EE 472 Lab 3
Learning the Development Environment - The Next Step

Jonathan Ellington
Patrick Ma
Jarrett Gaddy

Contents

1 Abstract 1

2 Introduction 1

3 Discussion of the Lab 1
3.1 Design Specification . . . 1
3.1.1 Specification Overview . . . 1
3.1.2 Identified Use Cases . . . 2
3.1.3 Detailed Specifications (update) . . . 3
3.1.4 Detailed Task Specifications (updates) . . . 5
3.2 Software Implementation . . . 7
3.2.1 Functional Decomposition . . . 7
3.2.2 System Architecture . . . 7
3.2.3 High-level Implementation in C . . . 8
3.2.4 Task Implementation . . . 9

4 Presentation, Discussion, and Analysis of the Results 13
4.1 Results . . . 13
4.2 Discussion of Results . . . 13
4.3 Analysis of Any Errors . . . 14
4.4 Analysis of Implementation Issues and Workarounds . . . 14

5 Test Plan 15
5.1 Test Specification . . . 16
5.1.1 Scheduler . . . 16
5.1.2 Measure Task . . . 16
5.1.3 Compute Task . . . 17
5.1.4 Keypad Task . . . 17
5.1.5 Display Task . . . 17
5.1.6 Warning/Alarm Task . . . 17
5.1.7 Serial Task . . . 18
5.1.8 Status Task . . . 18
5.2 Test Cases . . . 18

6 Summary and Conclusion 20
6.1 Final Summary . . . 20
6.2 Project Conclusions . . . 20

A Breakdown of Lab Person-hours (Estimated) 21

B Activity Diagrams 22

C Source Code 27
C.1 Main Function . . . 27
C.2 Global Data . . . 27
C.3 Timebase . . . 30
C.4 Scheduler . . . 30
C.5 Tasks . . . 34

C.5.1 Task Interface . . . 34
C.5.2 Startup Task . . . 35
C.5.3 Measure Task . . . 36
C.5.4 Compute Task . . . 41
C.5.5 Keypad Task . . . 43
C.5.6 Display Task . . . 46
C.5.7 Warning/Alarm Task . . . 50
C.5.8 Serial Task . . . 58
C.5.9 Status . . . 60
C.6 Circular Buffer . . . 62

List of Tables

1 Specifications for measurement data . . . 4
2 Empirically determined task runtimes . . . 13
3 Estimated instructions per task, rounded to the nearest instruction . . . 14
4 Initial values and warning/alarm states . . . 17

List of Figures

1 Use case diagram . . . 2
2 Functional Decomposition . . . 7
3 System Architecture Diagram . . . 9
4 Control Flow Diagram . . . 11
5 Measure Activity Diagram . . . 22
6 Compute Activity Diagram . . . 23
7 Compute Activity Diagram . . . 24
8 Warning Activity Diagram . . . 24
9 Serial Activity Diagram . . . 25
10 Display Activity Diagram . . . 26
11 Status Activity Diagram . . . 26

1 ABSTRACT

In this lab the students are to take on the role of an embedded system design team. They will design modifications to the medical instrument previously designed. When the device finds metrics are out of the acceptable range, the user is notified, thus saving them from potential health risks. The students first laid out the design for their system using various design tools, then they implemented the system in software. Finally the students tested their system and verified 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. This implementation will build upon the previous implementation of the device by adding functionality. Added functions include heart rate sensor, keypad input, menu display, data logging, and UART serial communication.

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 8 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.

The initial Phase 1 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

In addition, the following requirements have been added:

- Utilize a hardware-based time reference
- support dynamic task creation and deletion
- support a user input device
- support data logging capabilities
- support remote communication capability
- Improve overall system performace
- Improve overall system safety

3.1.2 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.

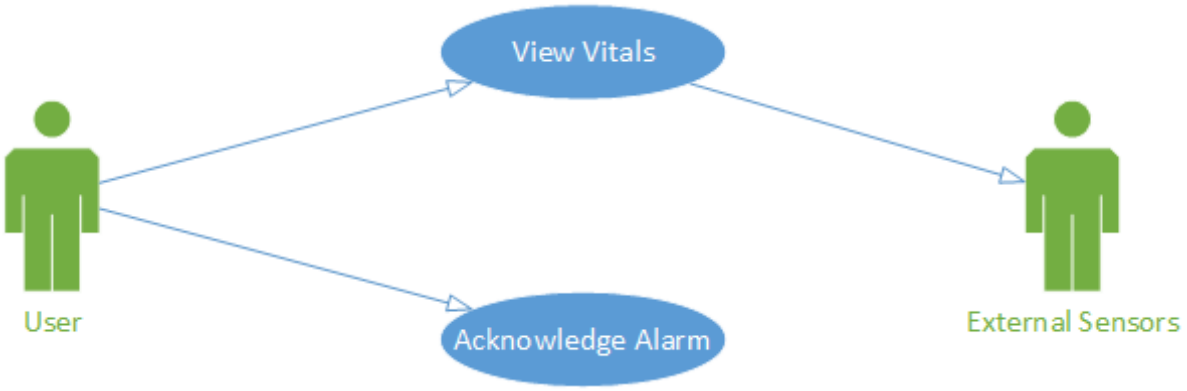

Figure 1: Use case diagram

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. The user can select toggle between a summary of current vitals as measured by the system or view measurements of each sensor individually.

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 significant amounts of time pass 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.

Use Case #3: View Measurement Logs

In the third use case, the user wishes to view previously recorded measurements for a given vital sign.

As the device is running, the user opts to enter View Logs mode. From this point, the user can select which vital sign they wish to examine. Upon selection, the data logged from previous measurements is displayed.

Possible exception conditions may include the following:

- *User wishes to view more data than the machine remembers* - The machine will not support this operation. The system is only able to display the amount of data defined by the device. Future variants may support an external storage or logging functionality
- *Machine loses power while reading or writing* - if the system loses power, any data stored in dynamic memory will be lost. On restart, data will be overwritten and lost. Future device versions may allow for storage in nonvolatile memory
- *Ongoing measurements trigger warning or alarms* - since measurements are taken continuously, the device may enter an alarm or warning state. In this case, the display will not change unless prompted to by the user. Any alert indicators (visual, audible, or remote messaging) will operate as normal in the background

Use Case #4: System Alert to Remote Terminal

In the event that the system enters an alert state (e.g. alarm or warning), the system can send a message to a remote terminal connected to the device. The messages sent can inform a second actor of the cause of alert and provide any additional useful information.

Possible exception conditions may include:

- *Improper configuration of the Remote Terminal* - If the remote terminal connection is improperly configured or initialized, data received may be corrupted and not display properly. The device cannot ensure a proper connection and it is the responsibility of the remote user to ensure the correct configuration is used
- *Remote Connection Lost* – If the user terminates the connection or the connection is lost, messages sent by the device may not arrive at the remote terminal or data may be corrupted. The device will not necessarily monitor the status of any remote connection; this responsibility is the remote terminals. In the event a connection is broken, the device system must continue to perform the other system functions without ill effects.

3.1.3 Detailed Specifications (update)

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

The system must have the following inputs:

- Alarm acknowledgment capability using a pushbutton
- Buttons or switches to allow user to access system modes and menu items
- Sensor measurement input capability consisting of:
  - Body temperature measurement
  - Pulse rate measurement signal
  - 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
- External data connection to a remote terminal

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 and the system cannot block and cease operation for 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%.

requirement modification: An alarm state occurs when the systolic blood pressure falls outside its stated normal range by 20%.

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.

3.1.4 Detailed Task Specifications (updates)

- New task: KeypadTask
  - The keypad task will scan the keypad and decode any keypresses
  - The task will have support the following user inputs:
  - Mode selection between 2 modes (1 button)
  - Menu selection between 3 options (1 button)
  - Alarm acknowledgement (1 button)
  - Up and down scroll functionality (1-2 buttons)
  - A new set of global variable will be created to store the state of the keypad and key presses
- New Task: Initialize (Startup):
  - Changes to Warn/AlarmTask:
  - The warnings will be activated and indicated as before in project 1
  - The alarm state is changed to activate only when the systolic pressure is 20% above the normal range.
  - The alarm will sound in 1 second tones (1 second on, 1 second off)
  - When an alarm or warning state occurs, the serial communication task will be added to the task queue
  - The deactivation period of the alarm sound is defined as 5 measurement periods
- New task: Serial Communication:
  - The task is enabled by the warn/alarm task
  - When run, the task will open an RS-232 connection at XXXX baud, XXinsertdetails hereXXX
  - The present corrected measurement will be displayed on the terminal in the same fashion as the display task annunciation mode
  - After sending data to the terminal, the serial communication task will remove itself from the task queue
- Changes to the MeasureTask:
  - Once a complete set of measurements has been taken, the compute task is added to the task queue
  - Pointers to the variables used in the measure task will be relocated to accommodate the new data architecture
  - The pulse measurement will monitor and count the frequency of a pulse rate event interrupt
  - A new value will be stored to memory if the present reading is greater than $\pm 15\%$ of the previous measurement

  - The measurement limits will correspond to 200bpm and 10bpm, determined empirically.

- Changes to ComputeTask:
  - All measurements will be recomputed
  - After computing the corrected values for all measurements, the ComputeTask will remove itself from the task queue

- Changes to DisplayTask:
  - Display will now support multiple display options
  - Menu mode will allow selection of each of the individual measurements. Upon selection of a measurement, the current value of the measurement will be displayed onscreen
  - Annunciation mode will display the current status of each measurement as in project 1, and provide the same functionality as the display in project 1.

- Changes to Warn/AlarmTask:
  - The warnings will be activated and indicated as before in project 1
  - The alarm state is changed to activate only when the systolic pressure is 20% above the normal range.
  - The alarm will sound in 1 second tones (1 second on, 1 second off)
  - When an alarm or warning state occurs, the serial communication task will be added to the task queue
  - The deactivation period of the alarm sound is defined as 5 measurement periods

- Changes to Schedule:
  - System will maintain a list of activated and deactivated tasks. The list must be updatable during runtime based on the system state
  - The hardware timer will provide a system interrupt every 250ms or equal to the minor cycle, whichever is shorter
  - At runtime, upon a timer interrupt event, all tasks will be added or removed according to the task activation list. The tasks will then be run
  - Task Control Blocks will have forward and backward pointers to allow references to the next task
  - The scheduler cannot block for five seconds
  - The scheduler will toggle a GPIO pin at least once during execution of the task queue

- Changes to StatusTask:
  - There are no changes to the status task

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 high level functional blocks were developed. These blocks are shown in Figure 2.

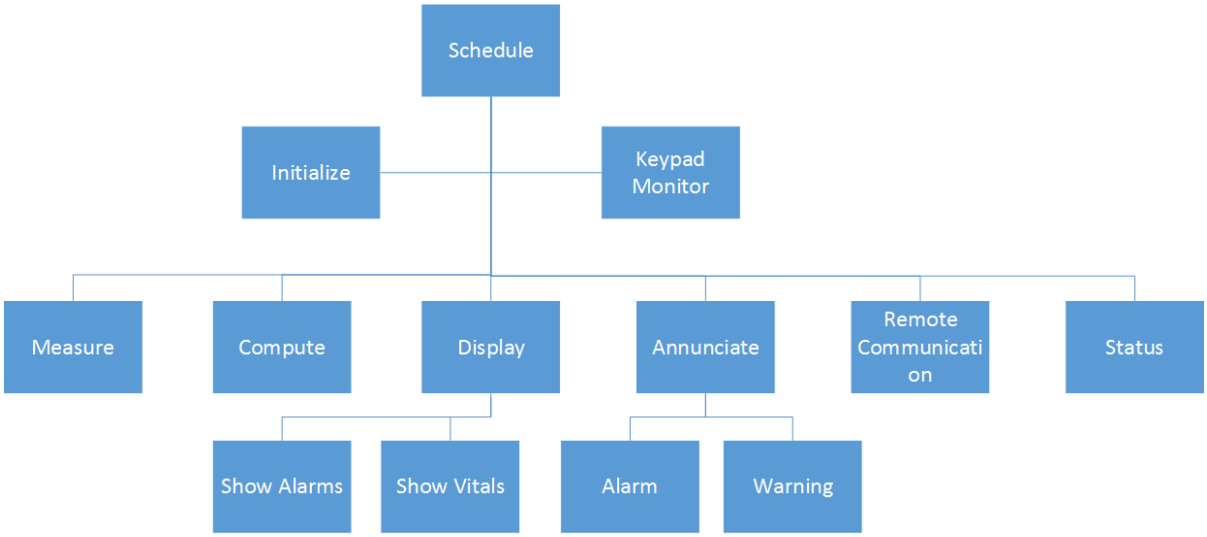

Figure 2: Functional Decomposition

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 eight main tasks:

Initialization Task This task Initializes data structures and does system startup-related jobs. This task is not actually scheduled to run, it only executes a single time at system startup.

Measure Task The measurement task is in charge of interacting with the blood pressure, temperature, and pulse sensors. Each of these is simulated. The blood pressure and temperature are simulated in the CPU. The task will measure the pulse rate by parsing an externally generated square wave of varying frequency; the pulse rate being proportional to the frequency.

Compute Task Compute takes the simulated raw data and converts it to the correct units of measurement. Raw temperature data to Celsius, blood pressure to mm Hg, and pulse rate to BPM.

Keypad Monitor Keypad will check the keypad for user input. It should provide the user with four keys: two for scrolling, one for selection, and one to go back.

Display Task The display task will show a user interface on the Stellaris OLED. The user will interact with the display using a keypad. Under normal operation, a menu will be displayed asking users which measurement they would like to see. If the user presses back while in this menu, they enter annunciate mode which displays all the measurements currently in warning or alarm state.

Warning/Alarm Task Under normal operation, this task will light a green LED signifying that everything is OK. If one of the measurements enters a warning state, the task will flash a red LED at a rate specific to the warning for that measurement. If there is an alarm state, it sound the alarm by driving the speaker. At any time if the battery goes too low, the yellow LED will illuminate.

