMem.temperatureRaw);
  mdata->systolicPressRaw = &(globalDataMem.systolicPressRaw);
  mdata->diastolicPressRaw = &(globalDataMem.diastolicPressRaw);
  mdata->pulseRateRaw = &(globalDataMem.pulseRateRaw);
}

void setTemp(int *temp) {
    static unsigned int i = 0;
    static tBoolean goingUp = true;

    if (*temp > 50)
      goingUp = false;
```

```
    else if (*temp < 15)
      goingUp = true;

    if (goingUp) {
        if (i%2==0) (*temp)+=2;
        else (*temp)--;
    }
    else {
        if (i%2==0) (*temp)-=2;
        else (*temp)++;
    }

    i++;
}

void setSysPress(int *syspress) {
    // This is written to lab spec, with a flag to indicate "complete".
    // Right now, it does nothing, but I imagine it should probably be a global
    // variable to indicate to the compute task that the pressure measurement
    // is ready, since this measurement takes a nontrivial amount of time

    static unsigned int i = 0;

    tBoolean complete = false;

    if (*syspress > 100) {
      complete = true;
      *syspress = SYS_RAW_INIT;
    }

    if (i%2==0) (*syspress)+=3;
    else (*syspress)--;

    i++;
}

void setDiaPress(int *diapress) {
    static unsigned int i = 0;

    tBoolean complete = false;
    if (*diapress < 40) {
      complete = true;
      *diapress = DIA_RAW_INIT;
    }

    if (i%2==0) (*diapress)-=2;
    else (*diapress)++;

    i++;
}

```

```
void setPulse(int *pulse) {
    static unsigned int i = 0;

    static tBoolean goingUp = true;

    if (*pulse < 15)
      goingUp = true;
    else if (*pulse > 40)
      goingUp = false;

    if (goingUp) {
      if (i%2 == 0) (*pulse)--;
      else (*pulse)+=3;
    }
    else {
      if (i%2 == 0) (*pulse)++;
      else (*pulse)-=3;
    }

    i++;
}

void measureTask(void *dataptr) {
  // only run on major cycle
  if (IS_MAJOR_CYCLE) {   // on major cycle
    MeasureData *data = (MeasureData *) dataptr;

    setTemp(data->temperatureRaw);
    setSysPress(data->systolicPressRaw);
    setDiaPress(data->diastolicPressRaw);
    setPulse(data->pulseRateRaw);

#if DEBUG
    char num[30];
    sprintf(num, "Raw temp: %d", *(data->temperatureRaw));
    RIT128x96x4StringDraw(num, 0, 0, 15);

    sprintf(num, "Raw Syst: %d", *(data->systolicPressRaw));
    RIT128x96x4StringDraw(num, 0, 10, 15);

    sprintf(num, "Raw Dia: %d", *(data->diastolicPressRaw));
    RIT128x96x4StringDraw(num, 0, 20, 15);

    sprintf(num, "Raw Pulse: %d", *(data->pulseRateRaw));
    RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
  }
}
```

C.5.2 Compute Task

../code/compute.h

```c
/*
 * compute.h
 * Author(s): PatrickMa
 * 1/28/2014
 *
 * Defines the public interface for computeTask
 * initializeComputeData() should be called before running computeTask()
 */

/* Points to data used by compute */
extern void *computeData;

/*
 * Initializes the compute function. Should only be called once at the
 * beginning (i.e. at startup).
 */
void initializeComputeTask(void *computeData);

/* Carry out the compute task */
void computeTask(void *computeDataPtr);
```

../code/compute.c

```c
/*
 * compute.c
 * Author(s): PatrickMa
 * 1/28/2014
 *
 * Implements compute.c
 */

#include "compute.h"
#include "globals.h"
#include "timebase.h"

// Used for debug display
#if DEBUG
#include "drivers/rit128x96x4.h"
#include <stdlib.h>
#include <stdio.h>
#endif

// computeData structure internal to compute task
typedef struct computeData {
  // raw data pointers
  int *temperatureRaw;
  int *systolicPressRaw;
  int *diastolicPressRaw;
  int *pulseRateRaw;

  //corrected data pointers
  float *tempCorrected;
```

```c
  float *systPressCorrected;
  float *diastPressCorrected;
  float *pulseCorrected;
} ComputeData;

static ComputeData data;              // the internal data
void *computeData = (void *) &data;   // set the external ptr to the data

