EE 472 Lab 4
Learning the Development Environment - The Next Next Step

Jonathan Ellington
Patrick Ma
Jarrett Gaddy

Contents

List of Tables

List of Figures

1 ABSTRACT

In this lab the students are to take on the role of an embedded system design team. They will design 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 visual display of the sensor values
- Accept variety of input data types
- Provide visual indication of warning states
- Provide 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 performance
- Improve overall system safety

In Phase 3, the following requirements were added:

  - Implement a Real time operating system
  - Utilize an on-chip thermometer for raw temperature readings
  - Display user-selected sensor readings only
  - Implement support for EKG sensor readings

In Phase 4, the following requirements were added:

  - Implement a web server and web-based user interface

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.

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.

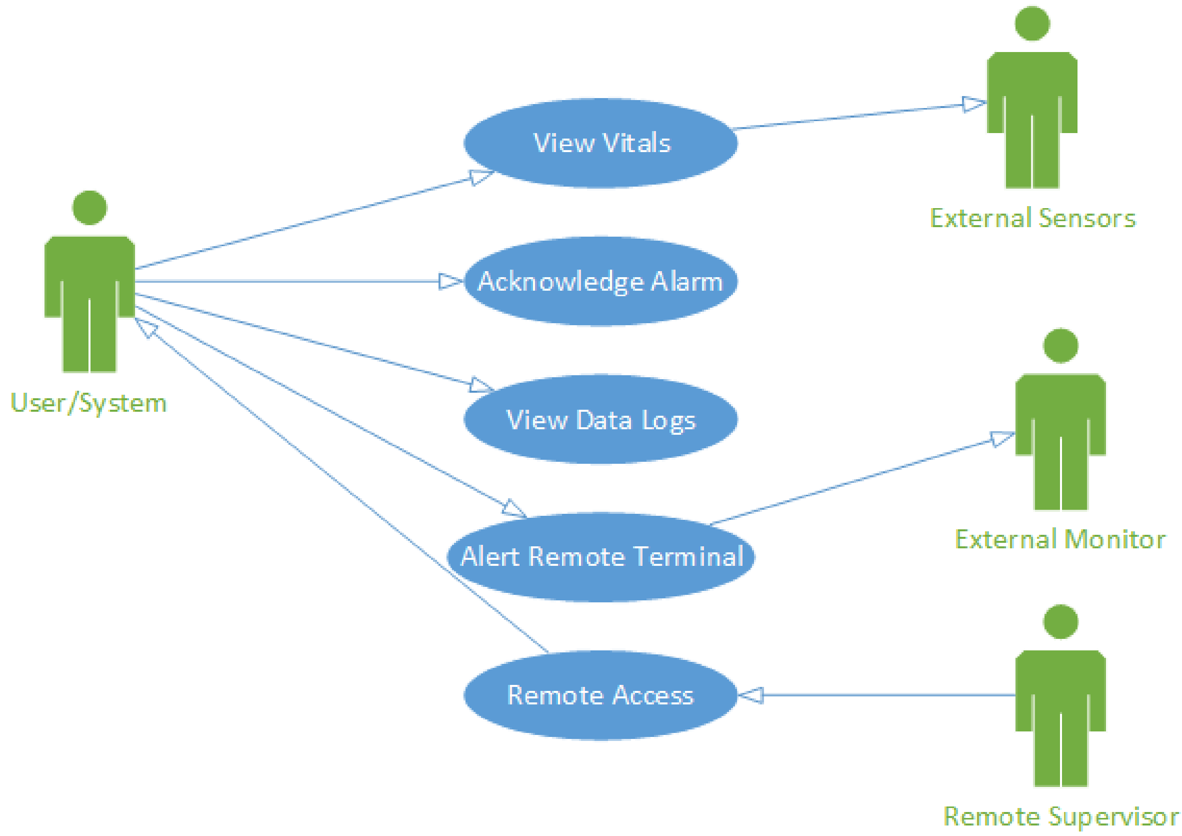


Figure 1: Use case diagram

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

Identified exceptions to use case:

- *Alarm is never acknowledged* - If the user never acknowledges an alarm, the system maintains the audible alarm.
- *User continually presses acknowledge button* - The system will only check for a key press at the end of the silencing period. Therefore, the alarm will be silenced if the button is pressed continuously.
- *No speaker or auditory device present* - The auditory alarm is produced via an onboard speaker. If this speaker is removed, no additional audible alarm is supported. This may change in future implentations.

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.

Use Case # 5: Remote Access via Network
This device may also be used by an individual at a location outside of normal doctor or medical facilties.

In these cases, a medical professional may be unable to provide direct intervention. Instead, the patient can be monitored remotely from the device through the use of an Internet connection. The remote observer should interact with the device through a terminal just as though they were using the local keypad. Such interactions include starting and stopping measurements, changing measurement selections, and viewing recent measured values.

Possible exception conditions may include:

- *Loss of Network Connection* - The nature of Internet carries the possibility of an unexpected loss of connection. In these instances, information sent from the device or remote terminal may become lost. Various network protocols exist which address and handle these issues. However, the device cannot rely on the remote terminal for operation. In the event data is lost, the local system must continue to operate according to the last received commands or from subsequent local commands.

- *Corruption of Data* - In a remote setting, many environmental variables may affect connection quality. Data sent to and from the device may be corrupted upon reception. The system must be capable of handling corrupted data. In the case of undecipherable commands, the system must fail gracefully.

- *Improperly Formatted Command* - The remote user may send a command that is unsupported by the system. In this case, the user must be informed that an improper command was sent.

3.1.3 Detailed Specifications

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

The system must have the following inputs:

- Alarm acknowledgment capability using a push button
- Buttons or switches to allow user to access system modes and menu items
- Ethernet and webserver to process incoming data requests
- Sensor measurement input capabilities consisting of:
  - Body temperature measurement
  - Pulse rate measurement signal
  - Systolic blood pressure measurement
  - Diastolic blood pressure measurement
  - EKG frequency 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
  - EKG Frequency
- Visual indication of warning state with a flashing LED
- Visual indication of a low battery state with an LED
- Audible indication of an alarm state using a speaker
- External data connection to a remote terminal
- Internet connectivity over an Ethernet connection

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.

A measurement enters a warning state when its value falls outside the stated normal range by 5%.

An alarm state occurs when a measured value falls outside of its specified normal range by more than 15%.

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.

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
EKG Frequency	Hz	-	-	-	-
Remaining Battery	%	200	40 %	-	Constant

Table 1: Specifications for measurement data

3.1.4 Detailed Task Specifications

- **KeypadTask** The keypad allows the user to interact with the system locally. The user can acknowledge alerts and navigate the menu options.
  - 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:
    - Measurement Select Menu
    - Annunciation
  - Menu selection between 6 options:
    - Scan all or measure a specific sensor (Temperature, Blood Pressure, Pulse Rate, or EKG)
  - Alarm acknowledgement
  - Up and down scroll functionality
  - A new set of global variables will be created to store the state of the keypad and key presses
- **Initialize (StartupTask):** The startup task sets up the system hardware when the device is first powered on.
  - The Startup task must be the first task to run
  - It must not be part of the task queue and must only run once
  - The task must configure and activate the system timebase
  - Configure and initialize all hardware subsystems
  - Enable any necessary interrupts
  - Assign priorities to each task prior to creation of the task queue
- **Serial Communication:** Serial communication produces an alert via RS-232 connection to a remote terminal. The connection is strictly uni-directional, serving to alert the remote station that some aspect of the system is in a warning or alarm state.

  There are no changes to the Serial Communication task since lab 3.
  - The task is enabled by the warn/alarm task

- When run, the task will open an RS-232 connection at 115,200 baud, no flow control, no parity, and 1 stop bit
- The present corrected measurement will be displayed on the terminal in the same fashion as the display task annunciation mode. See Figure 2
- After sending data to the terminal, the serial communication task will remove itself from the task queue

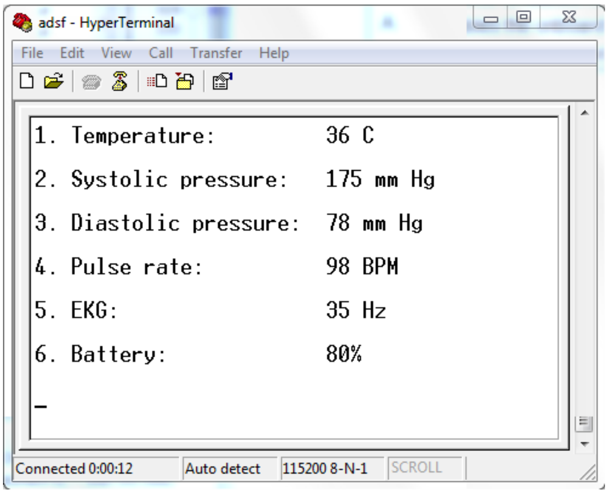


Figure 2: Expected screen layout of remote serial terminal

- **MeasureTask:** The measure function performs measurement collection of the various sensors attached to the system. The current sensors are a thermometer, blood pressure sensor, and heart rate monitor.