Remote Communication Task This task is in charge of sending data to a remote terminal. If any of the states are in a warning or alarm condition, this task will transmit the (corrected) data to the remote terminal.

Status Task Status receives information about the battery on the system and updates it's current data accordingly.

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
- **task.h** - Defines the TCB interface for a task

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:

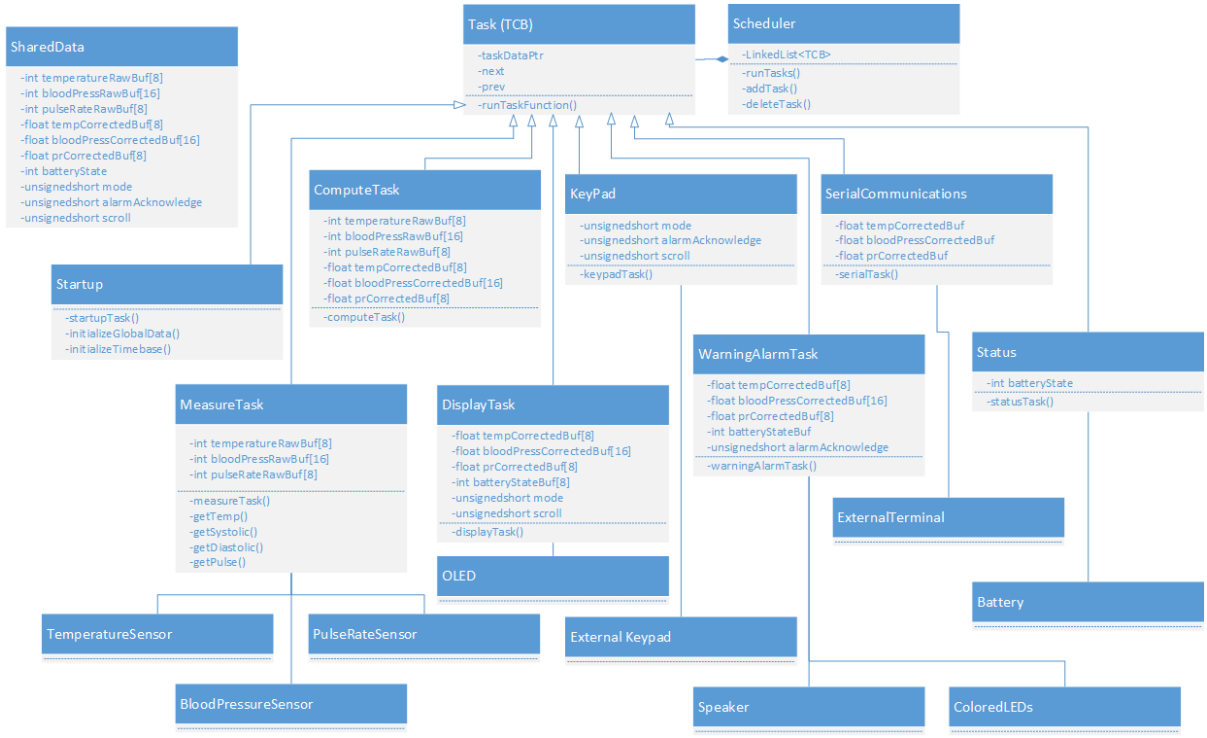

Figure 3: System Architecture Diagram

```
1   struct TCB {
2     void *taskDataPtr;
3     void (*taskRunFn)(void *);
4   }
5
```

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 and starts the hardware timer then runs into an infinite while loop. Inside the while loop a run method is called. The run method is part of the scheduler. The run method has a runTask flag which determines whether or not anything should actually be run this call. The flag is set to true by the hardware timer's interrupt handler. Once the runTask flag is true the run method will keep track of whether the device is on a minor cycle or a major cycle and run the preform task method of each task. The runTask flag is then set to false so that the tasks will not be executed again until the hardware interrupt has again

been triggered. The tasks included in this project are Compute, Measure, Warning, keyPad, OLEDDisplay, Serial, 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 contains pointers to the preform task function and the data for the task and also has fields for pointers to a next TCB and previous TCB. The TCB is used by the scheduler to run the task. The scheduler contains a doubly linked list of TCB objects that uses the TCB next and previous elements to point to the next and previous tasks in the task queue. In this case there are 7 tasks but not always 7 tasks in the list of tasks to run. The compute task is only to be run after the measure task, and the serial task only needs to be run at certain times. When a task is not being used it's TCB is not included in the linked list of tasks. Therefore and updateQueue method was created. This method checks flags set within the tasks that are running and determines if a task that isnt in the list needs to be added, or if a task that is in the list needs to be removed. The scheduler's run task contains a loop that runs through the linked list of TCBs and runs the function pointed to by the TCB with the argument of the data pointer stored in the TCB until the null value pointed to by the last element in the list is reached. After running all tasks the runTask flag is set to false again. The hardware timer will count up until it reaches the number of ms that corresponds to a minor cycle then trigger a hardware interrupt. The interrupt handler sets the runTask flag back to true and the task linked list will be updated, traversed and run again.

control flow is shown in Figure 4.

Each task has its own unique purpose in the system, and each uses a different part of the global data.

Measure Task 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. In the newly improved design a heart rate monitor has been added. The heart rate monitor uses an interrupt handler to count the number of rising edges on an input in a 2 Major cycle period. The number is then used to calculate the heart rate the sensor is receiving. Additionally the measure task adds the compute task to the task queue after it has run.

Compute Task The compute task is very simple. This task will only run after it has been placed into the task queue by the measure task and it will remove itself from the queue after running. 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 and keypad data. A diagram of the Compute activity is shown in the appendix in Figure 6.

Warning Task 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, F, and G. These are enabled using the SysCtlPeripheralEnable

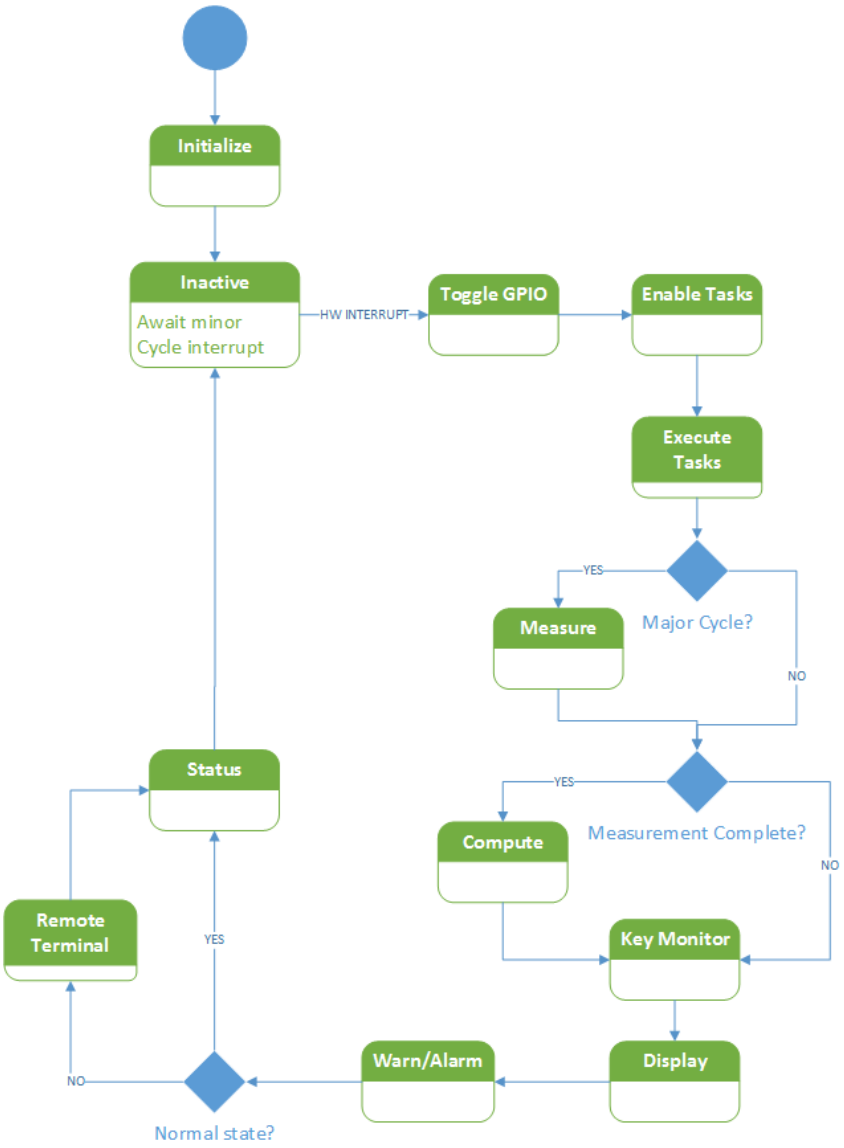

Figure 4: Control Flow Diagram

library call. Additionally the task set up pins C 5, 6, and 7 as outputs, and pins F 0 and G 1 as PWM outputs. 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 8 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 the systolic blood pressure data is out of range by more than 20%. If the value is out of that range then the PWM output is enabled using PWMGenEnable and pulsed with a 2 second period. 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 acknowledge button is sensed in the keyPad task and a global data value global.alarmAcknowledge is set. When the alarmAcknowledge field is set to 1 the alarm will go to its resting state. 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. In addition to the led flashing the warning will also tell the scheduler

to add the serialCommunication task to the task list. 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.

Keypad Task The keyPad task uses the GPIO libraries to set up 5 inputs. 1 input is pin E 0 and is used as the alarm acknowledge button. The other 4 pins are inputs on pins D 4, D 5, D 6, and D 7, and are used to detect button presses on the external keypad. The keypad works by connecting 2 of the outputs from the keypad together. There are 4 row outputs and 4 column outputs for a total of 16 possible combinations. The design was simplified by only using 1 column of 4 buttons. By doing this, the column keyPad wire can be supplied with a constant 5V from the Stellaris board, then each of the 4 row lines can be connected to a GPIO input. when a button in the live column is pushed the 5V column line is connected to 1 of the GPIO pins which lets the keyPad task know which button was pushed. Every time the keyPad task is run the GPIO will tell the task what buttons are pressed. 2 of the buttons correspond to up and down for scroll in the menu, 1 button corresponds to select in the menu, and 1 button changes the mode between menu and annunciation. The data that the buttons manipulate are the global keyPad data parts which are global.select, global.scroll, and global.mode. Additionaly the input on pin E 0 manipulates the global.alarmAcknowledge signal. The activity diagram for this task is shown in the appendix in Figure 7.

Display Task To show a user their current medical measurements, the system also has an oledDisplay task. This task uses the corrected data from measurements, the battery state, and the keyPad data. The display task has an activity diagram shown in Figure 10 in the appendix. This task uses the usnprintf() 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 display has 2 modes. The mode that the screen currently displays is determined by the global.mode data which is set in the keyPad task. When mode is 0 this is the menu mode. This mode displays each of the 4 measurement types, temperature, blood pressure, pulse rate, and battery without the data for each. The task then takes the global.scroll data from keyPad and displays a cursor next to the measurement that scroll is currently at, 0 corresponding to temperature, 1 to blood pressure, and so on. If the global.select is set to 1 in keyPad then the currently scrolled to measurement is selected and the screen will change to showing only that measurement and its data. When global.mode is equal to 0 the annunciation screen will instead be displayed. This screen shows each measurement followed by its data.

Serial Task The serial task is only run when a warning occurs. The warning causes the TCB for the serial task to be added to the TCB schedule linked list. When the serial task is run the first time it will initialize a UART connection using UARTConfigSet and UARTEnable driver functions to enable the UART that communicates to an FTDI chip on the Stellaris board. The FTDI chip then converts the UART to a virtual serial port over the USB cable to the PC. After the UART is initialized on the first run and in all subsequent runs of the serial task, all the measurements and their data are formatted and printed into 1 buffer using the usnprintf function. A loop then iterates through the buffer writing each character in the buffer to the UART. An activity diagram showing the serial task is located in the appendix as Figure 9.

Status Task 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 11 in the appendix.

4 PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS

4.1 Results

Patrick

The project was completed and demonstrated on February 13, 2014.

Demonstration of the system to the interested parties showed that the system met the majority of 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.

During the demonstration, all tasks worked as designed and expected with the exception of

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

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

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

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 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

- BPM Range error - Serial error

There were two errors found in the final project. First, the pulse rate range is not exactly as specified. Second, it was found that the serial communications task did not produce the correct values.

The specified corrected pulse rate was to be between 10 BPM and 200 BPM. For simplicity, the raw pulse rate was implemented as a 1-1 mapping from frequency to raw pulse rate. For example, a 1 Hz frequency would produce a value of 1 for raw pulse rate, and a 15 Hz signal would produce a value of 15 for raw pulse rate. This caused two issues. When the Input frequency is 1, the measured BPM (using the specified raw to correct conversion) is $8+3\cdot(1)=11BPM$. This is larger than the specified 10 BPM minimum. Also, when the input frequency is 64 Hz, a corrected value of $8+3\cdot(64)=200BPM$ is expected. However, as the frequency increased, the overhead of the other running tasks became significant. As a result, more rising edges could fit in our measurement interval than we expected. This resulted in a maximum BPM of roughly 206 BPM.

There was also an error in the Serial communication protocol in which the usnprintf command used to format the data was causing a runtime error when printing formatted data to a buffer. Rather than printing the actual corrected values to the buffer (to be sent to the remote terminal), it seemed the command was printing addresses. It was believed this was caused by an error with the CircularBuffer implementation, though no other tasks had issues with it (and extensive tests validated the CircularBuffer).

4.4 Analysis of Implementation Issues and Workarounds

The medical instrument design in this project was completed and tested successfully to meet almost all the requirements, the designers did face a few difficulties in designing the device, however, because this design was additional functionality added to a previous projects design, many of the errors previously encountered were easily avoided due to experience of the students, and already completed coding work.