/*
 * Initializes the computeData task values (pointers to variables, etc)
 */
void initializeComputeTask(void *data) {
  ComputeData *cData = (ComputeData *) data;
  cData->temperatureRaw = &(globalDataMem.temperatureRaw);
  cData->systolicPressRaw = &(globalDataMem.systolicPressRaw);
  cData->diastolicPressRaw = &(globalDataMem.diastolicPressRaw);
  cData->pulseRateRaw = &(globalDataMem.pulseRateRaw);

  cData->tempCorrected = &(globalDataMem.temperatureCorrected);
  cData->systPressCorrected = &(globalDataMem.systolicPressCorrected);
  cData->diastPressCorrected = &(globalDataMem.diastolicPressCorrected);
  cData->pulseCorrected = &(globalDataMem.pulseRateCorrected);
}


/*
 * Linearizes the raw data measurement and converts value into human
 * readable format
 */
void computeTask(void *computeData) {
  if (IS_MAJOR_CYCLE) {
    ComputeData *cData = (ComputeData *) computeData;
    *(cData->tempCorrected) = 5 + 0.75 * (*(cData->temperatureRaw));
    *(cData->systPressCorrected) = 9 + 2 * (*(cData->systolicPressRaw));
    *(cData->diastPressCorrected) = 6 + 1.5 * (*(cData->diastolicPressRaw));
    *(cData->pulseCorrected) = 8 + 3 * (*(cData->pulseRateRaw));

#if DEBUG
    char num[30];
    sprintf(num, "Corrected temp: %f", *(cData->tempCorrected));
    RIT128x96x4StringDraw(num, 0, 40, 15);

    sprintf(num, "Raw Syst: %f", *(cData->systPressCorrected));
    RIT128x96x4StringDraw(num, 0, 50, 15);

    sprintf(num, "Raw Dia: %f", *(cData->diastPressCorrected));
    RIT128x96x4StringDraw(num, 0, 60, 15);

    sprintf(num, "Raw Pulse: %f", *(cData->pulseCorrected));
    RIT128x96x4StringDraw(num, 0, 70, 15);
#endif
  }
}
```

C.5.3 Display Task

../code/oleddisplay.h

```
/*
 * oledDisplay.h
 * Author(s): Jarrett Gaddy
 * 1/28/2014
 *
 * Defines the interface for the oledDisplay task
 * initializeDisplayData() should be called before running oleddisplayTask()
 */

/* Points to the data used by oledDisplay */
extern void *oledDisplayData;

/* Initialize displayData, must be done before running oledDisplayTask() */
void initializeDisplayTask(void *displayData);

/* Perform the oledDisplay task */
void oledDisplayTask(void *dataptr);
```

../code/oleddisplay.c

```
/*
 * oledDisplay.c
 * Author(s): jarrett Gaddy
 * 1/28/2014
 *
 * Implements oledDisplay.h
 */

#include "globals.h"
#include "timebase.h"
#include "oledDisplay.h"
#include "inc/hw_types.h"
#include "drivers/rit128x96x4.h"
#include <stdlib.h>
#include <stdio.h>


// Internal data structure
typedef struct oledDisplayData {
  float *temperatureCorrected;
  float *systolicPressCorrected;
  float *diastolicPressCorrected;
  float *pulseRateCorrected;
  int *batteryState;
} OLEDDisplayData;

static OLEDDisplayData data;              // internal data
```

```c
void *oledDisplayData = (void *)&data;  // external pointer to internal data

void initializeDisplayTask(void *data) {
  RIT128x96x4Init(1000000);