  No new changes were made to the measure task

  - Measurements must be captured every 5 seconds
  - 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.

- **ComputeTask:** The compute task performs numerical calculations to linearize and convert the raw sensor measurements into human-readable values.

  No new changes were made to the compute task

  - The following measurements will be recomputed each time the compute task is run
    1. Temperature

2. Systolic blood pressure
3. Diastolic blood pressure
4. Pulse rate
5. EKG frequency

  - This task must only be scheduled when new data is available from the Measure Task
  - The Compute Task must apply any linearization corrections to the raw data
  - The Compute task will store the computed and corrected values into a computed data buffer
  - After computing the corrected values for all measurements, the ComputeTask will remove itself from the task queue

- **DisplayTask:** In the Display task, data collected by the system is displayed on the local OLED display based upon user input received from the keypad. In addition to simply displaying data, the display task will format the data and present a series of menus for the user. The front panel of the OLED display in the Annunciation state is shown in Figure 3a

  No new changes were made to the display task

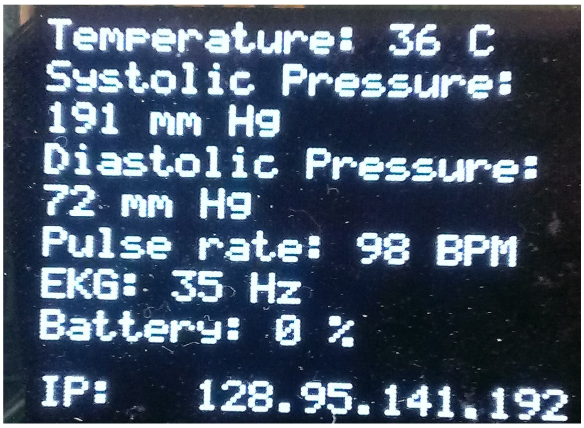
  - Display must support multiple display options
  - Menu mode will allow selection of each of the individual measurements. Upon selection of a measurement, the current value of the measurement will be displayed onscreen
  - Annunciation mode will display the current status of each measurement as in project 1, and provide the same functionality as the display in project 1.
  - The display screen must appear similar to those given in Figure 3

- **Warn/AlarmTask:** The warn/alarm task performs system monitoring functions, creating alerts when measured values exceed the normally accepted ranges. No changes were made to this task in Phases 3 or 4

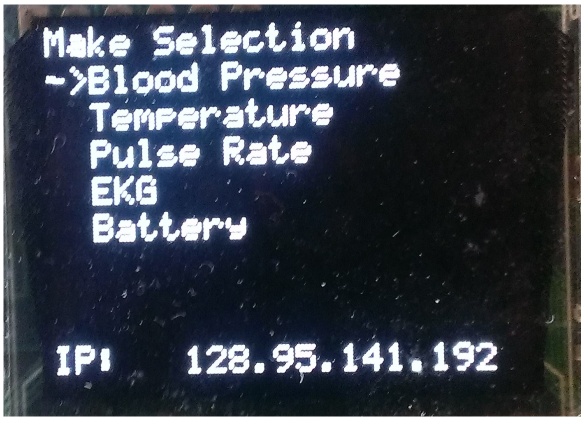
  - The warnings will be activated and indicated as before in project 1
  - The alarm state will be triggered whenever any value is outside 15% of the normal range
  - The alarm will sound in 1 second tones (1 second on, 1 second off)
  - When an alarm or warning state occurs, the serial communication task will be added to the task queue
  - The deactivation period of the alarm sound is defined as 5 measurement periods

- **Schedule:** The Schedule function has been replaced by the RTOS scheduler. Just as before, the Scheduler manages the order in which tasks are executed. It manages resources based on the current running task needs and guarantees tasks complete within the desired timeframe.

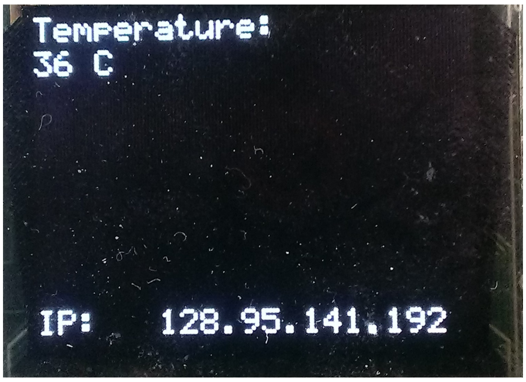
  - The Scheduler will provide non-premptive priority-based scheduling



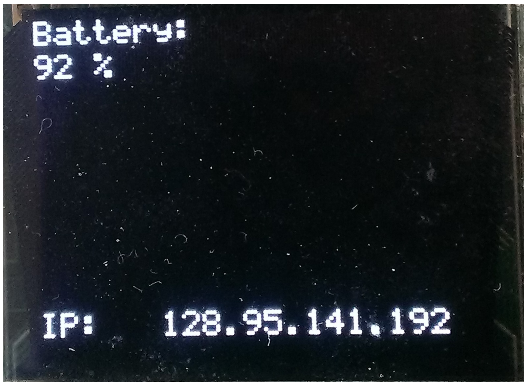
(a) Annunciation mode



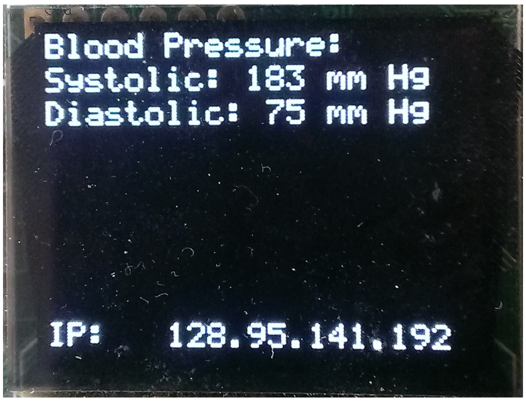
(b) Menu Select mode



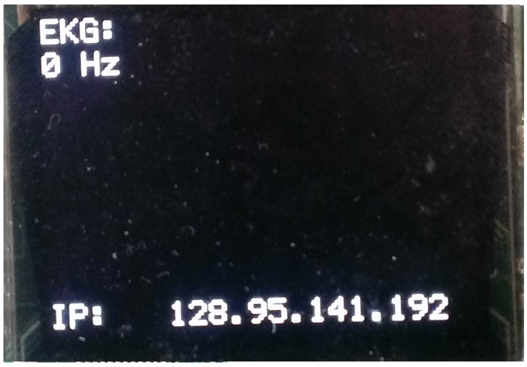
(c) Temperature measurement



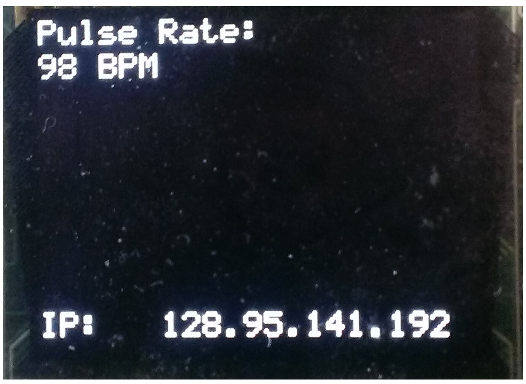
(d) Battery status



(e) Blood pressure measurement



(f) EKG measurement



(g) Pulse rate measurement

Figure 3: Expected OLED display screens

- The hardware timer will provide a system interrupt every 250ms or equal to the minor cycle, whichever is shorter
- Task Control Blocks will interact with other tasks via flags and communication buffers
- Tasks will run at least once every five seconds
- The scheduler cannot block for five seconds

- **StatusTask:** This task monitors the status of the onboard battery.
  - Each time the status task is called, the current amount of battery charge left is queried and stored
- **New Task: EKG Capture** Together EKG Capture and EKG Process will receive EKG sensor information and convert it to human-usable data for analysis
  - The EKG Capture function must convert a time-varying sinusoidal signal to a digital format
  - Each sample must range from 0-3Vdc
  - A total of 256 samples per sequence must be taken at 8-bit precision
  - The capture function must be capable of discerning inputs signals ranging from 35Hz to 3.75kHz
  - Once a sequence is complete, the EKG Process function must be called
- **New Task: EKG Process** The EKG Process function transforms the raw EKg sampled sequence and converts it to a single frequency for analysis.
  - The EKG Process function must perform a Fast Fourier Transform (FFT) on the raw data
  - Following conversion to the frequency domain, the FFT result must be converted to the expected frequency range
  - The 16 most recent EKG results must be stored in a buffer for transmission or review
- **New Task: Command** When interacting with the device remotely, the command task serves as the interpreter of console/browser commands. It conveys system information to the browser in response to webserver requests.
  - Command must be scheduled whenever a command has been received by the web server or when a message must be transmitted to the remote computer.
  - Receive mode:
    * A received command must be interpreted and acted upon if valid
    * A response acknowledgement or non-acknowledgement must be returned to the webserver whenever a command is received
    * Legal commands are given in Appendix B
  - Messages to be sent to a remote computer must be compiled into proper HTML code before being sent to the web server

- **New Task: Remote Communications** Remote communications, or the web server, operates as the gateway between the device and the wider internet. It handles communications protocols and incoming requests to the system.
  - The remote communication task must initialize the network interface
  - This task must also connect to and configure a local area network, handling TCP/IP, HTTP, and other protocols
  - Set up a webserver and webpage to act as a gateway for remote users
  - Update the webpage as needed
  - The user interface must accept a textual command entered by the user as defined in Appendix B

  The web browser user interface must contain a text input box and display information in a manner consistent with Figure 4.