Many of the challenges the designers of this project faced were in the keypad input and the data output. The keypad input posed a difficult hardware challenge as there are 16 input keys on the keypad but only 8 connections for the microprocessor to connect to the keypad. This

means that to identify a single key press 4 connections must be set as outputs and 4 as inputs. The inputs can then be scanned as the outputs are set 1 at a time to find which key is pressed. Instead of implementing this design, the students instead opted to use only 4 buttons on the keypad. This allowed the strobe design to be ignored. Instead 1 row of keys was activated all the time and that row was scanned for button presses.

In addition to keypad input, there was also difficulty in implementing data formatting functions. After adding a hardware delay, IAR workbench no longer allows the use of sprintf which had previously been used to print and format data. The usnprintf command was instead used to format and print data to a buffer, however, the students found that usnprintf does not have the ability to print floating point data. This issue was resolved by changing measurements that were previously printed as floats to be printed as integers. Usnprintf also caused issues when printing certain data for the serial task. In this case the usnprintf was causing a runtime error and freezing the operation of our device. This issue remained unresolved.

All problems but one were solved before demonstrating the product to the interested parties. The final project still contained the serial error previously mentioned.

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.

The inclusion of additional specifications for Phase II of the project requires additional tests to ensure the system meets the customer requirements.

Phase II Tests:

- Scheduler loops through linked list properly
- Scheduler adds and removes from the linked list the following tasks correctly:
  - Compute task added by measure task
  - Serial communication task added by annunciation task
  - Compute task removed by itself
  - Serial communication task removed by itself
- Warning task alarm meets the following two requirements

1. Has one (1) second tones; a total period of 2 seconds
2. Activates only when systoic pressure is greater than 20% above normal
3. Has an auditory deactivation or "sleep" period of 5 measurement cycles

- Serial task displays the temperature, blood pressure, pulse rate, and battery status as listed in Section 3.1
- Keypad task captures user inputs, sets the appropriate inputs, and causes the assocated changes in system state
- Hardware timer updates the system's mnor cycle counter

More detailed explanation of the tests performed is provided in the following sections.

5.1 Test Specification

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. Also, the scheduler needs to successfully add and remove tasks from the queue dynamically. Specifically, the Measure Task should be able to tell the scheduler to add the Compute Task and the Warning/Alarm Task should be able to schedule the Serial Task. Both Compute and Serial should be able to be removed from the schedule.

5.1.2 Measure Task

For this design, the temperature and blood pressure values were simulated on the CPU. The pulse rate was simulated using an externally generated square wave of varying frequency.

- **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 rate should match one-to-one with the frequency of the input signal. For example, a 15 Hz signal should produce a raw pulse rate of 15.
- **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.

The Measurement Task should also successfully add the Compute Task to the schedule queue.

5.1.3 Compute Task

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

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

The compute task should also successfully remove itself from the schedule queue.

5.1.4 Keypad Task

The keypad should be tested to successfully capture user input. When the select button is pressed, the measurement selection value should reflect the selected measurement. When the up scroll button is pressed, the scroll value should be incremented, and when the down scroll button is pressed, the scroll value should be decremented. If the alarm acknowledge button is pressed, this should be reflected in the alarmAcknowledge global value.

5.1.5 Display Task

On load, the display task should present the user with an option to select the desired measurement. If the back button is pressed, the annunciation screen should be displayed, showing the measurements in warning or alarm state.

5.1.6 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 4.

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

This task should also add the Serial task if any of the measurements are in a warning or alarm condition.

5.1.7 Serial Task

If any of the measurements are in warning or alarm state, this task should send this data serially to a remote terminal. The task should send all the data (not just the data in warning or alarm state). It should be printed in the following format:

```
1. Temperature                0 C
2. Systolic Blood Pressure    0 mm Hg
3. Diastolic Blood Pressure   0 mm Hg
4. Pulse Rate                 0 BPM
5. Battery                    0 %
```

5.1.8 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 4 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 acknowledgment 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 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.

In this device, a new pulse rate monitor device was added. To test the pulse rate monitor the monitor input was connected to a function generator generating square waves. As the square wave frequency was increased the pulse rate value was expected to increase. This was verified to be working correctly. As the pulse rate passed through the warning zones the design for pulse rate warnings was also verified to be working correctly.

Additionally, the improved medical device now has a menu that is displayed on the OLED display and navigated using the testbench keypad. The design operating these functions was tested by navigating to each part of the menu using the keys on the keypad and visually verifying that each part of the menu displayed the correct data. Each menu, annunciation, main menu, and each measurements selection mode, was visually verified to be working as intended. The keypad functionality was verified as each button was used to navigate through the menu.

The newly added serial communication was then tested using the hyperterm program on the lab station PC. The serial connection was established over a virtual COM port on the USB connection from the PC to the Stellaris board. The program had the intended functionality of displaying each of the measurements, and its current data on the serial port whenever the alarm state was entered. The functionality could be verified by watching the hyperterm screen to see if the data displayed when a warning state occurred. The data on hyperterm could then be compared to the OLED display data to verify its accuracy.

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

A medical monitoring system with a user interface, alarm notification, and remote terminal display was designed, implemented, and tested. The design simulated temperature and blood pressure, and obtained pulse rate data from an external function generator. These results were converted to a human readable form, and tested to see if there was a warning or alarm state. In the event of warning or alarm, the user was notified visually with LEDs and aurally with an alarm sound. The warning and alarm data was transferred to a remote terminal.

Some implementation errors were encountered. The pulse rate range could not fit the specification exactly, and the serial communications task was not printing the correct values to the remote terminal. Aside from these two implementation errors, the device worked as specified.

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	20	10	10	12+
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.