  OLEDDisplayData *mdata = (OLEDDisplayData *)data;
  mdata->temperatureCorrected = &(globalDataMem.temperatureCorrected);
  mdata->systolicPressCorrected = &(globalDataMem.systolicPressCorrected);
  mdata->diastolicPressCorrected = &(globalDataMem.diastolicPressCorrected);
  mdata->pulseRateCorrected = &(globalDataMem.pulseRateCorrected);
  mdata->batteryState = &(globalDataMem.batteryState);
}


void oledDisplayTask(void *dataptr) {
  // only run on major cycle
//  if (IS_MAJOR_CYCLE) {   // on major cycle
    OLEDDisplayData *data = (OLEDDisplayData *) dataptr;

    char num[30];
    sprintf(num, "Temperature: %.2f C ", *(data->temperatureCorrected));
    RIT128x96x4StringDraw(num, 0, 0, 15);

    sprintf(num, "Systolic Pressure:");
    RIT128x96x4StringDraw(num, 0, 10, 15);

  sprintf(num, "%.0f mm Hg ", *(data->systolicPressCorrected));
    RIT128x96x4StringDraw(num, 0, 20, 15);

    sprintf(num, "Diastolic Pressure:");
    RIT128x96x4StringDraw(num, 0, 30, 15);

  sprintf(num, "%.0f mm Hg ", *(data->diastolicPressCorrected));
    RIT128x96x4StringDraw(num, 0, 40, 15);

    sprintf(num, "Pulse rate: %d BPM ", (int) *(data->pulseRateCorrected));
    RIT128x96x4StringDraw(num, 0, 50, 15);

    sprintf(num, "Battery: %d %% ", *(data->batteryState)/2);
    RIT128x96x4StringDraw(num,0,60,15);
//  }
}
```

C.5.4 Warning/Alarm Task

../code/warning.h

```c
/*
 * warning.h
 * Author(s): Jarrett Gaddy
 * 1/28/2014
 *
```

```
 * Defines the interface for the warning task
 * initializeWarningData() should be called before running warningTask()
 */


#define WARN_LOW 0.95 //warn at 5% below min range value
#define WARN_HIGH 1.05 //warn at 5% above max range value
#define ALARM_LOW 0.90 //alarm at 10% below min range value
#define ALARM_HIGH 1.10 //alarm at 10% above max range value

#define ALARM_SLEEP_PERIOD 50   // duration to sleep in terms of minor cycles

#define WARN_RATE_PULSE    4    // flash rate in terms of minor cycles
#define WARN_RATE_TEMP     2
#define WARN_RATE_PRESS    1

#define TEMP_MIN 36.1
#define TEMP_MAX 37.8
#define SYS_MAX 120
#define DIA_MAX 80
#define PULSE_MIN 60
#define PULSE_MAX 100
#define BATTERY_MIN 40




/* Points to the data used by warning */
extern void *warningData;

/* Initialize displayData, must be done before running warningTask() */
void initializeWarningTask(void *displayData);

/* Perform the warning task */
void warningTask(void *dataptr);
```

../code/warning.c

```
/*
 * warning.c
 * Author(s): jarrett Gaddy, PatrickMa
 * 1/28/2014
 *
 * Implements warning.h
 */

#include "globals.h"
#include "timebase.h"
#include "warning.h"
#include "inc/hw_types.h"
#include "drivers/rit128x96x4.h"
#include <stdlib.h>
```

```c
#include <stdio.h>

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/pwm.h"
#include "driverlib/sysctl.h"
#include "drivers/rit128x96x4.h"


#define ALARM_SLEEP_PERIOD 50    // duration to sleep in terms of minor cycles

#define WARN_RATE_PULSE      4     // flash rate in terms of minor cycles
#define WARN_RATE_TEMP       2
#define WARN_RATE_PRESS      1

#define LED_GREEN  GPIO_PIN_6
#define LED_RED    GPIO_PIN_5
#define LED_YELLOW GPIO_PIN_7

typedef enum {OFF, ON, ASLEEP} alarmState;
typedef enum {NONE, WARN_PRESS, WARN_TEMP, WARN_PULSE} warningState;
typedef enum {NORMAL, LOW} batteryState;

//pin E0 for input on switch 3
//pin C5 C6 and C7 for led out

// Internal data structure
typedef struct WarningData {
  float *temperatureCorrected;
  float *systolicPressCorrected;
  float *diastolicPressCorrected;
  float *pulseRateCorrected;
  int *batteryState;
} WarningData;

static WarningData data;                      // internal data
static unsigned long ulPeriod;


void *warningData = (void *)&data;  // external pointer to internal data

/*
 * initializes task variables
 */
void initializeWarningTask(void *data) {
  //
  // Enable the peripherals used by this code. I.e enable the use of pin banks, etc.
  //
```

```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);        // bank C
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);        // bank E
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);        // bank F
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);        // bank G