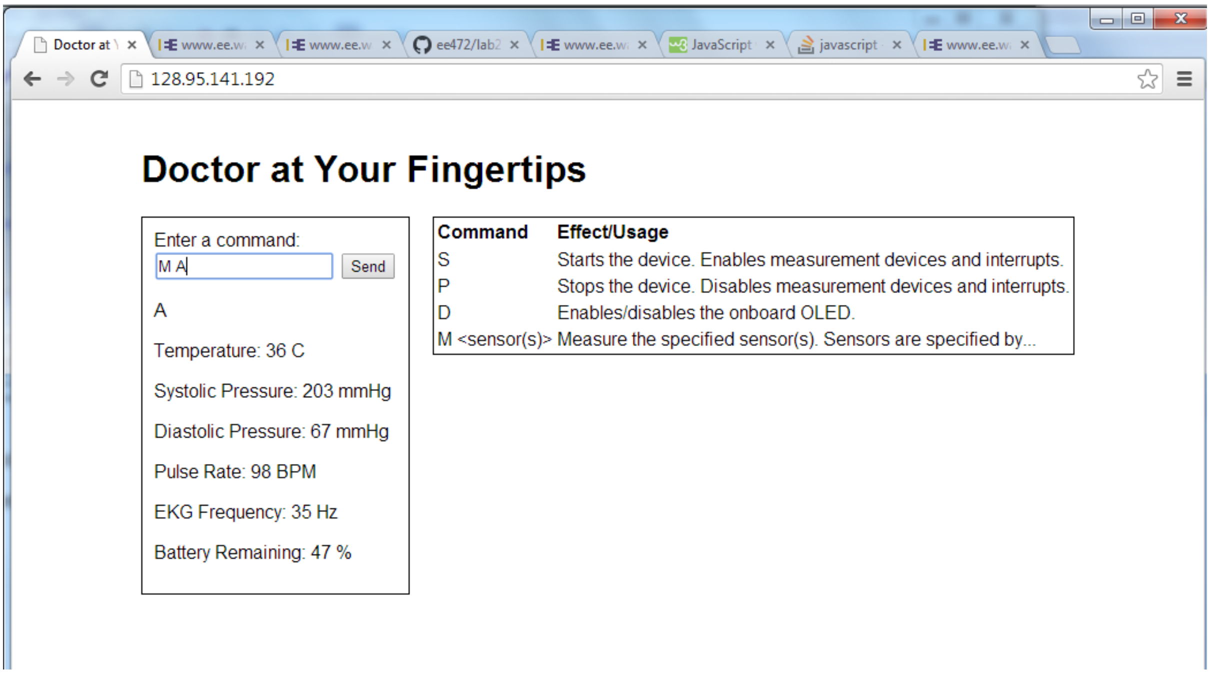


Figure 4: Expected Web Browser Interface

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 5.

The functional blocks were then refined further, showing the main functions the system needed to perform. This refined functional decomposition is shown in Figure 6.

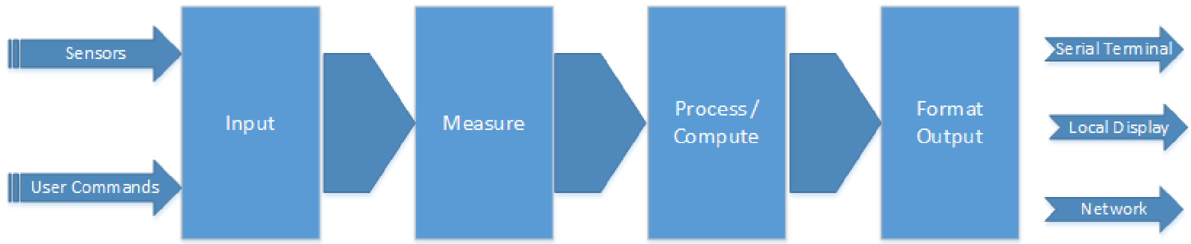


Figure 5: Top level system functionality

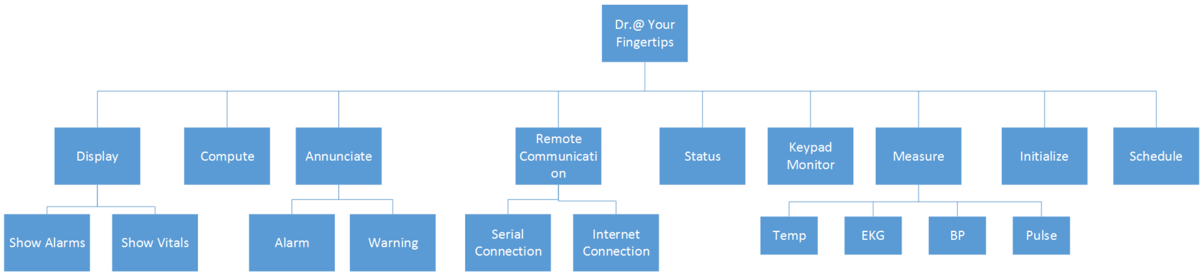


Figure 6: Low Level Functional Decomposition

3.2.2 System Architecture

Next, the system architecture was developed (Figure 7). At a high level the system works on two main concepts, the scheduler and tasks. In this project, FreeRTOS was used as the scheduler. 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. Under FreeRTOS, each task is given a priority, and tasks that are waiting to execute are started based on their priority. 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 7 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.

EKG Capture This task is in charge of getting raw EKG data. This data will be simulated. A sample frequency is defined (based on the specified min and max frequencies and the Nyquist theorem), and an ADC is triggered at the same speed as the sample frequency. The input to the ADC will be a sine wave of variable frequency, from 37 Hz to 3750 Hz. Because of this, the sample frequency chosen was 8000 Hz (more than twice the max frequency).

EKG Process This task is in charge of processing raw EKG data. It will run a FFT over the raw EKG data. The results of the FFT operation are then used to determine the maximum frequency in the signal sampled. These data are stored in system memory.

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. The monitor updates the local visual display and updates internal system states in response to mirror any selections or movements by the user

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.

Serial 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. The displayed data will be continually updated until the system returns to a normal condition. Once in a normal condition, the last non-normal state is displayed.

Command Task The command task should parse commands from the remote communications task and send a response. The response is based on the particular command sent. For example, if the user requests to view all measurements, the response should be an HTML formatted response containing all of the data for each measurement. Each response should also contain an ACK or NACK, depending on whether or not the command was valid or not.

Remote Communications Remote Communications is not actually a task. Instead, the http server, httpd, that is shipped along with the TI embedded development kit was used. This server is interrupt driven and runs entirely in the background. It communicates with tasks using buffers.

Status Task Status receives information about the battery on the system and updates its current data accordingly.

Each of these tasks interact using the shared data shown in Figure 7.