B ACTIVITY DIAGRAMS

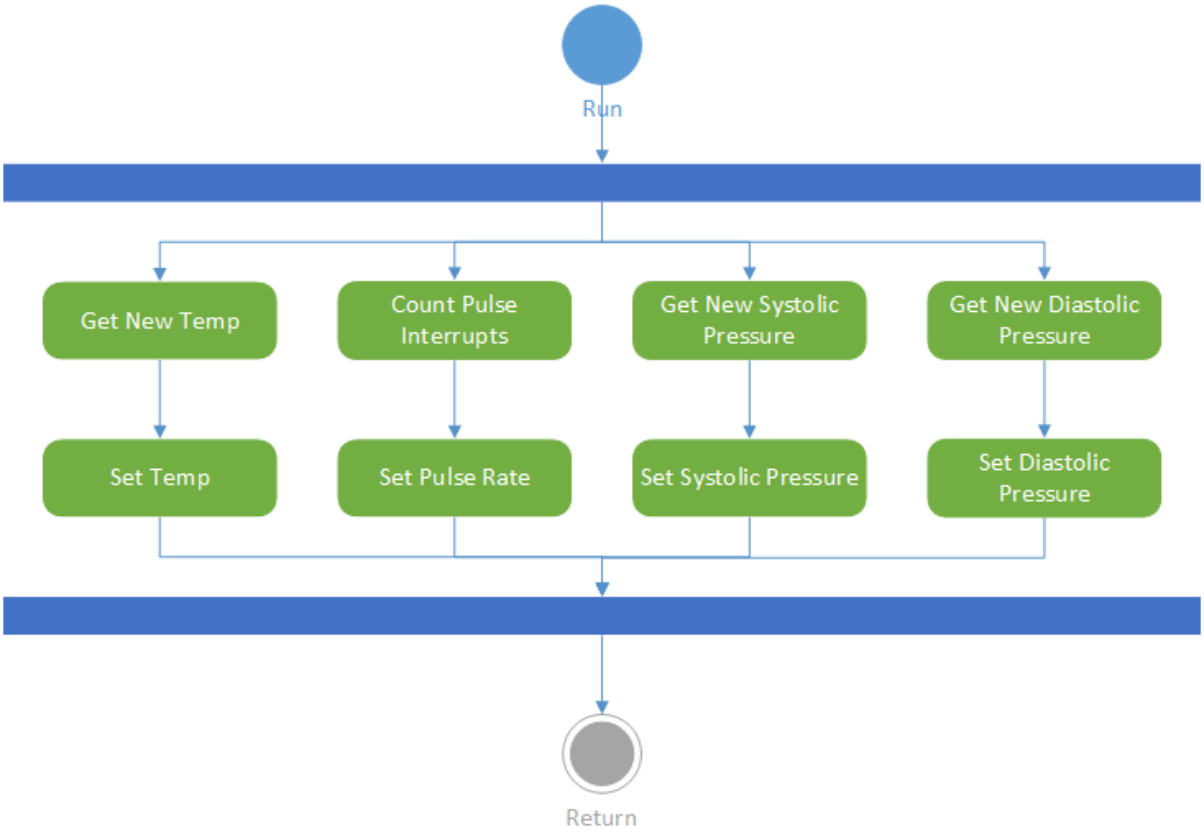

Figure 5: Measure Activity Diagram

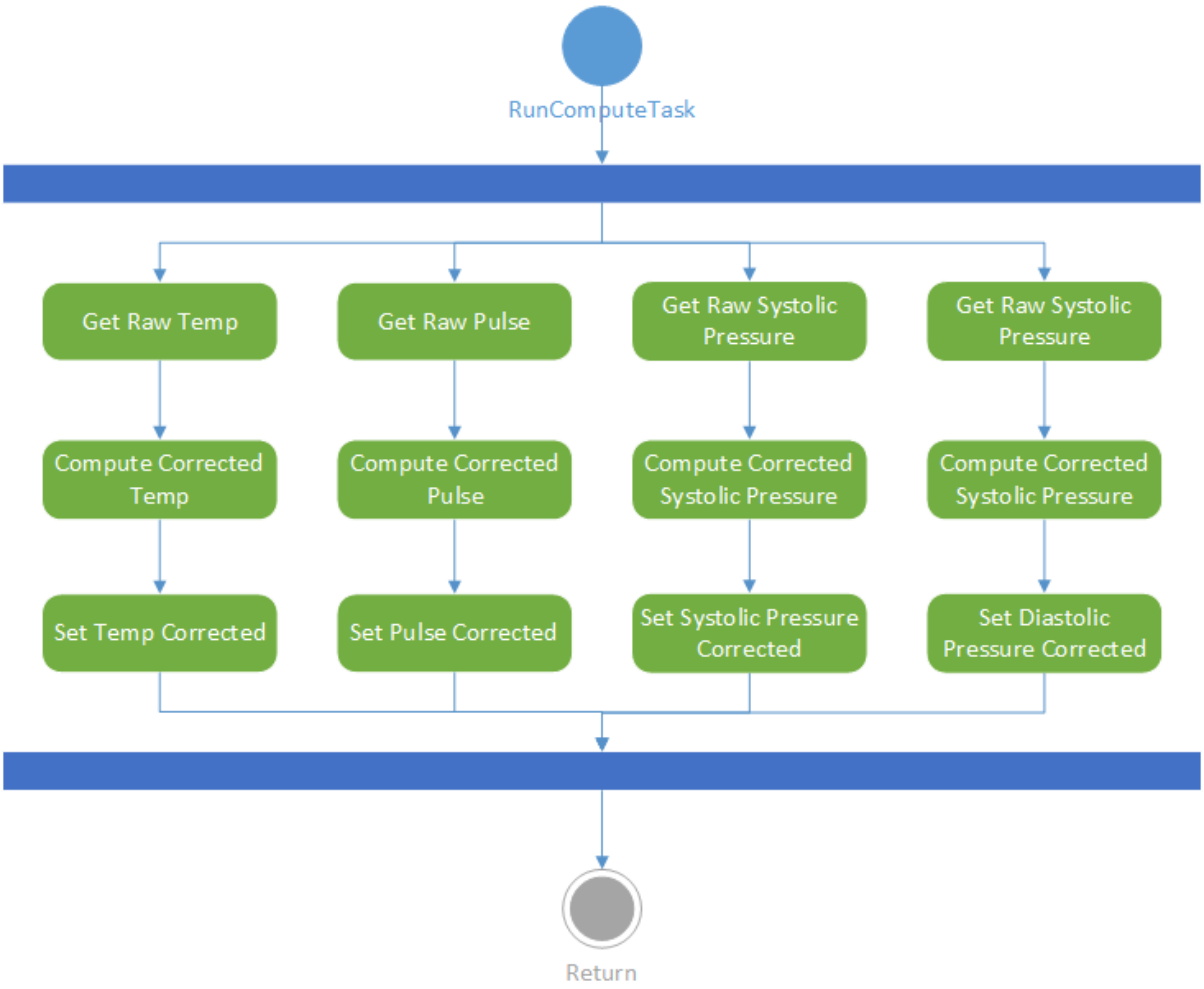

Figure 6: Compute Activity Diagram

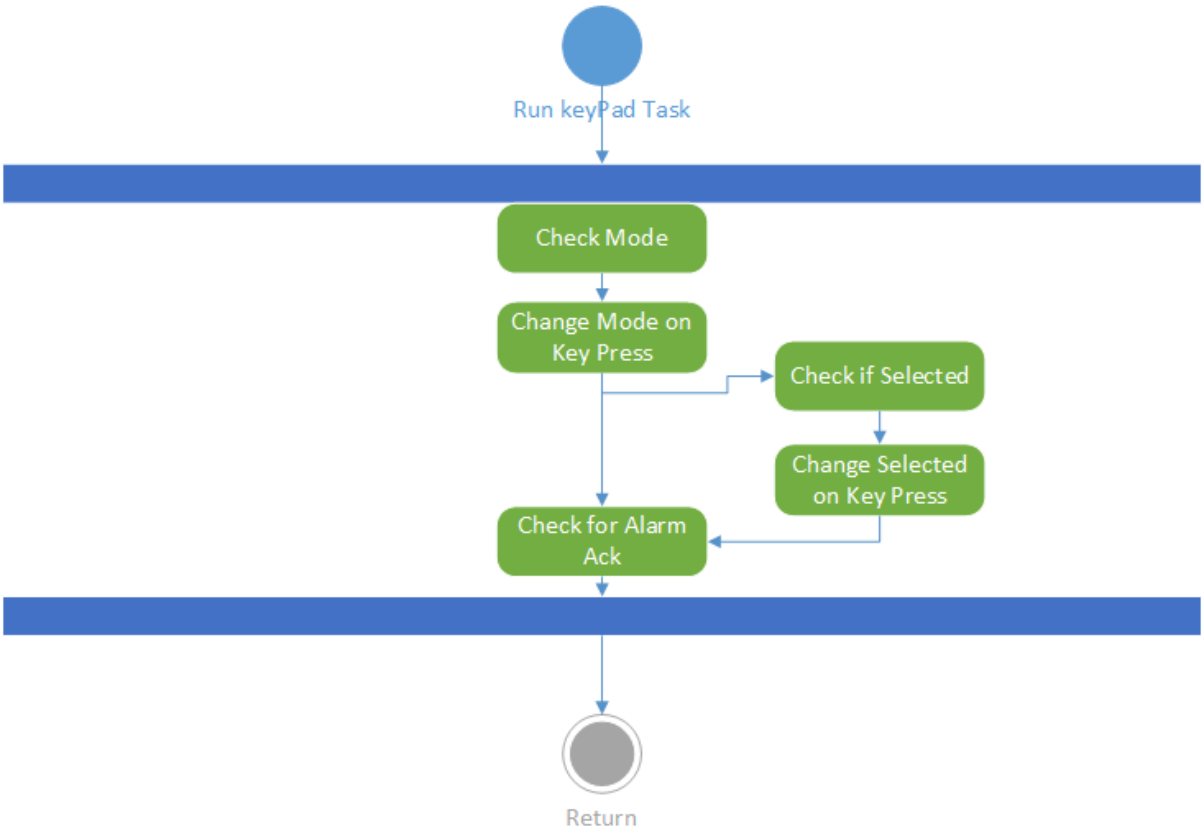

Figure 7: Keypad Activity Diagram

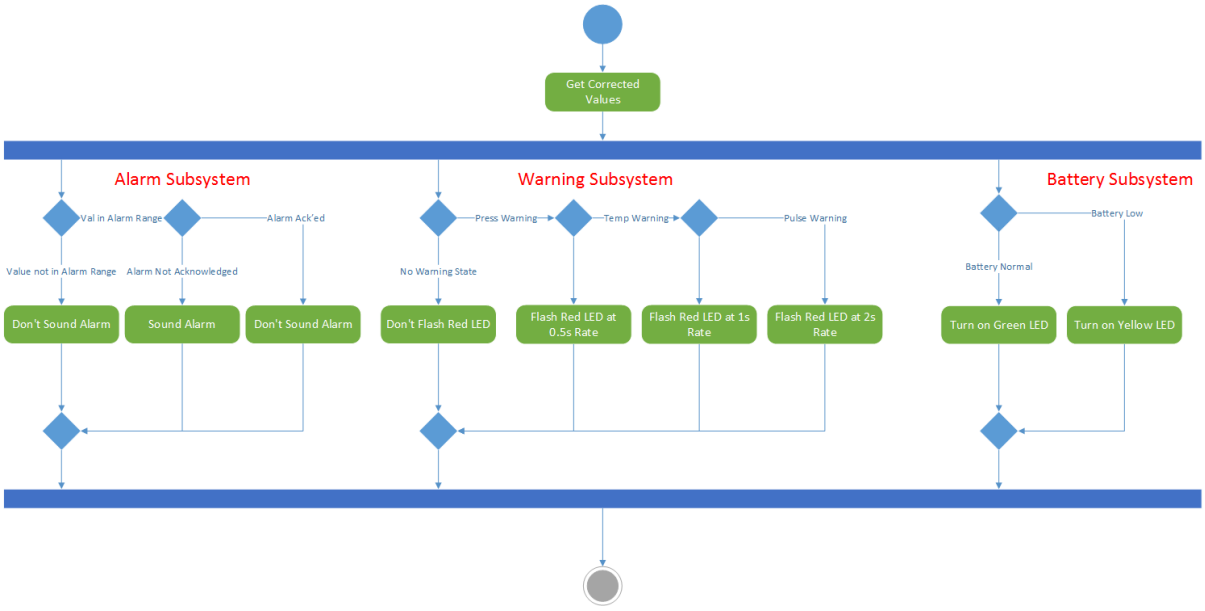

Figure 8: Warning Activity Diagram

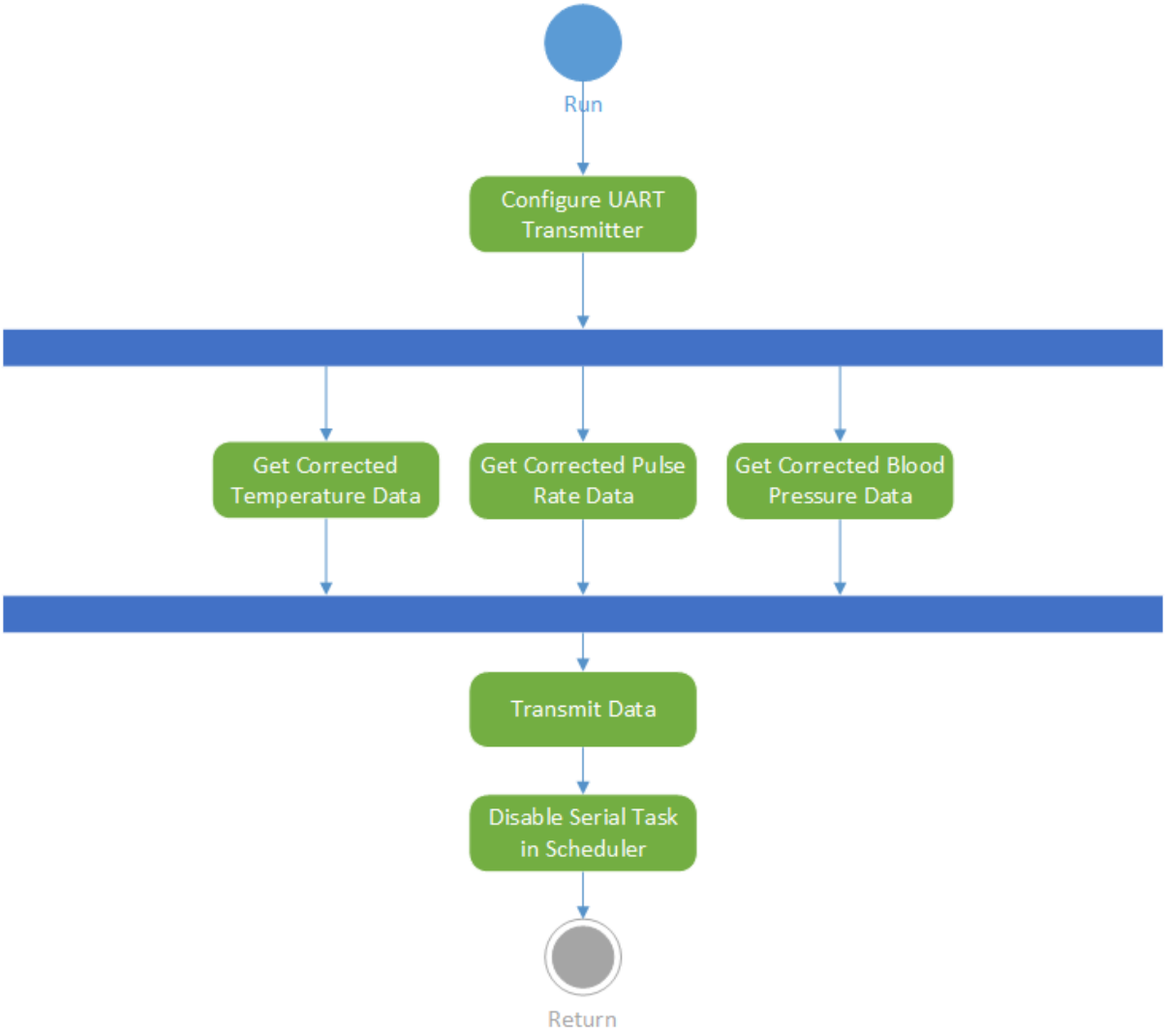

Figure 9: Serial Activity Diagram

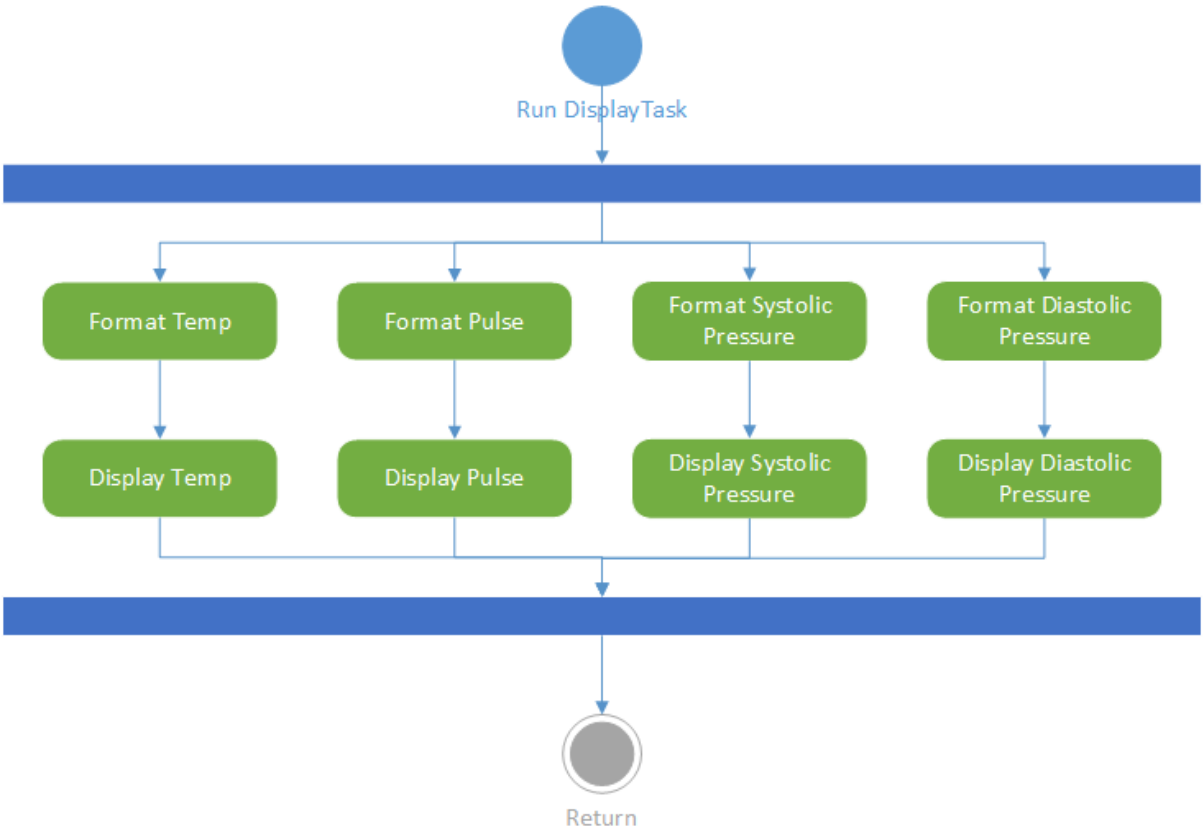

Figure 10: Display Activity Diagram

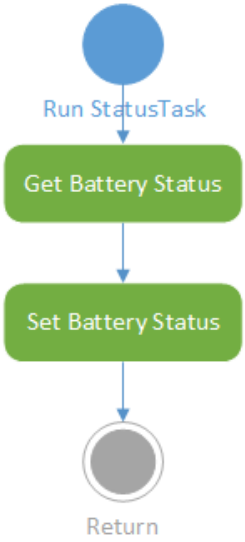

Figure 11: Status Activity Diagram

C SOURCE CODE

Source code for this project is provided below.

C.1 Main Function

../code/main.c

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

#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "inc/hw_memmap.h"
#include "driverlib/systick.h" // for systick interrupt (minor cycle)
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"


#define NUM_TASKS 5
#define BOARD_CLK_RATE  8000000 // 8MHz

#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);

  startup();    // initialize system state and variables

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

C.2 Global Data

../code/globals.h

```c
/*
 * globals.h
 * Author(s): Jonathan Ellington, Patrick Ma
 * 1/28/2014
```

```
 *
 * Defines global data for tasks to access
 * MUST be initialized before using
 */

#include "inc/hw_types.h"
#include "CircularBuffer.h"
#include "stddef.h"
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"

#define DEBUG 0

#define TEMP_RAW_INIT 80        // initial 80
#define SYS_RAW_INIT 80        // initial 50
#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 global_data {
  CircularBuffer temperatureRaw;
  CircularBuffer systolicPressRaw;
  CircularBuffer diastolicPressRaw;
  CircularBuffer pulseRateRaw;

  CircularBuffer temperatureCorrected;
  CircularBuffer systolicPressCorrected;
  CircularBuffer diastolicPressCorrected;
  CircularBuffer pulseRateCorrected;

  unsigned short batteryState;
  unsigned short mode;
  unsigned short measurementSelection;
  tBoolean alarmAcknowledge;
  tBoolean select;
  unsigned short scroll;
} GlobalData;

extern GlobalData global;

// initializes the global variables for use by system
void initializeGlobalData();

// allows use of pin47 for debug, toggles the pin hi or lo alternatively
void debugPin47();
```

../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 global;

// The arrays to be wrapped in a
// circular buffer
static int temperatureRawArr[8];
static int systolicPressRawArr[8];
static int diastolicPressRawArr[8];
static int pulseRateRawArr[8];

static float temperatureCorrectedArr[8];
static float systolicPressCorrectedArr[8];
static float diastolicPressCorrectedArr[8];
static float pulseRateCorrectedArr[8];

void initializeGlobalData() {
  // Wrap the arrays
  global.temperatureRaw = cbWrap(temperatureRawArr, sizeof(int), 8);
  global.systolicPressRaw = cbWrap(systolicPressRawArr, sizeof(int), 8);
  global.diastolicPressRaw = cbWrap(diastolicPressRawArr, sizeof(int), 8);
  global.pulseRateRaw = cbWrap(pulseRateRawArr, sizeof(int), 8);

  global.temperatureCorrected = cbWrap(temperatureCorrectedArr, sizeof(float), 8);
  global.systolicPressCorrected = cbWrap(systolicPressCorrectedArr, sizeof(float), 8);
  global.diastolicPressCorrected = cbWrap(diastolicPressCorrectedArr, sizeof(float), 8);
  global.pulseRateCorrected = cbWrap(pulseRateCorrectedArr, sizeof(float), 8);

  int tr = TEMP_RAW_INIT;
  int sr = SYS_RAW_INIT;
  int dr = DIA_RAW_INIT;
  int pr = PULSE_RAW_INIT;

  float tc = TEMP_CORR_INIT;
  float sc = SYS_CORR_INIT;
  float dc = DIA_CORR_INIT;
  float pc = PULSE_CORR_INIT;