    // configure the pin C5 for 4mA output
    GPIOPadConfigSet(GPIO_PORTC_BASE,LED_RED, GPIO_STRENGTH_4MA, GPIO_PIN_TYPE_STD);
    GPIODirModeSet(GPIO_PORTC_BASE, LED_RED, GPIO_DIR_MODE_OUT);

    // configure the pin C6 for 4mA output
    GPIOPadConfigSet(GPIO_PORTC_BASE,LED_GREEN, GPIO_STRENGTH_4MA, GPIO_PIN_TYPE_STD);
    GPIODirModeSet(GPIO_PORTC_BASE, LED_GREEN, GPIO_DIR_MODE_OUT);

    // configure the pin C7 for 4mA output
    GPIOPadConfigSet(GPIO_PORTC_BASE,LED_YELLOW, GPIO_STRENGTH_4MA, GPIO_PIN_TYPE_STD);
    GPIODirModeSet(GPIO_PORTC_BASE, LED_YELLOW, GPIO_DIR_MODE_OUT);

    // configure the pin E0 for input (sw3). NB: requires pull-up to operate
    GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
        GPIO_PIN_TYPE_STD_WPU);
    GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_DIR_MODE_IN);


    /*  This function call does the same result of the above pair of calls,
     *  but still requires that the bank of peripheral pins is enabled via
     *  SysCtrlPeripheralEnable()
     */
    //  GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, LED_RED);

    ///////////////////////////////////////////
    //  This section defines the PWM speaker characteristics
    ///////////////////////////////////////////

    //
    // Set the clocking to run directly from the crystal.
    //
    SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

    //
    // Set GPIO F0 and G1 as PWM pins.  They are used to output the PWM0 and
    // PWM1 signals.
    //
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0);
    GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_1);

    //
    // Compute the PWM period based on the system clock.
    //
```

```
  ulPeriod = SysCtlClockGet() / 65;

  //
  // Set the PWM period to 440 (A) Hz.
  //
  PWMGenConfigure(PWM0_BASE, PWM_GEN_0,
      PWM_GEN_MODE_UP_DOWN | PWM_GEN_MODE_NO_SYNC);
  PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, ulPeriod);

  //
  // Set PWM0 to a duty cycle of 25% and PWM1 to a duty cycle of 75%.
  //
  PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, ulPeriod / 4);
  PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ulPeriod * 3 / 4);

  //
  // Enable the PWM0 and PWM1 output signals.
  //
  PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT | PWM_OUT_1_BIT, true);

  // initialize the warning data pointers
  WarningData *mdata = (WarningData *)data;
  mdata->temperatureCorrected = &(globalDataMem.temperatureCorrected);
  mdata->systolicPressCorrected = &(globalDataMem.systolicPressCorrected);
  mdata->diastolicPressCorrected = &(globalDataMem.diastolicPressCorrected);
  mdata->pulseRateCorrected = &(globalDataMem.pulseRateCorrected);
  mdata->batteryState = &(globalDataMem.batteryState);
}

/////////////////////////////////////////////////////////////

/*
 * Warning task function
 */
void warningTask(void *dataptr) {

  static alarmState aState = OFF;
  static warningState wState = NONE;
  static batteryState bState = NORMAL;

  static warningState prevState;
  prevState = wState;

  static int wakeUpAlarmAt = 0;

  // Get measurement data
  WarningData *data = (WarningData *) dataptr;
  float temp = *(data->temperatureCorrected);
  float sysPress = *(data->systolicPressCorrected);
  float diaPress = *(data->diastolicPressCorrected);
  float pulse = *(data->pulseRateCorrected);
  int battery = *(data->batteryState);

```

```
// Alarm condition
if ( (temp < TEMP_MIN*ALARM_LOW || temp > (TEMP_MAX*ALARM_HIGH)) ||
    (sysPress > SYS_MAX*ALARM_HIGH) ||
    (diaPress > DIA_MAX*ALARM_HIGH) ||
    (pulse < PULSE_MIN*ALARM_LOW || pulse > PULSE_MAX*ALARM_HIGH) ) {

  // Should only turn alarm ON if it was previously OFF.  If it is
  // ASLEEP, shouldn't do anything.
  if (aState == OFF) aState = ON;
}
else
  aState = OFF;