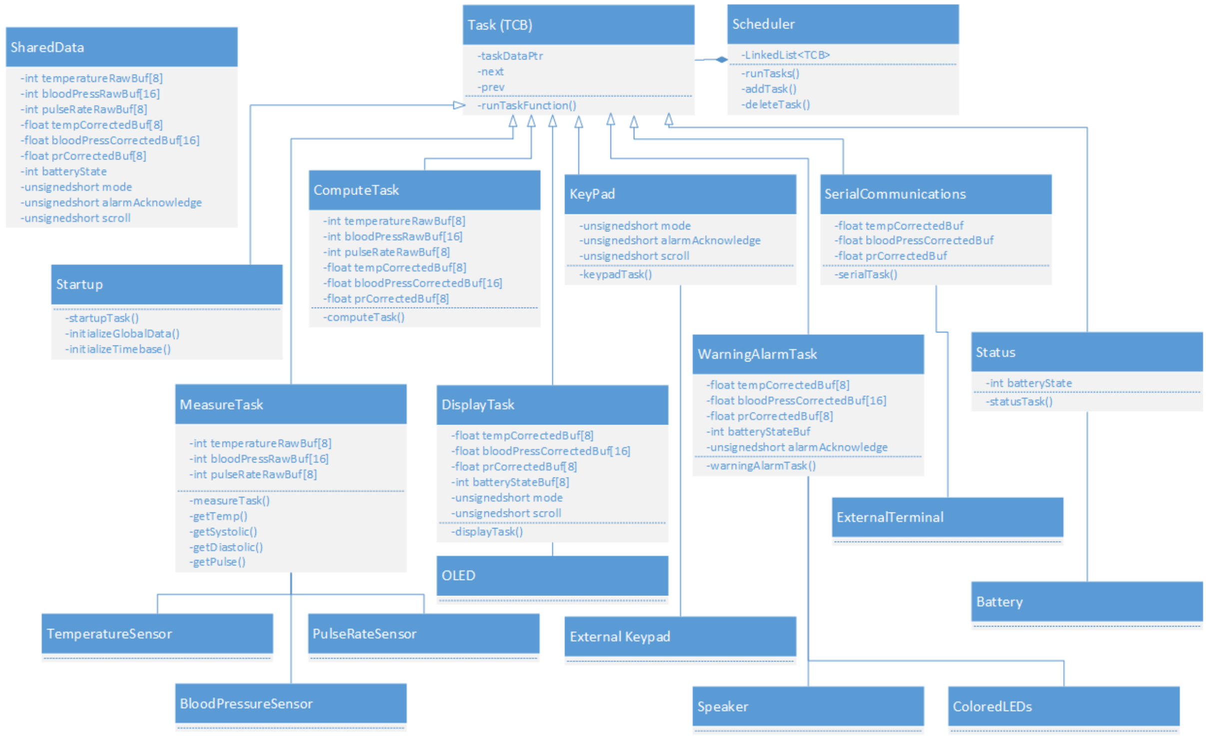


Figure 7: System Architecture Diagram

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
- **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 shown in Listing 1.

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

Listing 1: TCB Construct

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 8.

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 9 in Appendix C), 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

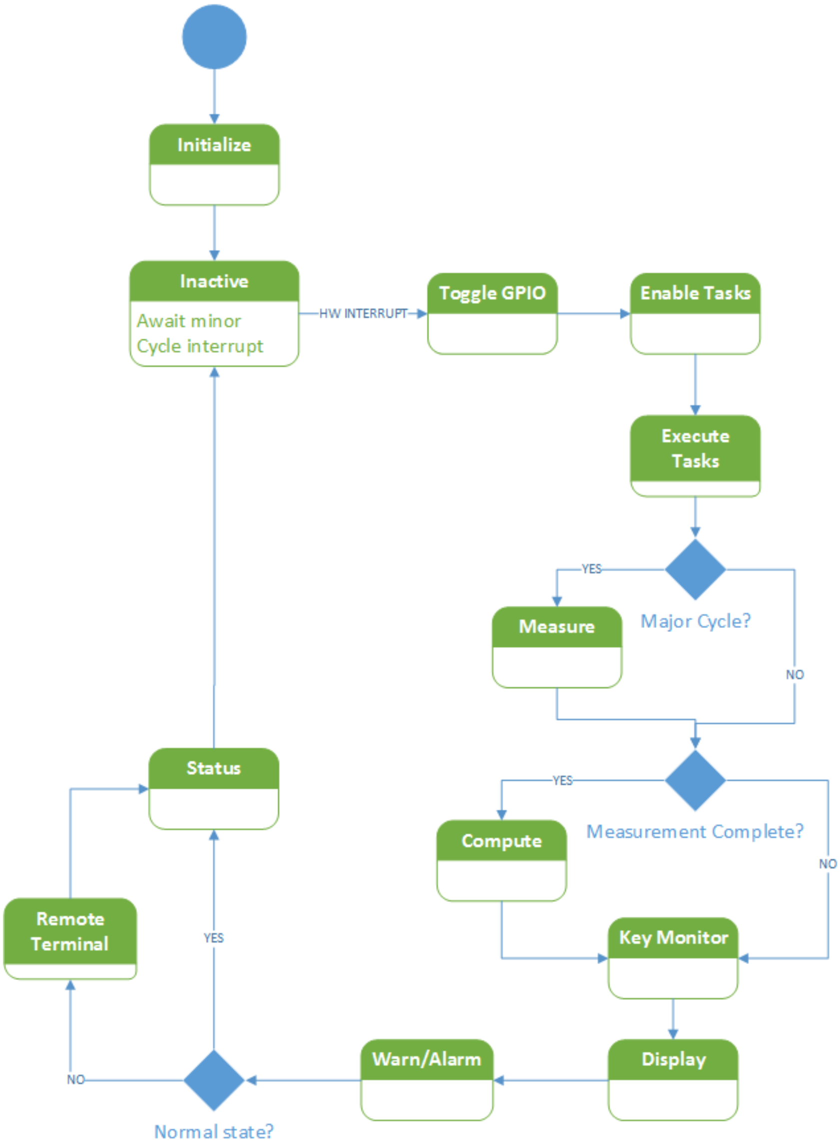


Figure 8: Control Flow Diagram

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 in Figure 10 of Appendix C

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 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 12 in Appendix C. 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 Figure 11 in Appendix C.

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 14 in Appendix C. 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 Appendix C as Figure 13.

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 15 in Appendix C.

4 PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS

4.1 Results

The project was completed and demonstrated on March 17, 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 the serial communication task, serialTask. After initially performing as expected, the system would freeze up. When this happened, the system was unresponsive and did not produce any annuniation.

Using an oscilloscope, the run times of each task were empirically determined. The procedure used involved asserting a GPIO pin high immediately after entering the function in question and deasserting the pin just prior to exiting the function. The trace of the pin output helped determine the exact function timing. The results are given in Table 2

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.

Task	Runtime (μs)
Measure	20.3
Compute	55.4
Display	33200
Warning	17.7
Serial	12000
keypad	5.72
ekgCapture	26600
ekgProcess	1345
Status	5.6

Table 2: Empirically determined task runtimes

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

Task	Instructions
Measure	162
Compute	443
Display	265600
Warning	142
Serial	96,000
keypad	45.8
ekgCapture	212,800
ekgProcess	10,760
Status	45

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

of the cycles are likely spent waiting for memory. For example, the status task only has two comparisons and an arithmetic operation, but has to reference the data in global memory. The exception here is the display task, which was about three orders of magnitude more instructions 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 33.2 ms was not significant.

4.3 Analysis of Any Errors

There were two errors found in the final project. Both errors involved a lack of accuracu in the reported values measured by the sensors. The pulse rate measurement was off in the same manner as in Lab 3. The other measurement, from the EKG sensor, was off by a similarly small amount. The cause of this error differed however.

A previously discussed error in the serial terminal display has been successfully resolved.

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.

In the case of the EKG measurements, the accuracy of the reported value may vary by as much as $\pm 5\%$ BPM. The actual inaccuracy is a function of the actual EKG frequency. We have attributed this inaccuracy to the implementation of the EKG measurement functions. Specifically the timing mechanism used sample the analog sensor relies on a software defined delay function to capture the signal. This choice was made because the onboard hardware did not have a sufficient number of hardware based timers to use. As a result, the EKG measurements are forced to rely on a system that is not as accurate initially. Previous experience has indicated that a software timer can made highly accurate given enough calibration and adjustment of correcting variables. Time constraints limited our calibration to a lower level of accuracy for the time being. In addition, a more precise determination of the actual sampling rate will yield correspondingly more accurate measurements of the EKG frequency. We have made a decision to consider the level of precision sufficiently accurate for this prototype version. A subsequent version will build upon these lessons learned here.

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:

Phase I Tests:

- 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 systolic 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 associated changes in system state
- Hardware timer updates the system's minor cycle counter

The specifications added during Phase III require that the following Phase III properties be tested to satisfaction.

Phase III Tests:

- EKG related functions are able to accurately and reliably measure signals within the specified ranges
- The System operates using an non-preemptive, priority-based real time operating system
- Users may use the keypad to retrieve and view specific measurements
- The system startup task properly initializes the hardware needed for operation

The following tests are required to ensure the sepcifications and requirements for Phase IV are met.

Phase IV Tests:

- Command task properly handles expected commands
- Command Task properly handles unexpected and incorrect commands
- Web server properly responds to browser requests and forwards information to the system
- Remote connection can initialize and maintains a connection for a significant period of time
- User interaction from keypad and web browser can occur together without significant loss of functionality

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

5.1 Test Specification

5.1.1 Scheduler

The scheduler (FreeRTOS) 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 6.