  // Add initial values
  cbAdd(&(global.temperatureRaw), &tr);
```

```
48    cbAdd(&(global.systolicPressRaw), &sr);
49    cbAdd(&(global.diastolicPressRaw), &dr);
50    cbAdd(&(global.pulseRateRaw), &pr);
51
52    cbAdd(&(global.temperatureCorrected), &tc);
53    cbAdd(&(global.systolicPressCorrected), &sc);
54    cbAdd(&(global.diastolicPressCorrected), &dc);
55    cbAdd(&(global.pulseRateCorrected), &pc);
56
57    // Set normal variables
58    global.batteryState = 200;
59    global.mode = 0;
60    global.measurementSelection = 0;
61    global.alarmAcknowledge = false;
62    global.select = false;
63    global.scroll = 0;
64  }
65
66  // debug tool
67  void debugPin47() {
68    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);          // debug
69    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
70    long a = GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_2);
71    if(0 == a)
72    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
73    else
74    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
75  }
```

C.3 Timebase

../code/timebase.h

```
1  /*
2   * timebase.h
3   * Author(s): Jonathan Ellington
4   * 1/28/2014
5   *
6   * Defines the major and minor cycles the system runs on
7   */
8
9  #define MINOR_CYCLE 250       // minor cycle, in milliseconds
10 #define MAJOR_CYCLE 4       // major cycle, in number of minor cycles
11 #define PULSE_CYCLE 2        // pulse rate measurement time, in number of
12                              // major cycles.
13 #define IS_PULSE_CYCLE (minor_cycle_ctr % (MAJOR_CYCLE * PULSE_CYCLE) == 0)
14 #define IS_MAJOR_CYCLE (minor_cycle_ctr % MAJOR_CYCLE == 0)
15
16 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.
 */

#include "inc\hw_types.h"

// globally setable variables
extern tBoolean computeActive;
extern tBoolean serialActive;


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


/* 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
 */

#include "task.h"
#include "schedule.h"
#include "timebase.h"
#include "globals.h"

// Each task include
#include "measure.h"
#include "compute.h"
#include "display.h"
#include "warning.h"
#include "status.h"
#include "serial.h"
#include "keyPad.h"

// Used for debug display
```

```
24 #if DEBUG
25 #include "drivers/rit128x96x4.h"
26 #include "utils/ustdlib.h"
27 #endif
28
29
30 unsigned int minor_cycle_ctr = 0;   // minor cycle counter
31
32 static TCB *currentTaskPtr; // taskQueue pointers
33 static TCB *listHeadPtr;
34 static TCB *listTailPtr;
35
36 // flags to track task states
37 tBoolean computeActive;
38 tBoolean serialActive;
39 static tBoolean computeInQueue;
40 static tBoolean serialInQueue;
41
42 tBoolean runSchedule = false;
43
44
45 // Private functions
46 void delay_in_ms(int ms);
47 void insertNextNode(TCB *newNode);
48 void deleteCurrent();
49 void updateQueue();
50 void debug();
51
52 // Must initialize before running this function!
53 void runTasks() {
54   if(runSchedule) {
55   runSchedule = false;
56
57   updateQueue();
58
59   while (NULL != currentTaskPtr) {
60     currentTaskPtr->runTaskFunction(currentTaskPtr->taskDataPtr);
61     currentTaskPtr = currentTaskPtr->nextTCB; // go to next task
62   }
63   }
64 }
65
66
67 // Initialize the taskQueue with each task
68 void initializeQueue() {
69   // listhead > measure > keyPad > display > warn > status > null
70   listHeadPtr = &measureTask; // from measure.h
71   measureTask.nextTCB = &keyPadTask;
72   keyPadTask.nextTCB = &displayTask;
73   displayTask.nextTCB = &warningTask;
74   warningTask.nextTCB = &statusTask;
75   statusTask.nextTCB = NULL;
76
```

```
77    // backwards pointers listTailPtr > status > ...
78    listTailPtr = &statusTask;
79    statusTask.prevTCB = &warningTask;
80    warningTask.prevTCB = &displayTask;
81    displayTask.prevTCB = &keyPadTask;
82    keyPadTask.prevTCB = &measureTask;
83    measureTask.prevTCB = NULL;
84
85    currentTaskPtr = listHeadPtr; // and set up to start at the top
86  }
87
88  /* Traverse the taskQueue and insert/delete tasks based on set flags.
89   * currentTaskPtr pointer is reset to the head of the list */
90  void updateQueue() {
91    // update computeTask
92    if (computeActive && !computeInQueue) {
93      currentTaskPtr = &measureTask;
94      insertNextNode(&computeTask);
95      computeInQueue = true;
96    }
97    if (!computeActive && computeInQueue) {
98      currentTaskPtr = &computeTask;
99      deleteCurrent();
100     computeInQueue = false;
101   }
102   // update serialTask
103   if (serialActive && !serialInQueue) {
104     currentTaskPtr = &warningTask;
105     insertNextNode(&serialTask);
106     serialInQueue = true;
107   }
108   if (!serialActive && serialInQueue) {
109     currentTaskPtr = &serialTask;
110     deleteCurrent();
111     serialInQueue = false;
112   }
113   currentTaskPtr = listHeadPtr;
114 }
115
116 // inserts the given node into the list as the next node. currentTaskPtr
      pointer
117 // moves to point at the newly inserted node.
118 // Source: code derived in part from JD Olsen, Innovative Softwear and TA,
      Ltd
119 void insertNextNode(TCB *newNode) {
120   if(NULL == listHeadPtr) { // empty list
121     listHeadPtr = newNode;
122     listTailPtr = newNode;
123   }
124   else {  // insert the newNode between two nodes
125     newNode -> nextTCB = currentTaskPtr -> nextTCB;
126     currentTaskPtr -> nextTCB = newNode;
127     newNode -> prevTCB = currentTaskPtr;
```

```
128     if (NULL == (newNode -> nextTCB)) { // newest node is at list end
129       listTailPtr = newNode;
130     } else {  // in the middle somewhere
131       newNode -> nextTCB -> prevTCB = newNode;
132     }
133   }
134   currentTaskPtr = newNode;
135 }
136
137 /* Removes the currentTaskPtr node from the taskQueue.
138  * Moves currentTaskPtr pointer to the nextTCB task in the list. */
139 void deleteCurrent() {
140   if (NULL == currentTaskPtr -> nextTCB) {  // edge case: at last task
141     listTailPtr = currentTaskPtr -> prevTCB;
142     listTailPtr -> nextTCB = NULL;
143   } else {  // reassign pointers
144     currentTaskPtr -> nextTCB -> prevTCB = currentTaskPtr ->prevTCB;
145     currentTaskPtr -> prevTCB -> nextTCB = currentTaskPtr -> nextTCB;
146   }
147   currentTaskPtr = currentTaskPtr -> nextTCB; // update currentTaskPtr
       pointer
148 }
149
150 // Software delay
151 void delay_in_ms(int ms) {
152   for (volatile int i = 0; i < ms; i++)
153     for (volatile int j = 0; j < 800; j++);
154 }
155
156 // for debug (obviously)
157 void debug() {
158   char num[30];
159     static int test = 0;
160     usnprintf(num, 30, "Test number: %d  ", test);
161     RIT128x96x4StringDraw(num, 0, 90, 15);
162     test++;
163 }
```

C.5 Tasks

C.5.1 Task Interface

../code/task.h

```
1 /*
2  * task.h
3  * Author(s): Jonathan Ellington
4  * 1/28/2014
5  *
6  * Defines the interface for a task
7  */
8
```

```
#ifndef _TASK_H
#define _TASK_H

typedef struct tcb_struct {
  void (*runTaskFunction) (void*);
  void *taskDataPtr;
  struct tcb_struct *nextTCB; // pointer to next TCB
  struct tcb_struct *prevTCB; // points to previous TCB
} TCB;

#endif // _TASK_H
```

C.5.2 Startup Task

../code/startup.h

```
/*
 * startup.h
 * author: Patrick ma
 * Date: 2/13/2014
 *
 * initializes the system and system variables
 */

extern tBoolean runSchedule;

// Interrupt handler for hw timer
void SysTickIntHandler();

// configure the hardware timer
void initializeHWCounter();

// initialize system variables and hardware timer
void startup();
```

../code/startup.c

```
/*
 * startup.c
 * author: Patrick ma
 * Date: 2/13/2014
 *
 * initializes the system and system variables
 */

#include "timebase.h"
#include "globals.h"
#include "driverlib/systick.h" // for systick interrupt (minor cycle)
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "startup.h"
#include "schedule.h"
```

```
16
17
18
19
20 /*
21  * Minor Cycle Handler
22  */
23 void SysTickIntHandler (void) {
24   minor_cycle_ctr = minor_cycle_ctr + 1;
25   runSchedule = true;
26 }
27
28 /*
29  * configure the hw interrupt for minor cycle
30  */
31 void initializeHWCounter() {
32 SysTickPeriodSet(SysCtlClockGet() * MINOR_CYCLE / 1000);  // set timer
      period
33
34 // enable interrupts (master)
35 IntMasterEnable();  // this may be redundant, consider moving
36 SysTickIntEnable();   // enable systick interrupt
37 SysTickEnable();  // enable systic counter
38 }
39
40
41 /*
42  * Initializes hw counter ans system state variables
43  */
44 void startup() {
45   initializeHWCounter();
46
47   #if DEBUG
48   RIT128x96x4Init(1000000);
49   #endif
50
51   // Initialize global data
52   initializeGlobalData();   // from globals.h
53
54   computeActive = false;  // neither serial or compute task runs at start up
55   serialActive = false;
56
57   initializeQueue(); // start up task queue with basic tasks
58 }
```

C.5.3 Measure Task

../code/measure.h

```
1 /*
2  * measure.h
3  * Author(s): Jonathan Ellington
```

```
 * 1/28/2014
 *
 * Defines the interface for the measureTask.
 * initializeMeasureData() should be called before running measureTask()
 */

#include "task.h"

/* Points to the TCB for measure */
extern TCB measureTask;
```

../code/measure.c

```
/*
 * measure.h
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Implements measure.c
 *
 * Uses port PA4 for interrupt input for pulse rate
 */

#include "task.h"
#include "CircularBuffer.h"
#include "globals.h"
#include "timebase.h"
#include "measure.h"
#include "schedule.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "inc/hw_ints.h"
#include "driverlib/debug.h"

// Used for debug display
#if DEBUG
#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
#endif

// prototype for compiler
void measureRunFunction(void *dataptr);
void PulseRateISR(void);

// Internal data structure
typedef struct measureData {
  CircularBuffer *temperatureRaw;
  CircularBuffer *systolicPressRaw;
  CircularBuffer *diastolicPressRaw;
```

```
41    CircularBuffer *pulseRateRaw;
42  } MeasureData;
43
44  static int pulseRate = 0;
45  static MeasureData data;  // internal data
46  TCB measureTask = {&measureRunFunction, &data};  // task interface
47
48  void initializeMeasureTask() {
49  #if DEBUG
50    RIT128x96x4Init(1000000);
51  #endif
52    // Load data memory
53    data.temperatureRaw = &(global.temperatureRaw);
54    data.systolicPressRaw = &(global.systolicPressRaw);
55    data.diastolicPressRaw = &(global.diastolicPressRaw);
56    data.pulseRateRaw = &(global.pulseRateRaw);
57
58
59    /* Interrupt setup
60     * Note: using statically registered interrupts, because they're faster
61     *       this means we aren't using the dynamic GPIOPortIntRegister()
       function,
62     *       instead, an entry in the interrupt table is populated with the
       address
63     *       of the interrupt handler (under GPIO Port A) and this is enabled
       with
64     *       IntEnable(INT_GPIOA) */
65
66    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
67
68    // Set PA4 as input
69    GPIOPinTypeGPIOInput(GPIO_PORTA_BASE, GPIO_PIN_4);
70
71    // Enable interrupts on PA4
72    GPIOPinIntEnable(GPIO_PORTA_BASE, GPIO_PIN_4);
73
74    // Set interrupt type
75    GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_4, GPIO_RISING_EDGE);
76
77    // Enable interrupts to the processor.
78    IntMasterEnable();
79
80    // Enable the interrupts.
81    IntEnable(INT_GPIOA);
82  }
83
84  void setTemp(CircularBuffer *tbuf) {
85    static unsigned int i = 0;
86    static tBoolean goingUp = true;
87
88    int temp = *(int *)cbGet(tbuf);
89
90    if (temp > 50)
```

```
91      goingUp = false;
92    else if (temp < 15)
93      goingUp = true;
94
95    if (goingUp) {
96      if (i%2==0) temp+=2;
97      else temp--;
98    }
99    else {
100     if (i%2==0) temp-=2;
101     else temp++;
102   }
103
104   cbAdd(tbuf, &temp);
105
106   i++;
107 }
108
109 void setBloodPress(CircularBuffer *spbuf, CircularBuffer *dpbuf) {
110   // This is written to lab spec, with a flag to indicate "complete".
111   // Right now, it does nothing, but I imagine it should probably be a
        global
112   // variable to indicate to the compute task that the pressure measurement
113   // is ready, since this measurement takes a nontrivial amount of time
114
115   static unsigned int i = 0;
116
117   static tBoolean sysComplete = false;
118   static tBoolean diaComplete = false;
119
120   int syspress = *(int *)cbGet(spbuf);
121
122   // Restart systolic measurement if diastolic is complete
123
124   if (syspress > 100)
125     sysComplete = true;
126
127   if (! sysComplete) {
128     if (i%2==0) syspress+=3;
129     else syspress--;
130   }
131
132   int diapress = *(int *)cbGet(dpbuf);
133
134   // Restart diastolic measurement if systolic is complete
135   if (diapress < 40)
136     diaComplete = true;
137
138   if (!diaComplete) {
139     if (i%2==0) diapress-=2;
140     else diapress++;
141   }
142
```