// Warning Condition
if ( sysPress > SYS_MAX*ALARM_HIGH || diaPress > DIA_MAX*ALARM_HIGH )
  wState = WARN_PRESS;
else if ( temp < TEMP_MIN*WARN_LOW || temp > TEMP_MAX*WARN_HIGH )
  wState = WARN_TEMP;
else if ( pulse < PULSE_MIN*WARN_LOW || pulse > PULSE_MAX*WARN_HIGH )
  wState = WARN_PULSE;
else
  wState = NONE;

// Battery Condition
if (battery < BATTERY_MIN)
  bState = LOW;

// Handle speaker, based on alarm state
switch (aState) {
  case ON:
    PWMGenEnable(PWM0_BASE, PWM_GEN_0);
    break;
  case ASLEEP:
    PWMGenDisable(PWM0_BASE, PWM_GEN_0);
    break;
  default:  // OFF
    PWMGenDisable(PWM0_BASE, PWM_GEN_0);
    break;
}

// Handle warning cases
static int toggletime;
switch (wState) {
  case WARN_PRESS:
    GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);

    if (wState != prevState) {
      GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
      toggletime = WARN_RATE_PRESS;
    }
    else if (0 == minor_cycle_ctr%toggletime) {
      if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
        GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
```

```
        else
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00); // need to flash

        //toggletime+=WARN_RATE_PRESS;
      }

      break;
    case WARN_TEMP:
      GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);

      if (wState != prevState) {
        GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
        toggletime = WARN_RATE_TEMP;
      }
      else if (0 == minor_cycle_ctr%toggletime) {
        if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
        else
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00); // need to flash

        //toggletime+=WARN_RATE_TEMP;
      }
      break;
    case WARN_PULSE:
      GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);

      if (wState != prevState) {
        GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
        toggletime = WARN_RATE_PULSE;
      }
      else if (0==minor_cycle_ctr%toggletime) {
        if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF); // need to flash
        else
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00); // need to flash

        //toggletime+=WARN_RATE_PULSE;
      }
      break;
    default:  // NORMAL
      GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0xFF);
      GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0x00);
      break;
  }

  if (bState == LOW) {
    GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0x00);
  }
  else
    GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0x00);

  /* This is the alarm override
```

```
     * Upon override, the alarm is silenced for some time.
     * silence length is defined by ALARM_SLEEP_PERIOD
     *
     * If the button is pushed, the value returned is 0
     * If the button is NOT pushed, the value is non-zero
     */
    if (0 == GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_0) && (aState == ON) )
    {
      //GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0XFF);  // for debug, lights
       led
      aState = ASLEEP;
      wakeUpAlarmAt = minor_cycle_ctr + ALARM_SLEEP_PERIOD;
    }

    // Check whether to resound alarm
    if (minor_cycle_ctr == wakeUpAlarmAt && aState == ASLEEP) {
      aState = ON;
      GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0X00);  // for debug, kills
     led
    }
}
```

C.5.5 Status

../code/status.h

```
/*
 * status.h
 * Author(s): PatrickMa
 * 1/28/2014
 *
 * Defines the public interface for the status task
 *
 * initializeStatusTask() should be called once before performing status
 * functions
 */

/* Points to the data used by Status */
extern void *statusData;

/* Initialize StatusData, must be done before running functions */
void initializeStatusTask (void *statusData);

/* Perform the status tasks */
void statusTask(void *dataPtr);
```

../code/status.c

```
/*
 * status.c
 * Author(s): PatrickMa
 * 1/28/2014
```

```c
 *
 * implements status.h
 */

#include "status.h"
#include "globals.h"
#include "timebase.h"

// StatusData structure internal to compute task
typedef struct {
  int *batteryState;
} StatusData;

static StatusData data;      // the internal data
void *statusData = (void *) &data;  // sets the external ptr to the data;

/* Initialize the StatusData task values */
void initializeStatusTask (void *data) {
  StatusData *sdata = (StatusData *) data;
  sdata->batteryState = &(globalDataMem.batteryState);
}

/* Perform status tasks */
void statusTask(void *data){
  if (IS_MAJOR_CYCLE) {
  StatusData *sData = (StatusData *) data;
  if (*(sData->batteryState) > 0)
        *(sData->batteryState) = *(sData->batteryState) - 1;
  }
}
```