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

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

Table 4: Initial values and warning/alarm states

5.1.7 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 as shown in Listing 2.

```
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 %
```

Listing 2: Remote Terminal Output

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.

5.1.9 EKG Capture

EKG Capture should be tested to grab correct values from an input source (sine wave).

5.1.10 EKG Process

EKG Process should successfully measure sine waves between 37 and 3750 Hz. Key frequencies should be chosen between this range, like 37, 40, 60, 100, 200, 500, 1000, 3000, and 3600, and 3750. The system should successfully detect each of these frequencies.

5.1.11 Command Task

The command task should successfully parse each of the commands shown in Appendix B. Upon receiving a command, it should also generate a correct response. For example, a “M T” command should give the response shown in Listing 3. The “A” refers to an ACK. If the command was invalid, the response should simply be an “E”.

```
A
<p>Temperature 0 C</p>
```

Listing 3: Command Response

5.1.12 Remote Communications

The server should be tested to successfully respond to HTTP requests. This means opening a file and sending it over HTTP to the requester. Since the system also has special URLs for commands, it should also recognize these commands.

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 6 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 cues, 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	20	30	12+
Jarrett	9	30	16	9
Jonathan	15	30	15	10

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 SUPPORTED WEB BROWSER COMMANDS

The Commands and Responses for the embedded application task are given as follows.

S The S command indicates START mode. The command shall start the embedded tasks by directing the hardware to initiate all the measurement tasks. In doing so, the command shall enable all the interrupts.

P The P command indicates STOP mode. This command shall stop the embedded tasks by terminating any running measurement tasks. Such an action shall disable any data collecting interrupts.

D The D command enables or disables the OLED display.

M¡payload¿ The M command. The M command requests the return of the most recent value(s) of the specified data. The M response. The M response returns the most recent value(s) of the specified data.

The ¡payload¿ is defined as one of the following:

- **A** - Instruct the system to measure all sensors. Will return the most recent measurement of each sensor
- **T** - Instruct system to measure Temperature. Will return the temperature in degrees Celsius
- **B** - Instruct system to measure the patient blood pressure. Will return the systolic and diastolic blood pressure in mmHg
- **P** - Instruct system to measure patient pulse rate. Will return the heart rate in beats per minute
- **E** - Instruct the system to perform an EKG measurement. Will return the primary EKG frequency
- **D** - DEBUG mode only. Returns the local display status
- **M** - DEBUG mode only. Returns system measurement status. Returns boolean true if measurement is enabled, false otherwise

A The A response acknowledges the receipt of the identified command.

E The E error response is given for incorrect commands or non-existent commands.