```
143   if (diaComplete && sysComplete) {
144     sysComplete = false;
145     diaComplete = false;
146     syspress = SYS_RAW_INIT;
147     diapress = DIA_RAW_INIT;
148   }
149
150   cbAdd(spbuf, &syspress);
151   cbAdd(dpbuf, &diapress);
152
153   i++;
154 }
155
156 // pulse rate interrupt handler
157 void PulseRateISR(void) {
158   pulseRate++;
159
160   GPIOPinIntClear(GPIO_PORTA_BASE, GPIO_PIN_4);
161 }
162
163 void measureRunFunction(void *dataptr) {
164   static tBoolean onFirstRun = true;
165   static int rate;
166   MeasureData *mData = (MeasureData *) dataptr;
167
168   if (onFirstRun) {
169     initializeMeasureTask();
170     rate = *(int*) cbGet(mData->pulseRateRaw);
171     onFirstRun = false;
172   }
173
174   // capture pulse rate
175   if (IS_PULSE_CYCLE) {
176     // Divide by two so raw pulse rate matches frequency
177     rate = pulseRate/2;
178     pulseRate = 0;
179   }
180
181   // only run on major cycle
182   if (IS_MAJOR_CYCLE) {
183     setTemp(mData->temperatureRaw);
184     setBloodPress(mData->systolicPressRaw, mData->diastolicPressRaw);
185
186       int prev = *(int*) cbGet(mData->pulseRateRaw);
187
188       // Only save if +- 15%
189       if (rate < prev*0.85 || rate > prev*1.15) {
190         cbAdd(mData->pulseRateRaw, (void *)&rate);
191       }
192     computeActive = true;    // run the compute task
193
194 #if DEBUG
195     char num[30];
```

```
196     int temp = *(int *)cbGet(mData->temperatureRaw);
197     int sys = *(int *)cbGet(mData->systolicPressRaw);
198     int dia = *(int *)cbGet(mData->diastolicPressRaw);
199     int pulse = *(int *)cbGet(mData->pulseRateRaw);
200     int batt = global.batteryState;
201 
202     usnprintf(num, 30, "Raw temp: %d  ", temp);
203     RIT128x96x4StringDraw(num, 0, 0, 15);
204 
205     usnprintf(num, 30, "Raw Syst: %d  ", sys);
206     RIT128x96x4StringDraw(num, 0, 10, 15);
207 
208     usnprintf(num, 30, "Raw Dia: %d  ", dia);
209     RIT128x96x4StringDraw(num, 0, 20, 15);
210 
211     usnprintf(num, 30, "Raw Pulse: %d  ", pulse);
212     RIT128x96x4StringDraw(num, 0, 30, 15);
213 
214     usnprintf(num, 30, "Raw Batt: %d  ", batt);
215     RIT128x96x4StringDraw(num, 0, 40, 15);
216 #endif
217   }
218 }
```

C.5.4 Compute Task

../code/compute.h

```
1  /*
2   * compute.h
3   * Author(s): PatrickMa
4   * 1/28/2014
5   *
6   * Defines the public interface for computeTask
7   * initializeComputeData() should be called before running computeTask()
8   */
9 
10 #include "task.h"
11 
12 /* Points to the TCB for compute */
13 extern TCB computeTask;
```

../code/compute.c

```
1  /*
2   * compute.c
3   * Author(s): PatrickMa
4   * 1/28/2014
5   *
6   * Implements compute.c
7   */
8 
```

```
9  #include "task.h"
10 #include "CircularBuffer.h"
11 #include "compute.h"
12 #include "globals.h"
13 #include "timebase.h"
14 #include "schedule.h"
15
16 // Used for debug display
17 #if DEBUG
18 #include "drivers/rit128x96x4.h"
19 #include "utils/ustdlib.h"
20 #endif
21
22 // computeData structure internal to compute task
23 typedef struct computeData {
24   // raw data pointers
25   CircularBuffer *temperatureRaw;
26   CircularBuffer *systolicPressRaw;
27   CircularBuffer *diastolicPressRaw;
28   CircularBuffer *pulseRateRaw;
29
30   //corrected data pointers
31   CircularBuffer *temperatureCorrected;
32   CircularBuffer *systolicPressCorrected;
33   CircularBuffer *diastolicPressCorrected;
34   CircularBuffer *pulseRateCorrected;
35 } ComputeData;
36
37 void computeRunFunction(void *computeData);
38
39 static ComputeData data;     // the internal data
40 TCB computeTask = {&computeRunFunction, &data};  // task interface
41
42 /*
43  * Initializes the computeData task values (pointers to variables, etc)
44  */
45 void initializeComputeTask() {
46   data.temperatureRaw = &(global.temperatureRaw);
47   data.systolicPressRaw = &(global.systolicPressRaw);
48   data.diastolicPressRaw = &(global.diastolicPressRaw);
49   data.pulseRateRaw = &(global.pulseRateRaw);
50
51   data.temperatureCorrected = &(global.temperatureCorrected);
52   data.systolicPressCorrected = &(global.systolicPressCorrected);
53   data.diastolicPressCorrected = &(global.diastolicPressCorrected);
54   data.pulseRateCorrected = &(global.pulseRateCorrected);
55 }
56
57 /*
58  * Linearizes the raw data measurement and converts value into human
59  * readable format
60  */
61 void computeRunFunction(void *computeData) {
```

```
62    static tBoolean onFirstRun = true;
63
64    if (onFirstRun) {
65      initializeComputeTask();
66      onFirstRun = false;
67    }
68
69    if (IS_MAJOR_CYCLE) {
70      ComputeData *cData = (ComputeData *) computeData;
71
72      float temp = 5 + 0.75 * (*(int*)cbGet(cData->temperatureRaw));
73      float systolic = 9 + 2 * (*(int*)cbGet(cData->systolicPressRaw));
74      float diastolic = 6 + 1.5 * (*(int*)cbGet(cData->diastolicPressRaw));
75      float pulseRate = 8 + 3 * (*(int*)cbGet(cData->pulseRateRaw));
76
77      cbAdd(cData->temperatureCorrected, &temp);
78      cbAdd(cData->systolicPressCorrected, &systolic);
79      cbAdd(cData->diastolicPressCorrected, &diastolic);
80      cbAdd(cData->pulseRateCorrected, &pulseRate);
81
82      computeActive = false;  // remove self from task queue
83
84 #if DEBUG
85      char num[30];
86      usnprintf(num, 30, "Corrected temp: %d", (unsigned int) temp);
87      RIT128x96x4StringDraw(num, 0, 50, 15);
88
89      usnprintf(num, 30, "Corrected Syst: %d", (unsigned int) systolic);
90      RIT128x96x4StringDraw(num, 0, 60, 15);
91
92      usnprintf(num, 30, "Corrected Dia: %d", (unsigned int) diastolic);
93      RIT128x96x4StringDraw(num, 0, 70, 15);
94
95      usnprintf(num, 30, "Corrected Pulse: %d", (unsigned int) pulseRate);
96      RIT128x96x4StringDraw(num, 0, 80, 15);
97 #endif
98    }
99 }
```

C.5.5 Keypad Task

../code/keypad.h

```
1 /*
2  * keyPad.h
3  * Author(s): Jarrett Gaddy
4  * 2/10/2014
5  *
6  * Defines the public interface for the keyPad task
7  *
8  * initializeKeyPadTask() should be called once before performing runkeyPad
9  * functions
```

```
 */

#include "task.h"  // for TCBs

/* Initialize KeyPadData, must be done before running functions */
void initializeKeyPadTask();

/* The keyPad Task */
extern TCB keyPadTask;
```

../code/keypad.c

```
/*
 * keyPad.c
 * Author(s): Jarrett Gaddy
 * 2/10/2014
 *
 * implements keyPad.h
 */

#include "keyPad.h"
#include "globals.h"
#include "timebase.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "inc/hw_memmap.h"

#define UP_SW GPIOPinRead(GPIO_PORTD_BASE, GPIO_PIN_4)
#define DOWN_SW GPIOPinRead(GPIO_PORTD_BASE, GPIO_PIN_5)
#define LEFT_SW GPIOPinRead(GPIO_PORTD_BASE, GPIO_PIN_6)
#define RIGHT_SW GPIOPinRead(GPIO_PORTD_BASE, GPIO_PIN_7)
#define ACK_SW GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_0)

// StatusData structure internal to compute task
typedef struct {
  unsigned short *mode;
  unsigned short *measurementSelection;
  unsigned short *scroll;
  tBoolean *alarmAcknowledge;
  tBoolean *select;
} KeyPadData;

void keyPadRunFunction(void *data);  // prototype for compiler

static KeyPadData data;  // the internal data
TCB keyPadTask = {&keyPadRunFunction, &data}; // task interface

/* Initialize the StatusData task values */
void initializeKeyPadTask() {

      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
```

```
42
43
44
45      GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
46        GPIO_PIN_TYPE_STD_WPU);
47      GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_DIR_MODE_IN);
48
49      GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_4, GPIO_STRENGTH_2MA,
50        GPIO_PIN_TYPE_STD);
51      GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_4, GPIO_DIR_MODE_IN);
52
53      GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_5, GPIO_STRENGTH_2MA,
54        GPIO_PIN_TYPE_STD);
55      GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_5, GPIO_DIR_MODE_IN);
56
57      GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_STRENGTH_2MA,
58        GPIO_PIN_TYPE_STD);
59      GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_DIR_MODE_IN);
60
61      GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_7, GPIO_STRENGTH_2MA,
62        GPIO_PIN_TYPE_STD);
63      GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_7, GPIO_DIR_MODE_IN);
64
65      //for ack switch
66      /* GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_STRENGTH_2MA,
67        GPIO_PIN_TYPE_STD_WPU);
68      GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_DIR_MODE_IN);*/
69
70
71
72    // Load data
73    data.mode = &(global.mode);
74    data.measurementSelection = &(global.measurementSelection);
75    data.scroll = &(global.scroll);
76    data.alarmAcknowledge = &(global.alarmAcknowledge);
77    data.select = &(global.select);
78
79    // Load TCB
80    keyPadTask.runTaskFunction = &keyPadRunFunction;
81    keyPadTask.taskDataPtr = &data;
82  }
83
84  /* Perform status tasks */
85  void keyPadRunFunction(void *keyPadData){
86
87    static tBoolean onFirstRun = true;
88
89    if (onFirstRun) {
90      initializeKeyPadTask();
91      onFirstRun = false;
92    }
93    KeyPadData *kdata = (KeyPadData *) keyPadData;
94    if( 0 == *(kdata->mode))
```

```
95    {
96
97      if(false == *(kdata->select))
98      {
99
100       if(UP_SW)
101         *(kdata->scroll) = *(kdata->scroll) +1;
102       else if(DOWN_SW)
103         *(kdata->scroll) = *(kdata->scroll) -1;
104       else if(RIGHT_SW)
105         *(kdata->select) = true;
106                           else if(LEFT_SW)
107                           {
108                               *(kdata->mode) = 1;
109                           }
110
111     }
112     else
113     {
114                       if(LEFT_SW)
115       *(kdata->select) = false;
116     }
117   }
118   else
119   {
120     if(RIGHT_SW)
121     {
122       *(kdata->mode) = 0;
123     }
124   }
125   if(1 == ACK_SW)
126   {
127     *(kdata->alarmAcknowledge) = true;
128   }
129   else
130     *(kdata->alarmAcknowledge) = false;
131 }
```

C.5.6 Display Task

../code/display.h

```
1  /*
2   * display.h
3   * Author(s): Jarrett Gaddy
4   * 1/28/2014, updated 2/10/2014
5   *
6   * Defines the interface for the displayTask.
7   * initializeDisplayData() should be called before running displayTask()
8   */
9
10 #include "task.h"
```

```
11
12 /* Initialize DisplayData, must be done before running displayTask() */
13 void initializeDisplayTask();
14
15 /* Points to the TCB for display */
16 extern TCB displayTask;
```

../code/display.c

```
1  /*
2   * display.c
3   * Author(s): jarrett Gaddy
4   * 2/10/2014
5   *
6   * Implements display.h
7   */
8
9  #include "globals.h"
10 #include "task.h"
11 #include "timebase.h"
12 #include "display.h"
13 #include "inc/hw_types.h"
14 #include "drivers/rit128x96x4.h"
15 #include "utils/ustdlib.h"
16 #include <stdlib.h>
17
18
19 // Internal data structure
20 typedef struct oledDisplayData {
21   CircularBuffer *temperatureCorrected;
22   CircularBuffer *systolicPressCorrected;
23   CircularBuffer *diastolicPressCorrected;
24   CircularBuffer *pulseRateCorrected;
25   unsigned short *batteryState;
26   unsigned short *mode;
27   unsigned short *measurementSelection;
28   unsigned short *scroll;
29   tBoolean *alarmAcknowledge;
30   tBoolean *select;
31   unsigned short *selection;
32 } DisplayData;
33
34 void displayRunFunction(void *dataptr);  // prototype for compiler
35
36 static DisplayData data;  // internal data
37 TCB displayTask = {&displayRunFunction, &data};          // task interface
38
39 void initializeDisplayTask() {
40   RIT128x96x4Init(1000000);
41
42   // Load data
43   data.temperatureCorrected = &(global.temperatureCorrected);
44   data.systolicPressCorrected = &(global.systolicPressCorrected);
```