C ACTIVITY DIAGRAMS

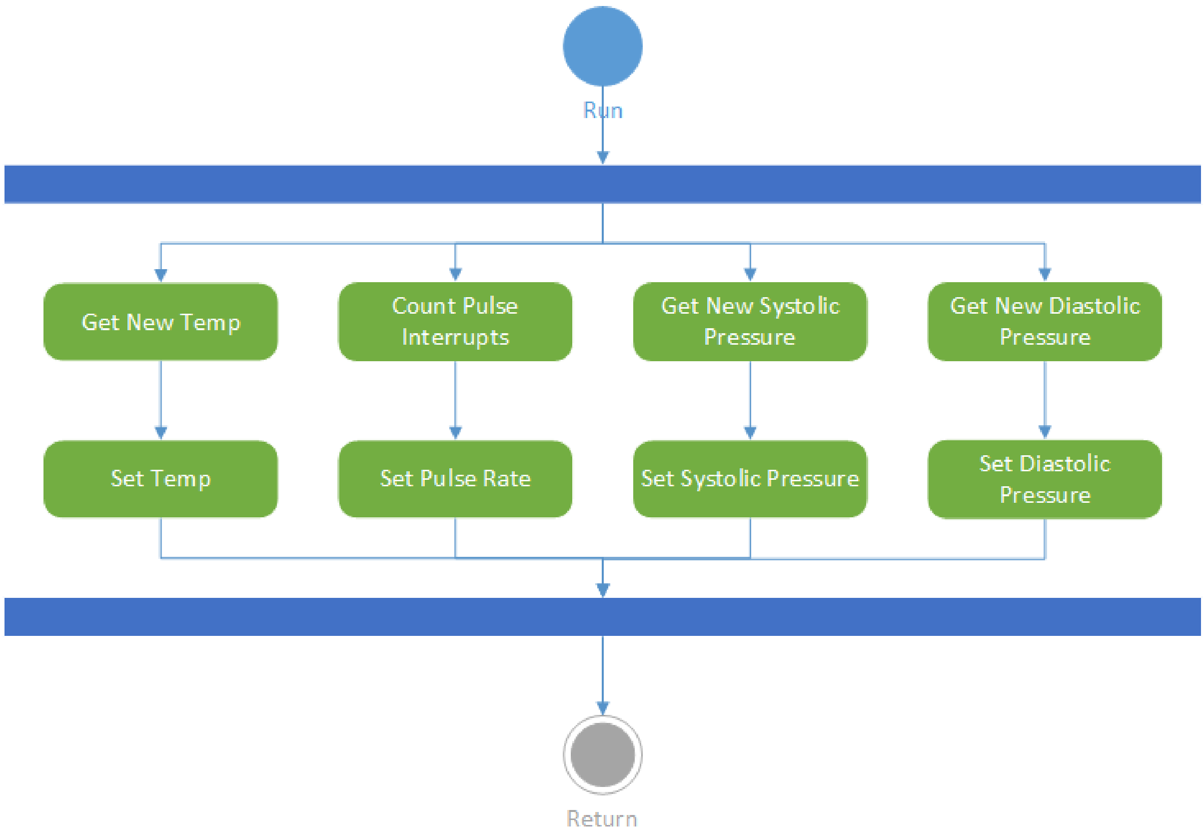


Figure 9: Measure Activity Diagram

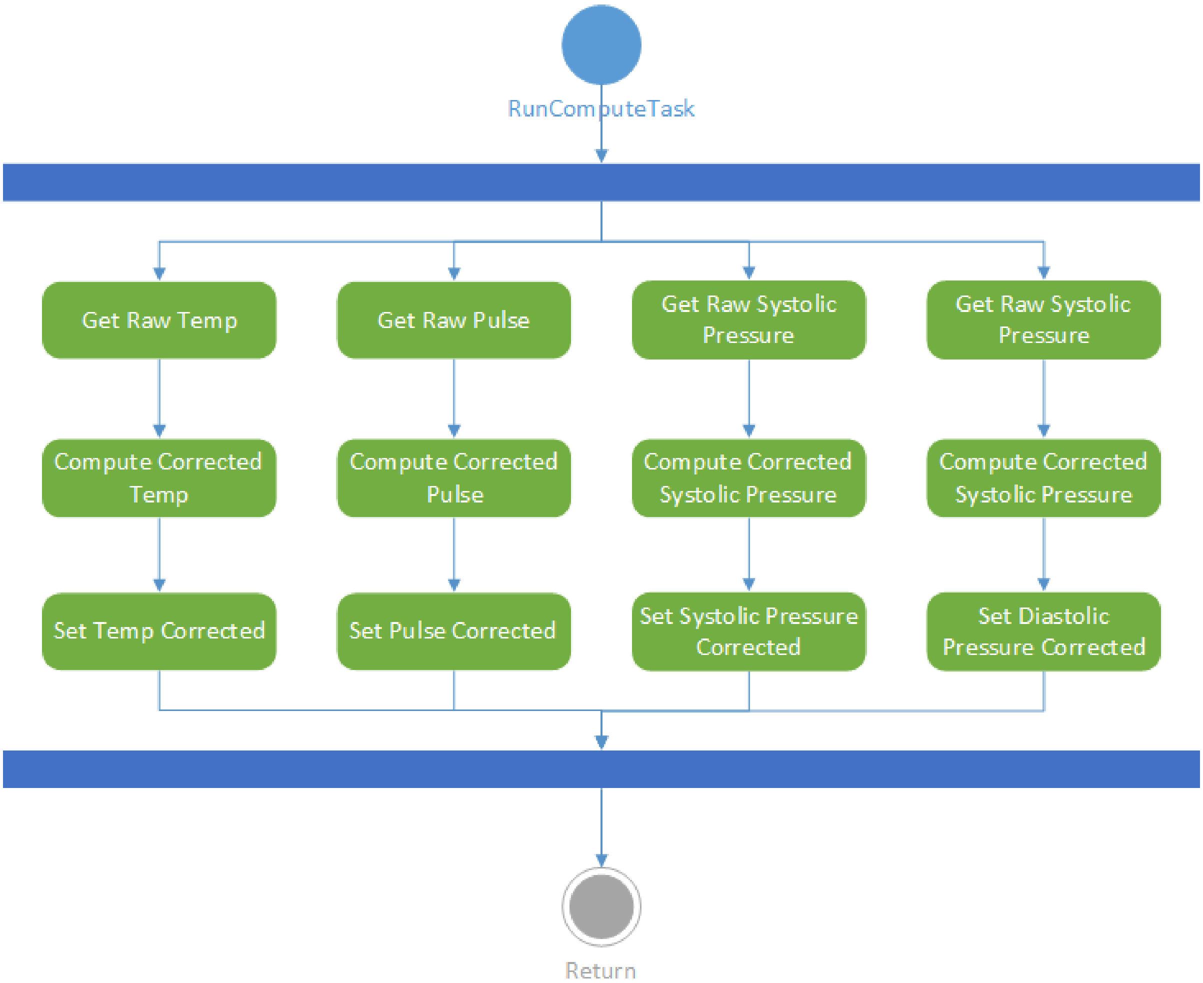


Figure 10: Compute Activity Diagram

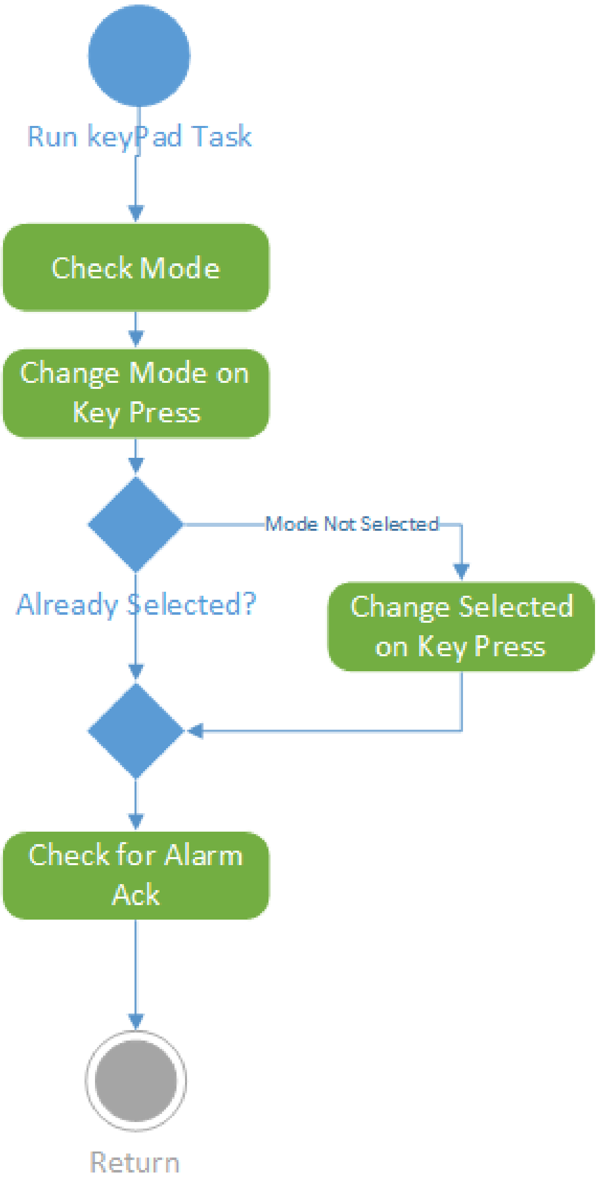


Figure 11: Keypad Activity Diagram

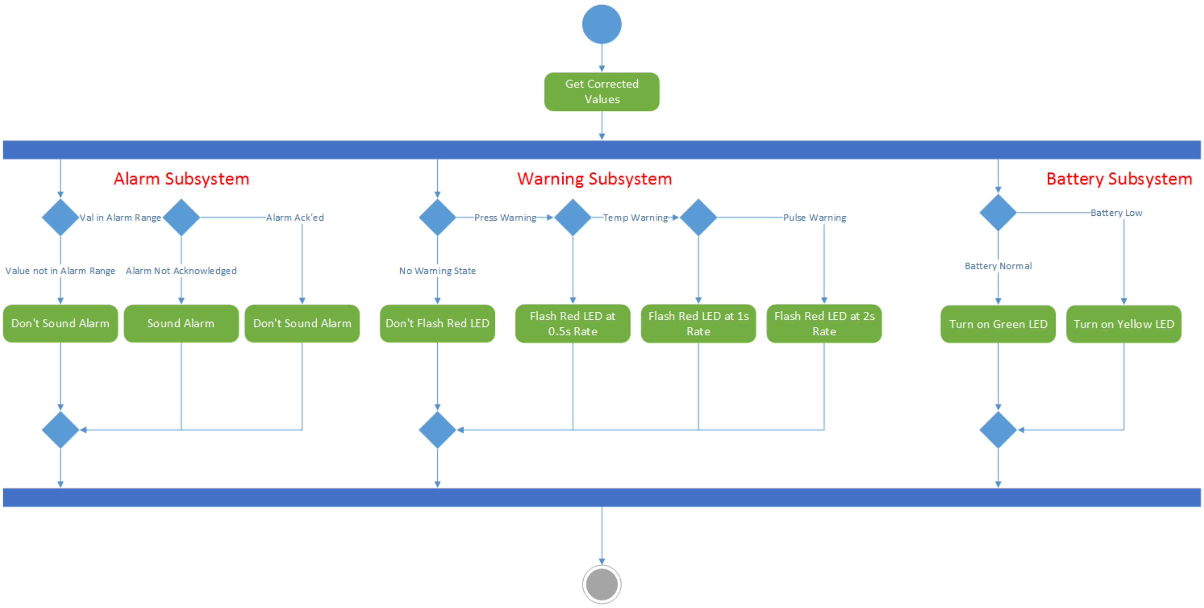


Figure 12: Warning Activity Diagram

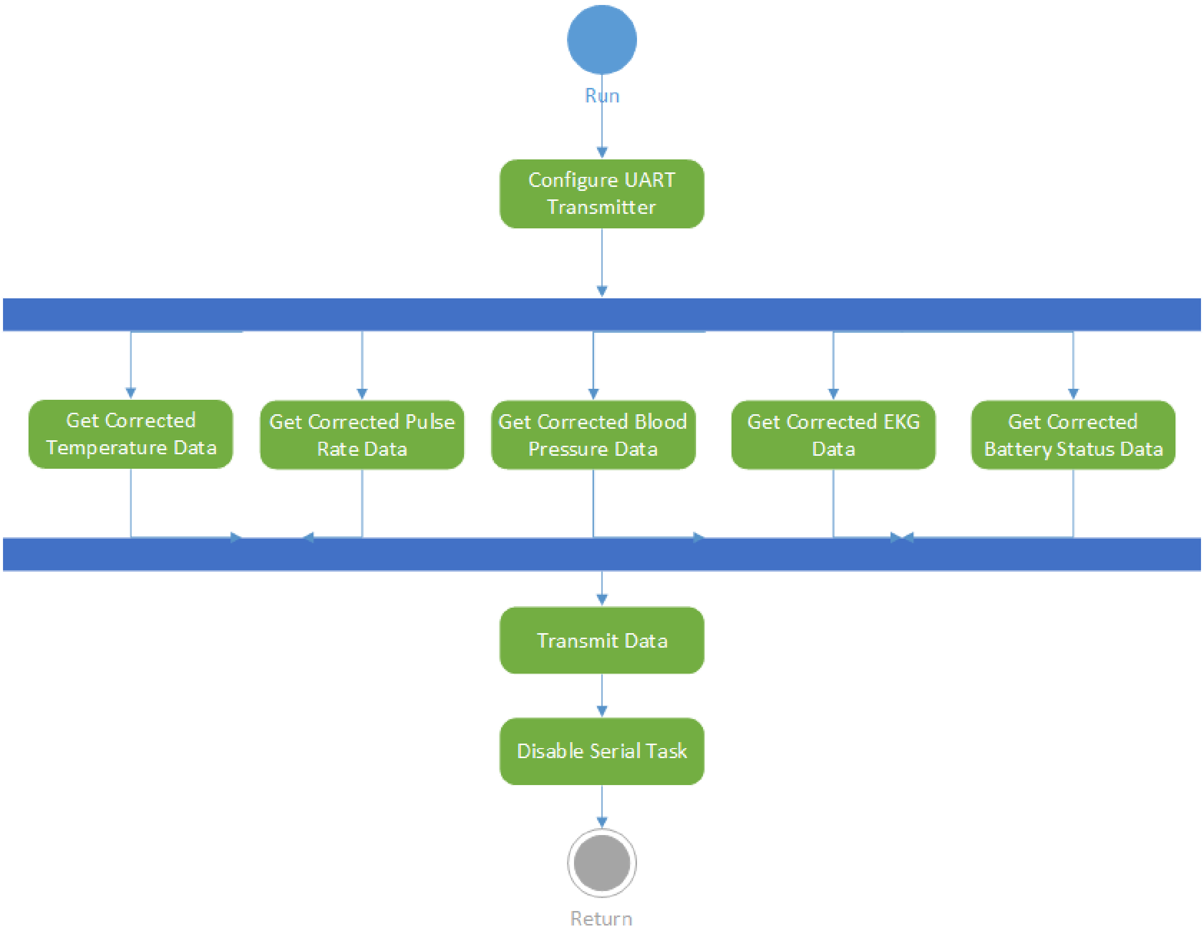


Figure 13: Serial Activity Diagram

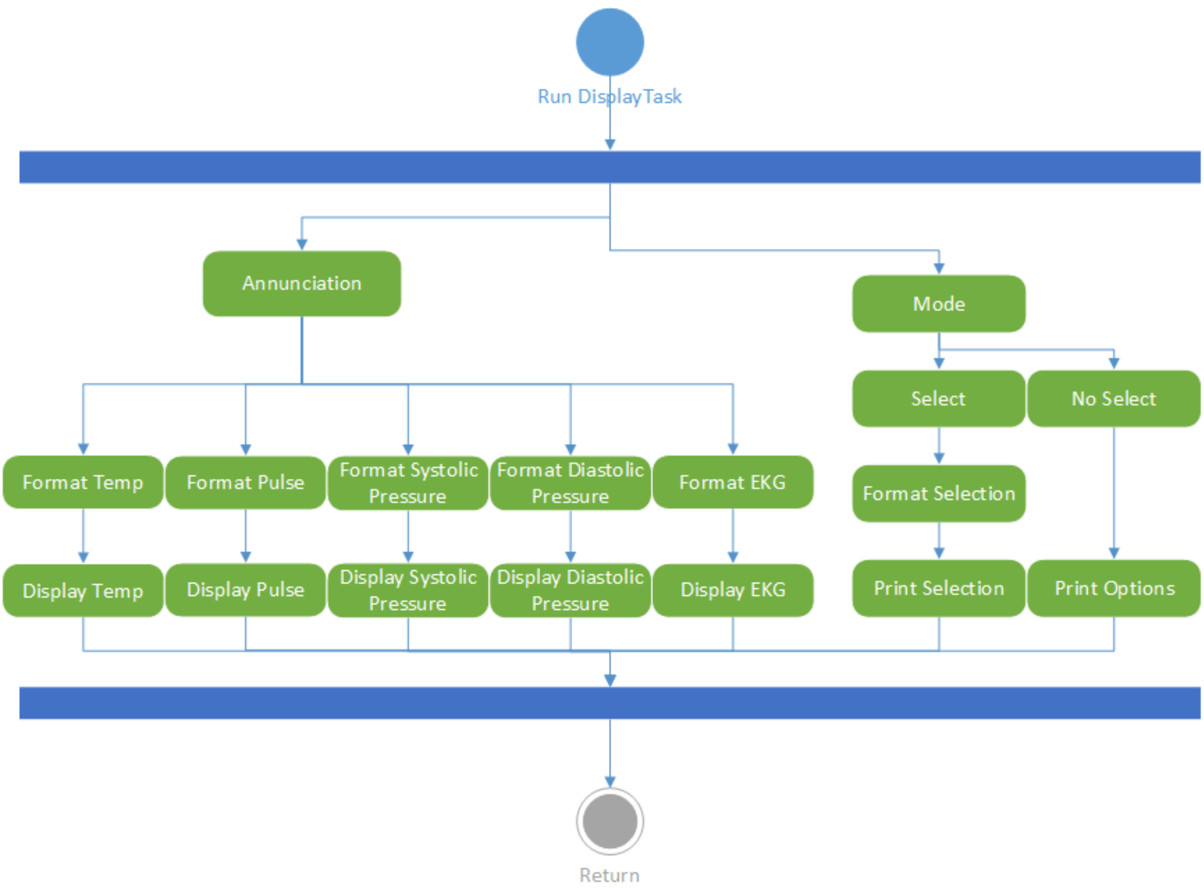


Figure 14: Display Activity Diagram

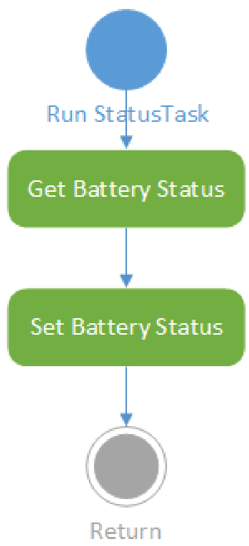


Figure 15: Status Activity Diagram

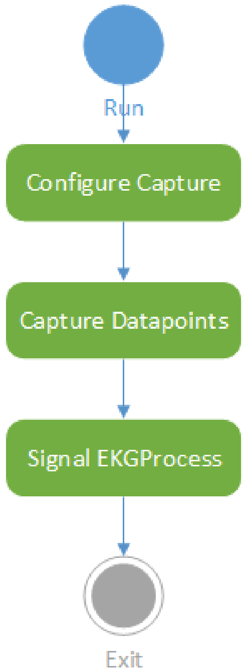


Figure 16: EKG Capture Activity Diagram

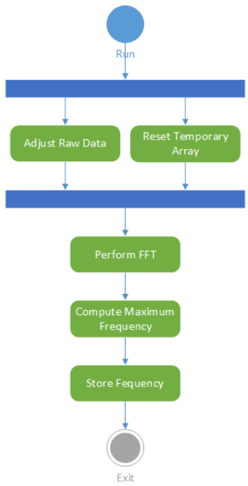


Figure 17: EKG Process Activity Diagram

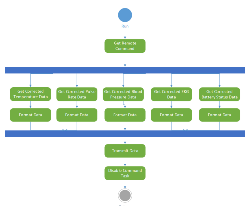


Figure 18: Command Activity Diagram

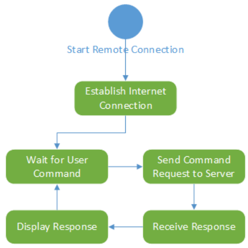


Figure 19: Remote Activity Diagram

D SOURCE CODE

Source code for this project is provided below.