```
45    data.diastolicPressCorrected = &(global.diastolicPressCorrected);
46    data.pulseRateCorrected = &(global.pulseRateCorrected);
47    data.batteryState = &(global.batteryState);
48
49
50    data.mode = &(global.mode);
51    data.measurementSelection = &(global.measurementSelection);
52    data.scroll = &(global.scroll);
53    data.alarmAcknowledge = &(global.alarmAcknowledge);
54    data.select = &(global.select);
55
56
57    // Load TCB
58    displayTask.runTaskFunction = &displayRunFunction;
59    displayTask.taskDataPtr = &data;
60  }
61
62
63  void displayRunFunction(void *dataptr) {
64    // only run on major cycle
65    //  if (IS_MAJOR_CYCLE) {   // on major cycle
66
67    static tBoolean onFirstRun = true;
68
69    if (onFirstRun) {
70      initializeDisplayTask();
71      onFirstRun = false;
72    }
73
74    DisplayData *dData = (DisplayData *) dataptr;
75
76    tBoolean selection = *(dData->select);
77    int scroll = *(dData->scroll);
78
79  #if !DEBUG
80    char num[40];
81    //char buf1[30];
82    char buf2[30];
83
84    if(0 == *(dData->mode))
85    {
86      if(false == selection)
87      {
88        RIT128x96x4StringDraw("Make Selection                    ", 0, 0,
       15);
89        RIT128x96x4StringDraw("  Blood Pressure                  ", 0, 10,
       15);
90        RIT128x96x4StringDraw("  Temperature                     ", 0, 20,
       15);
91        RIT128x96x4StringDraw("  Pulse Rate                      ", 0, 30,
       15);
92        RIT128x96x4StringDraw("  Battery                         ", 0, 40,
       15);
```

```
93        RIT128x96x4StringDraw("                                                    ", 0, 50,
      15);
94        RIT128x96x4StringDraw("                                                    ", 0, 60,
      15);
95        RIT128x96x4StringDraw("->", 0, 10*((scroll%4+1)), 15);
96      }
97      else
98      {
99        RIT128x96x4StringDraw("                                                    ", 0, 0,
     15);
100       if(0 == scroll%4)
101         RIT128x96x4StringDraw("Blood Pressure:", 0, 0, 15);
102       else if(1 == scroll%4)
103         RIT128x96x4StringDraw("Temperature:", 0, 0, 15);
104       else if(2 == scroll%4)
105         RIT128x96x4StringDraw("Pulse Rate:", 0, 0, 15);
106       else if(3 == scroll%4)
107         RIT128x96x4StringDraw("Battery:", 0, 0, 15);
108       else RIT128x96x4StringDraw("oops", 0, 0, 15);//just in case
109
110
111
112       if(0 == scroll%4)
113         usnprintf(buf2,30, "Systolic: %d mm Hg ", (int) *( (float*) cbGet(
     dData->systolicPressCorrected)));
114       else if(1 == scroll%4)
115         usnprintf(buf2,30,"%d C ", (int) *( (float*) cbGet(dData->
     temperatureCorrected)));
116       else if(2 == scroll%4)
117         usnprintf(buf2,30, "%d BPM ", (int) *( (float*) cbGet(dData->
     pulseRateCorrected)));
118       else if(3 == scroll%4)
119         usnprintf(buf2,30, "%d %% ", (int) *(dData->batteryState)/2);
120       //else buf2 = "oops"; //just in case
121
122       RIT128x96x4StringDraw("                                                    ", 0, 10,
      15);
123       RIT128x96x4StringDraw(buf2, 0, 10, 15);
124       if(0 == scroll%4)
125       {
126         usnprintf(buf2,30, "Diastolic: %d mm Hg         ", (int)*( (float*)
     cbGet(dData->diastolicPressCorrected)));
127
128         RIT128x96x4StringDraw(buf2, 0, 20, 15);
129       }
130       else RIT128x96x4StringDraw("                                                    ",
     0, 20, 15);
131       RIT128x96x4StringDraw("                                                    ", 0, 30,
      15);
132       RIT128x96x4StringDraw("                                                    ", 0, 40,
      15);
133       RIT128x96x4StringDraw("                                                    ", 0, 50,
      15);
```

```
134         RIT128x96x4StringDraw("                                    ", 0, 60,
        15);
135
136       }
137     }
138     else //(1 == *(data->mode)
139     {
140       usnprintf(num,40,"Temperature: %d C          ", (int) *( (float*) cbGet
        (dData->temperatureCorrected)));
141       RIT128x96x4StringDraw(num, 0, 0, 15);
142
143       usnprintf(num,40, "Systolic Pressure:          ");
144       RIT128x96x4StringDraw(num, 0, 10, 15);
145
146       usnprintf(num,40, "%d mm Hg                  ", (int) *( (float*) cbGet
        (dData->systolicPressCorrected)));
147       RIT128x96x4StringDraw(num, 0, 20, 15);
148
149       usnprintf(num,40, "Diastolic Pressure:         ");
150       RIT128x96x4StringDraw(num, 0, 30, 15);
151
152       usnprintf(num,40, "%d mm Hg                  ",(int) *( (float*)
        cbGet(dData->diastolicPressCorrected)));
153       RIT128x96x4StringDraw(num, 0, 40, 15);
154
155       usnprintf(num,40, "Pulse rate: %d BPM          ",(int) *( (float*)
        cbGet(dData->pulseRateCorrected)));
156       RIT128x96x4StringDraw(num, 0, 50, 15);
157
158       usnprintf(num,40, "Battery: %d %%              ",(int) *(dData->
        batteryState)/2);
159       RIT128x96x4StringDraw(num,0, 60,15);
160     }
161 #endif
162 }
```

C.5.7 Warning/Alarm Task

../code/warning.h

```
1  /*
2   * warning.h
3   * Author(s): Jarrett Gaddy
4   * 1/28/2014
5   *
6   * Defines the interface for the warning task
7   * initializeWarningData() should be called before running warningTask()
8   */
9
10 #include "task.h"
11
12 #define WARN_LOW 0.95 //warn at 5% below min range value
```

```
13 #define WARN_HIGH 1.05 //warn at 5% above max range value
14 #define ALARM_LOW 0.90 //alarm at 10% below min range value
15 #define ALARM_HIGH 1.20 //alarm at 20% above max range value
16
17
18 #define WARN_RATE_PULSE     4     // flash rate in terms of minor cycles
19 #define WARN_RATE_TEMP      2
20 #define WARN_RATE_PRESS     1
21
22 #define TEMP_MIN 36.1
23 #define TEMP_MAX 37.8
24 #define SYS_MAX 120
25 #define DIA_MAX 80
26 #define PULSE_MIN 60
27 #define PULSE_MAX 100
28 #define BATTERY_MIN 40
29
30 /* Initialize displayData, must be done before running warningTask() */
31 void initializeWarningTask();
32
33 /* The warning task */
34 extern TCB warningTask;
```

../code/warning.c

```
1  /*
2   * warning.c
3   * Author(s): jarrett Gaddy, PatrickMa
4   * 1/28/2014
5   *
6   * Implements warning.h
7   */
8
9  #include "globals.h"
10 #include "task.h"
11 #include "timebase.h"
12 #include "warning.h"
13 #include "schedule.h"
14 #include "inc/hw_types.h"
15 #include "drivers/rit128x96x4.h"
16 #include <stdlib.h>
17 #include <stdio.h>
18
19 #include "inc/hw_memmap.h"
20 #include "driverlib/debug.h"
21 #include "driverlib/gpio.h"
22 #include "driverlib/pwm.h"
23 #include "driverlib/sysctl.h"
24
25 #if DEBUG
26 #include "drivers/rit128x96x4.h"
27 #include "utils/ustdlib.h"
28 #endif
```

```
29
30 // alarm cycle period (on/off) in millisecond
31 #define ALARM_PERIOD 2000
32 #define ALARM_CYCLE_RATE     (ALARM_PERIOD / MINOR_CYCLE / 2)
33
34 // duration to sleep in terms of minor cycles
35 #define ALARM_SLEEP_PERIOD    (5 * MAJOR_CYCLE)
36
37 #define LED_GREEN GPIO_PIN_6
38 #define LED_RED   GPIO_PIN_5
39 #define LED_YELLOW GPIO_PIN_7
40
41 typedef enum {OFF, ON, ASLEEP} alarmState;
42 typedef enum {NONE, WARN_PRESS, WARN_TEMP, WARN_PULSE} warningState;
43 typedef enum {NORMAL, LOW} batteryState;
44
45 //pin E0 for input on switch 3
46 //pin C5 C6 and C7 for led out
47
48 // compiler prototypes
49 void warningRunFunction(void *dataptr);  // prototype for compiler
50
51 // Internal data structure
52 typedef struct WarningData {
53   CircularBuffer *temperatureCorrected;
54   CircularBuffer *systolicPressCorrected;
55   CircularBuffer *diastolicPressCorrected;
56   CircularBuffer *pulseRateCorrected;
57   unsigned short *batteryState;
58 } WarningData;
59
60 extern tBoolean serialActive;
61 static unsigned long ulPeriod; // sets the alarm tone period
62 static WarningData data;        // internal data
63
64 TCB warningTask = {&warningRunFunction, &data};  // task interface
65
66
67 /*
68  * initializes task variables
69  */
70 void initializeWarningTask() {
71   //
72   // Enable the peripherals used by this code. I.e enable the use of pin
      banks, etc.
73   //
74   SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
75   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);        // bank C
76   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);        // bank E
77   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);        // bank F
78   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);        // bank G
79
80
```

```
81  // configure the pin C5 for 4mA output
82  GPIOPadConfigSet(GPIO_PORTC_BASE,LED_RED, GPIO_STRENGTH_4MA,
      GPIO_PIN_TYPE_STD);
83  GPIODirModeSet(GPIO_PORTC_BASE, LED_RED, GPIO_DIR_MODE_OUT);
84
85  // configure the pin C6 for 4mA output
86  GPIOPadConfigSet(GPIO_PORTC_BASE,LED_GREEN, GPIO_STRENGTH_4MA,
      GPIO_PIN_TYPE_STD);
87  GPIODirModeSet(GPIO_PORTC_BASE, LED_GREEN, GPIO_DIR_MODE_OUT);
88
89  // configure the pin C7 for 4mA output
90  GPIOPadConfigSet(GPIO_PORTC_BASE,LED_YELLOW, GPIO_STRENGTH_4MA,
      GPIO_PIN_TYPE_STD);
91  GPIODirModeSet(GPIO_PORTC_BASE, LED_YELLOW, GPIO_DIR_MODE_OUT);
92
93  // configure the pin E0 for input (sw3). NB: requires pull-up to operate
94  GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
95      GPIO_PIN_TYPE_STD_WPU);
96  GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_DIR_MODE_IN);
97
98
99  /*  This function call does the same result of the above pair of calls,
100  *  but still requires that the bank of peripheral pins is enabled via
101  *  SysCtrlPeripheralEnable()
102  */
103 //  GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, LED_RED);
104
105 //////////////////////////////////////////
106 //  This section defines the PWM speaker characteristics
107 //////////////////////////////////////////
108
109 //
110 // Set the clocking to run directly from the crystal.
111 //
112 SysCtlPWMClockSet(SYSCTL_PWMDIV_1);
113
114 //
115 // Set GPIO F0 and G1 as PWM pins.  They are used to output the PWM0 and
116 // PWM1 signals.
117 //
118 GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0);
119 GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_1);
120
121 //
122 // Compute the PWM period based on the system clock.
123 //
124 ulPeriod = SysCtlClockGet() / 65;
125
126 //
127 // Set the PWM period to 440 (A) Hz.
128 //
129 PWMGenConfigure(PWM0_BASE, PWM_GEN_0,
130     PWM_GEN_MODE_UP_DOWN | PWM_GEN_MODE_NO_SYNC);
```

```
131   PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, ulPeriod);
132
133   //
134   // Set PWM0 to a duty cycle of 25% and PWM1 to a duty cycle of 75%.
135   //
136   PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, ulPeriod / 4);
137   PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ulPeriod * 3 / 4);
138
139   //
140   // Enable the PWM0 and PWM1 output signals.
141   //
142   PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT | PWM_OUT_1_BIT, true);
143
144   // initialize the warning data pointers
145   data.temperatureCorrected = &(global.temperatureCorrected);
146   data.systolicPressCorrected = &(global.systolicPressCorrected);
147   data.diastolicPressCorrected = &(global.diastolicPressCorrected);
148   data.pulseRateCorrected = &(global.pulseRateCorrected);
149   data.batteryState = &(global.batteryState);
150
151 }
152
153 //*********************************************************
154
155 /*
156  * Warning task function
157  */
158 void warningRunFunction(void *dataptr) {
159
160   static alarmState aState = OFF;
161   static warningState wState = NONE;
162   static batteryState bState = NORMAL;
163
164   static warningState prevState;
165   prevState = wState;
166
167   static int wakeUpAlarmAt = 0;
168
169   static tBoolean onFirstRun = true;
170
171   if (onFirstRun){
172     initializeWarningTask();
173     onFirstRun = false;
174   }
175
176
177   // Get measurement data
178   WarningData *wData = (WarningData *) dataptr;
179   float temp = *( (float*) cbGet(wData->temperatureCorrected));
180   float sysPress = *( (float*) cbGet(wData->systolicPressCorrected));
181   float diaPress = *( (float*) cbGet(wData->diastolicPressCorrected));
182   float pulse = *( (float*) cbGet(wData->pulseRateCorrected));
183   unsigned short battery = *(wData->batteryState);
```

```
184
185   // Alarm condition
186   if ( (sysPress > SYS_MAX*ALARM_HIGH) ) { //|| // Commented lines = lab2
187       // (temp < TEMP_MIN*ALARM_LOW || temp > (TEMP_MAX*ALARM_HIGH)) ||
188       // (diaPress > DIA_MAX*ALARM_HIGH) ||
189       // (pulse < PULSE_MIN*ALARM_LOW || pulse > PULSE_MAX*ALARM_HIGH) ) {
190
191     // Should only turn alarm ON if it was previously OFF.  If it is
192     // ASLEEP, shouldn't do anything.
193     if (aState == OFF) aState = ON;
194   }
195   else
196     aState = OFF;
197
198   // Warning Condition
199   if ( sysPress > SYS_MAX*ALARM_HIGH || diaPress > DIA_MAX*ALARM_HIGH )
200     wState = WARN_PRESS;
201   else if ( temp < TEMP_MIN*WARN_LOW || temp > TEMP_MAX*WARN_HIGH )
202     wState = WARN_TEMP;
203   else if ( pulse < PULSE_MIN*WARN_LOW || pulse > PULSE_MAX*WARN_HIGH )
204     wState = WARN_PULSE;
205   else
206     wState = NONE;
207
208   // Battery Condition
209   if (battery < BATTERY_MIN)
210     bState = LOW;
211
212   // Handle speaker, based on alarm state
213   static tBoolean pwmEnable = false;
214   switch (aState) {
215     case ON:
216
217       if (0 == (minor_cycle_ctr % ALARM_CYCLE_RATE)) { // toggle between on/
      off
218         if (pwmEnable)
219           PWMGenEnable(PWM0_BASE, PWM_GEN_0);
220         else
221           PWMGenDisable(PWM0_BASE, PWM_GEN_0);
222         pwmEnable = !pwmEnable;
223       }
224       break;
225     case ASLEEP:
226       PWMGenDisable(PWM0_BASE, PWM_GEN_0);
227       break;
228     default:  // OFF
229       PWMGenDisable(PWM0_BASE, PWM_GEN_0);
230
231       break;
232   }
233
234   // Handle warning cases
235   static int toggletime;
```

```
236   switch (wState) {
237     case WARN_PRESS:
238       GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);
239
240       if (wState != prevState) {
241         GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
242         toggletime = WARN_RATE_PRESS;
243       }
244       else if (0 == minor_cycle_ctr%toggletime) {
245         if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
246           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
247         else
248           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00);
249       }
250
251       break;
252     case WARN_TEMP:
253       GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);
254
255       if (wState != prevState) {
256         GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
257         toggletime = WARN_RATE_TEMP;
258       }
259       else if (0 == minor_cycle_ctr%toggletime) {
260         if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
261           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
262         else
263           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00);
264       }
265       break;
266     case WARN_PULSE:
267       GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);
268
269       if (wState != prevState) {
270         GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
271         toggletime = WARN_RATE_PULSE;
272       }
273       else if (0==minor_cycle_ctr%toggletime) {
274         if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
275           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
276         else
277           GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00);
278       }
279       break;
280     default:  // NORMAL
281       GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0xFF);
282       GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0x00);
283       break;
284   }
285
286   // activate the remote terminal task if in ANY warn or alarm state
287   if (NONE == wState && OFF == aState)
288     serialActive = false;
```

```
289     else
290       serialActive = true;
291
292     // battery state indicator
293     if (bState == LOW) {
294       GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0xFF);
295       GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0x00);
296     }
297     else
298       GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0x00);
299
300     /* This is the alarm override
301      * Upon override, the alarm is silenced for some time.
302      * silence length is defined by ALARM_SLEEP_PERIOD
303      *
304      * If the button is pushed, the value returned is 0
305      * If the button is NOT pushed, the value is non-zero
306      */
307     if ( ( 0 == global.alarmAcknowledge) && (aState == ON) )
308     {
309       //GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0XFF);  // for debug, lights
          led
310       aState = ASLEEP;
311       wakeUpAlarmAt = minor_cycle_ctr + ALARM_SLEEP_PERIOD;
312     }
313
314     // Check whether to resound alarm
315     if (minor_cycle_ctr == wakeUpAlarmAt && aState == ASLEEP) {
316       aState = ON;
317       //GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0X00);  // for debug, kills
        led
318     }
319
320 #if DEBUG
321       char num[30];
322
323
324       usnprintf(num, 30, "Cor temp: %d  ", (int) temp);
325       RIT128x96x4StringDraw(num, 0, 0, 15);
326
327       usnprintf(num, 30, "Cor Syst: %d  ", (int) sysPress);
328       RIT128x96x4StringDraw(num, 0, 10, 15);
329
330       usnprintf(num, 30, "Cor Dia: %d  ", (int) diaPress);
331       RIT128x96x4StringDraw(num, 0, 20, 15);
332
333       usnprintf(num, 30, "Cor Pulse: %d  ", (int) pulse);
334       RIT128x96x4StringDraw(num, 0, 30, 15);
335
336       usnprintf(num, 30, "Cor Batt: %d  ", (unsigned short) battery);
337       RIT128x96x4StringDraw(num, 0, 40, 15);
338
339
```

```
340     usnprintf(num, 30, "aState: %d  ", aState);
341     RIT128x96x4StringDraw(num, 0, 50, 15);
342
343     usnprintf(num, 30, "alarmAck: %d  ", global.alarmAcknowledge);
344     RIT128x96x4StringDraw(num, 0, 60, 15);
345
346     usnprintf(num, 30, "pwmEn: %d  ", pwmEnable);
347     RIT128x96x4StringDraw(num, 0, 70, 15);
348 #endif
349 }
```

C.5.8 Serial Task

../code/serial.h

```
1  /*
2   * serial.h
3   * Author(s): Jonathan Ellington
4   * 2/05/2014
5   *
6   * Defines the serial communications task.  This sends various data over
7   * RS-232.
8   *
9   * This function should only run if there is a warning (in other words, the
10  * scheduler should add it to the task queue in the event of a warning), and
11  * should subsequently delete itself from the task queue.
12  */
13
14 #include "task.h"  // for TCBs
15
16 /* The status Task */
17 extern TCB serialTask;
```

../code/serial.c

```
1  /*
2   * serial.c
3   * Author(s): Jonathan Ellington
4   * 2/05/2014
5   *
6   * Implements serial.c
7   */
8
9  #include "task.h"
10 #include "globals.h"
11 #include "timebase.h"
12 #include "serial.h"
13 #include "schedule.h"
14 #include "CircularBuffer.h"
15 #include "inc/hw_types.h"
16 #include "driverlib/uart.h"
17 #include "driverlib/gpio.h"
```

```
18 #include "inc/hw_memmap.h"
19 #include "utils/ustdlib.h"
20 #include <string.h>
21
22
23 // Internal data structure
24 typedef struct serialData {
25   CircularBuffer *temperatureCorrected;
26   CircularBuffer *systolicPressCorrected;
27   CircularBuffer *diastolicPressCorrected;
28   CircularBuffer *pulseRateCorrected;
29   unsigned short *batteryState;
30 } SerialData;
31
32 // Prototype
33 void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount);
34 void serialRunFunction(void *dataptr);
35
36 static SerialData data;  // internal data
37 TCB serialTask = {&serialRunFunction, &data};          // task interface
38
39 void initializeSerialTask() {
40   // Initialize Data
41   data.temperatureCorrected = &(global.temperatureCorrected);
42   data.systolicPressCorrected = &(global.systolicPressCorrected);
43   data.diastolicPressCorrected = &(global.diastolicPressCorrected);
44   data.pulseRateCorrected = &(global.pulseRateCorrected);
45   data.batteryState = &(global.batteryState);
46
47   // UART Stuff
48   // Enable the peripherals used by this example.
49   SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
50   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
51
52   // Set GPIO A0 (UART RX)
53   GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_1);
54
55   // Configure the UART for 115,200, 8-N-1 operation.
56   UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
57       (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
58        UART_CONFIG_PAR_NONE));
59
60   UARTEnable(UART0_BASE);
61 }
62
63 void serialRunFunction(void *dataptr) {
64   static tBoolean onFirstRun = true;
65   SerialData *sData = (SerialData *) dataptr;
66   if (onFirstRun) {
67     initializeSerialTask();
68     onFirstRun = false;
69   }
70
```

```
71    int temp = (int) *((float *)cbGet(sData->temperatureCorrected));
72    int sys = (int) *((float *)cbGet(sData->systolicPressCorrected));
73    int dia = (int) *((float *)cbGet(sData->diastolicPressCorrected));
74    int pulse = (int) (*(float *)cbGet(sData->pulseRateCorrected));
75    int batt = *(data.batteryState);
76
77    char buf[512];
78  //  usnprintf(buf, 512,
79  //            "\f1. Temperature:\t\t%d C\n\n\r"
80  //            "2. Systolic pressure:\t%d mm Hg\n\n\r"
81  //            "3. Diastolic pressure:\t%d mm Hg\n\n\r"
82  //            "4. Pulse rate:\t\t%d BPM\n\n\r"
83  //            "5. Battery:\t\t%d\n\n\r",
84  //           temp, sys, dia, pulse, batt);
85
86    // The cast removes a warning.  It is safe as long as buf contains
87    // plain ASCII (non extended)
88    UARTSend( (unsigned char *) buf, strlen(buf));
89
90    serialActive = false;
91  }
92
93  void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount) {
94    // Loop while there are more characters to send.
95    while(ulCount--)
96    {
97      // Write the next character to the UART.
98      // This blocks while the FIFO queue is full
99      UARTCharPut(UART0_BASE, *pucBuffer++);
100   }
101 }
```

C.5.9 Status

../code/status.h

```
1  /*
2   * status.h
3   * Author(s): PatrickMa
4   * 1/28/2014
5   *
6   * Defines the public interface for the status task
7   *
8   * initializeStatusTask() should be called once before performing status
9   * functions
10  */
11
12 #include "task.h"  // for TCBs
13
14 /* Initialize StatusData, must be done before running functions */
15 void initializeStatusTask();
16
```

```
17 /* The status Task */
18 extern TCB statusTask;
```

../code/status.c

```
1  /*
2   * status.c
3   * Author(s): PatrickMa
4   * 1/28/2014
5   *
6   * implements status.h
7   */
8
9  #include "status.h"
10 #include "globals.h"
11 #include "timebase.h"
12
13 // StatusData structure internal to compute task
14 typedef struct {
15   unsigned short *batteryState;
16 } StatusData;
17
18 void statusRunFunction(void *data);  // prototype for compiler
19
20 static StatusData data;  // the internal data
21 TCB statusTask = {&statusRunFunction, &data}; // task interface
22
23 /* Initialize the StatusData task values */
24 void initializeStatusTask() {
25   // Load data
26   data.batteryState = &(global.batteryState);
27
28   // Load TCB
29   statusTask.runTaskFunction = &statusRunFunction;
30   statusTask.taskDataPtr = &data;
31 }
32
33 /* Perform status tasks */
34 void statusRunFunction(void *data){
35   static tBoolean onFirstRun = true;
36
37   if (onFirstRun) {
38     initializeStatusTask();
39     onFirstRun = false;
40   }
41
42   if (IS_MAJOR_CYCLE) {
43     StatusData *sData = (StatusData *) data;
44       if (*(sData->batteryState) > 0)
45         *(sData->batteryState) = *(sData->batteryState) - 1;
46   }
47 }
```

C.6 Circular Buffer

../code/CircularBuffer.h

```
/* CircularBuffer.h
 * Jonathan Ellington
 * 2/7/14
 */

/* A circular buffer implementation.
 * Meant to work without any dynamically allocated memory by wrap()ing
 * user defined arrays.
 *
 * NOTE: This implementation keeps NO type information.  This means the
 *       USER IS RESPONSIBLE FOR KEEPING TRACK OF THE TYPE OF ARRAY THAT
 *       HAS BEEN WRAPPED!
 *
 * NOTE: The implementation here should also be hidden, but this is
 *       troublesome without dynamic memory.  Do not rely on any of these
 *       elements!
 */

#ifndef _CIRCULAR_BUFFER_H
#define _CIRCULAR_BUFFER_H

typedef struct _circBuf {
  void *array;
  int sizeElm;
  int nElm;
  int currElm;
} CircularBuffer;

/* Wrap an array in a circular buffer.
 *
 * @param arr       the array to wrap
 * @param sizeElem  the size of each element, in bytes
 * @param nElem     the number of elements in the array
 *
 * This function expects the array is freshly created.  It will overwrite
 * array contents on adds.
 */
CircularBuffer cbWrap(void *arr, int sizeElem, int nElem);

/* Returns a pointer to the current element in the circular buffer
 * Be sure not to clobber this value! */
void *cbGet(CircularBuffer *cb);

/* Returns a pointer to the wrapped array, with the oldest element
 * at the end and the newest element at the beginning */
void *cbGetArray(CircularBuffer *cb);

/* Adds elem to cb */
void cbAdd(CircularBuffer *cb, void *elem);
```

```
50
51 #endif // _CIRCULAR_BUFFER_H
```

../code/CircularBuffer.c

```
1  /* CircularBuffer.c
2   * Jonathan Ellington
3   * 2/7/14
4   */
5
6  #include "CircularBuffer.h"
7  #include <stdio.h>
8
9  CircularBuffer cbWrap(void *arr, int se, int ne) {
10   CircularBuffer cb;
11   cb.array = arr;
12   cb.sizeElm = se;
13   cb.nElm = ne;
14   cb.currElm = 0;
15
16   return cb;
17 }
18
19 /* Returns the current elment in the circular buffer */
20 void *cbGet(CircularBuffer *cb) {
21   int sizeElm = cb->sizeElm;
22   int index = cb->currElm;
23
24   unsigned char *bytePtr = (unsigned char *)cb->array;
25   bytePtr += sizeElm * index;
26
27   return (void *) bytePtr;
28 }
29
30 /* Returns a pointer to the wrapped array, with the oldest elment
31  * at the end and the newest elment at the beginning */
32 void *cbGetArray(CircularBuffer *cb) {
33   return NULL;
34 }
35
36 /* Adds elm to cb */
37 void cbAdd(CircularBuffer *cb, void *elm) {
38   cb->currElm = (cb->currElm + 1) % cb->nElm;
39
40   int sizeElm = cb->sizeElm;
41   int index = cb->currElm;
42
43   // copy sizeElm bytes from elm into the right spot
44   unsigned char *elmPtr = (unsigned char *) elm;
45   unsigned char *arrElmPtr = (unsigned char *) cb->array;
46   arrElmPtr += sizeElm * index;
47
48   for (int i = 0; i < sizeElm; i++) {
```

```
49        arrElmPtr[i] = elmPtr[i];
50    }
51 }
```