D.1 Main Function

../code/main.c

```c
/*
    FreeRTOS V7.0.1 - Copyright (C) 2011 Real Time Engineers Ltd.


    ***************************************************************************
     *                                                                       *
     *    FreeRTOS tutorial books are available in pdf and paperback.        *
     *    Complete, revised, and edited pdf reference manuals are also       *
     *    available.                                                         *
     *                                                                       *
     *    Purchasing FreeRTOS documentation will not only help you, by       *
     *    ensuring you get running as quickly as possible and with an        *
     *    in-depth knowledge of how to use FreeRTOS, it will also help       *
     *    the FreeRTOS project to continue with its mission of providing     *
     *    professional grade, cross platform, de facto standard solutions    *
     *    for microcontrollers - completely free of charge!                  *
     *                                                                       *
     *    >>> See http://www.FreeRTOS.org/Documentation for details. <<<     *
     *                                                                       *
     *    Thank you for using FreeRTOS, and thank you for your support!      *
     *                                                                       *
    ***************************************************************************


    This file is part of the FreeRTOS distribution and has been modified to
```

```
    demonstrate three simple tasks running.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    >>>NOTE<<< The modification to the GPL is included to allow you to
    distribute a combined work that includes FreeRTOS without being obliged to
    provide the source code for proprietary components outside of the FreeRTOS
    kernel.  FreeRTOS is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
    or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    http://www.FreeRTOS.org - Documentation, latest information, license and
    contact details.

    http://www.SafeRTOS.com - A version that is certified for use in safety
    critical systems.

    http://www.OpenRTOS.com - Commercial support, development, porting,
    licensing and training services.
*/


/*
 * Creates all the application tasks, then starts the scheduler.  The WEB
 * documentation provides more details of the standard demo application tasks.
 * In addition to the standard demo tasks, the following tasks and tests are
 * defined and/or created within this file:
 *
 * "OLED" task - the OLED task is a 'gatekeeper' task.  It is the only task that
 * is permitted to access the display directly.  Other tasks wishing to write a
 * message to the OLED send the message on a queue to the OLED task instead of
```

```
 * accessing the OLED themselves.  The OLED task just blocks on the queue waiting
 * for messages - waking and displaying the messages as they arrive.
 *
 * "Check" hook -  This only executes every five seconds from the tick hook.
 * Its main function is to check that all the standard demo tasks are still
 * operational.  Should any unexpected behaviour within a demo task be discovered
 * the tick hook will write an error to the OLED (via the OLED task).  If all the
 * demo tasks are executing with their expected behaviour then the check task
 * writes PASS to the OLED (again via the OLED task), as described above.
 *
 * "uIP" task -  This is the task that handles the uIP stack.  All TCP/IP
 * processing is performed in this task.
 */




/*************************************************************************
 * Please ensure to read http://www.freertos.org/portlm3sx965.html
 * which provides information on configuring and running this demo for the
 * various Luminary Micro EKs.
 *************************************************************************/

/* Set the following option to 1 to include the WEB server in the build.  By
default the WEB server is excluded to keep the compiled code size under the 32K
limit imposed by the KickStart version of the IAR compiler.  The graphics
libraries take up a lot of ROM space, hence including the graphics libraries
and the TCP/IP stack together cannot be accommodated with the 32K size limit. */

//  set this value to non 0 to include the web server

#define mainINCLUDE_WEB_SERVER      0


/* Standard includes. */
#include <stdio.h>

/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

/* Hardware library includes. */
#include "hw_memmap.h"
#include "hw_types.h"
#include "hw_sysctl.h"
```

```
#include "sysctl.h"
#include "gpio.h"
// #include "grlib.h"
#include "rit128x96x4.h"

/* Demo app includes. */
// #include "lcd_message.h"
// #include "bitmap.h"


#include "timebase.h"
#include "schedule.h"

/* Tasks */
#include "tcb.h"
#include "startup.h"
#include "measure.h"
#include "compute.h"
#include "display.h"
#include "warning.h"
#include "keypad.h"
#include "serial.h"
#include "status.h"
#include "keyPad.h"
#include "ekgCapture.h"
#include "ekgProcess.h"
#include "commandTask.h"

/*-----------------------------------------------------------*/

/*
  The time between cycles of the 'check' functionality (defined within the
  tick hook.
*/
#define mainCHECK_DELAY ( ( portTickType ) 5000 / portTICK_RATE_MS )

// Size of the stack allocated to the uIP task.
#define mainBASIC_WEB_STACK_SIZE            ( configMINIMAL_STACK_SIZE * 3 )

// The OLED task uses the sprintf function so requires a little more stack
   too.
#define mainOLED_TASK_STACK_SIZE        ( configMINIMAL_STACK_SIZE + 50 )

//  Task priorities.
#define mainQUEUE_POLL_PRIORITY         ( tskIDLE_PRIORITY + 2 )
#define mainCHECK_TASK_PRIORITY         ( tskIDLE_PRIORITY + 3 )
#define mainSEM_TEST_PRIORITY         ( tskIDLE_PRIORITY + 1 )
#define mainBLOCK_Q_PRIORITY          ( tskIDLE_PRIORITY + 2 )
#define mainCREATOR_TASK_PRIORITY             ( tskIDLE_PRIORITY + 3 )
#define mainINTEGER_TASK_PRIORITY             ( tskIDLE_PRIORITY )
#define mainGEN_QUEUE_TASK_PRIORITY       ( tskIDLE_PRIORITY )
```

```
//  The maximum number of messages that can be waiting for display at any
    one time.
  #define mainOLED_QUEUE_SIZE          ( 3 )

// Dimensions the buffer into which the jitter time is written.
  #define mainMAX_MSG_LEN            25

/*
  The period of the system clock in nano seconds.  This is used to calculate
  the jitter time in nano seconds.
*/

#define mainNS_PER_CLOCK ( ( unsigned portLONG ) ( ( 1.0 / ( double )
    configCPU_CLOCK_HZ ) * 1000000000.0 ) )


// Constants used when writing strings to the display.

#define mainCHARACTER_HEIGHT          ( 9 )
#define mainMAX_ROWS_128          ( mainCHARACTER_HEIGHT * 14 )
#define mainMAX_ROWS_96           ( mainCHARACTER_HEIGHT * 10 )
#define mainMAX_ROWS_64           ( mainCHARACTER_HEIGHT * 7 )
#define mainFULL_SCALE            ( 15 )
#define ulSSI_FREQUENCY           ( 3500000UL )

/*-----------------------------------------------------------*/

/*
 * Configure the hardware .
 */
static void prvSetupHardware( void );

/*
 * Hook functions that can get called by the kernel.
 */
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *
    pcTaskName );
void vApplicationTickHook( void );

/* The tasks */
void measure(void *vParameters);
void compute(void *vParameters);
void display(void *vParameters);
void keyPad(void *vParameters);
void warning(void *vParameters);
void serial(void *vParameters);
void status(void *vParameters);
void ekgCapture(void *vParameters);
void ekgProcess(void *vParameters);

void command(void *vParameters);

/*-----------------------------------------------------------*/
```

```
/*
  The queue used to send messages to the OLED task.
*/
xQueueHandle xOLEDQueue;

/*-----------------------------------------------------------*/

xTaskHandle computeHandle;
xTaskHandle serialHandle;
xTaskHandle measureHandle;
xTaskHandle displayHandle;
xTaskHandle ekgCaptureHandle;
xTaskHandle ekgProcessHandle;
xTaskHandle commandHandle;

/*************************************************************************
 * Please ensure to read http://www.freertos.org/portlm3sx965.html
 * which provides information on configuring and running this demo for the
 * various Luminary Micro EKs.
 *************************************************************************/

int main( void )
{
    prvSetupHardware();

    /* Startup task */
    startup();

    /* Start the tasks */
    xTaskCreate(measure, "measure task", 100,NULL, 2, &measureHandle);
    xTaskCreate(compute, "compute task", 100,NULL, 3, &computeHandle);
    xTaskCreate(ekgCapture, "ekgCapture task", 1024,NULL, 1, &
   ekgCaptureHandle);
    xTaskCreate(ekgProcess, "ekgProcess task", 1024,NULL, 3, &
   ekgProcessHandle);
    xTaskCreate(display, "display task", 100,NULL, 5, &displayHandle);
    xTaskCreate(keyPad, "keyPad task", 100,NULL, 4,NULL);
    xTaskCreate(warning, "warning task", 100,NULL, 5,NULL);
    xTaskCreate(serial, "serial task", 100,NULL, 5,&serialHandle);
    xTaskCreate(status, "status task", 100,NULL, 1,NULL);
    xTaskCreate(command, "command task", 100,NULL, 3, &commandHandle);

    vTaskSuspend(measureHandle);
    vTaskSuspend(computeHandle);
    vTaskSuspend(serialHandle);
    vTaskSuspend(commandHandle);
    vTaskSuspend(ekgProcessHandle);
    /* Start the scheduler. */
    vTaskStartScheduler();

    /* Will only get here if there was insufficient memory to create the
   idle task.
```

```
        (Created by vTaskStartScheduler())*/
    return 0;
}

/*
  three dummy tasks
*/

void measure(void *vParameters)
{
  while(1)
  {
      measureTask.runTaskFunction(measureTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE * MAJOR_CYCLE);
  }
}

void compute(void *vParameters)
{
  while(1)
  {
      computeTask.runTaskFunction(computeTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void keyPad(void *vParameters)
{
  while(1)
  {
      keyPadTask.runTaskFunction(keyPadTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void display(void *vParameters)
{
  while(1)
  {
      displayTask.runTaskFunction(displayTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void warning(void *vParameters)
{
  while(1)
  {
      warningTask.runTaskFunction(warningTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

```

```c
void serial(void *vParameters)
{
  while(1)
  {
      serialTask.runTaskFunction(serialTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void status(void *vParameters)
{
  while(1)
  {
      statusTask.runTaskFunction(statusTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void ekgCapture(void *vParameters)
{
  while(1)
  {
      ekgCaptureTask.runTaskFunction(ekgCaptureTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void ekgProcess(void *vParameters)
{
  while(1)
  {
      ekgProcessTask.runTaskFunction(ekgProcessTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

void command(void *vParameters)
{
  while(1)
  {
      commandTask.runTaskFunction(commandTask.taskDataPtr);
      vTaskDelay(MINOR_CYCLE);
  }
}

/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *
    pcTaskName )
{
    ( void ) pxTask;
    ( void ) pcTaskName;

```

```
    while( 1 );
}

/*-----------------------------------------------------------*/

void prvSetupHardware( void )
{
    /*
      If running on Rev A2 silicon, turn the LDO voltage up to 2.75V.  This is
      a workaround to allow the PLL to operate reliably.
    */

    if( DEVICE_IS_REVA2 )
    {
        SysCtlLDOSet( SYSCTL_LDO_2_75V );
    }

    // Set the clocking to run from the PLL at 50 MHz

    SysCtlClockSet( SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_8MHZ );

    /*
      Enable Port F for Ethernet LEDs
            LED0        Bit 3   Output
            LED1        Bit 2   Output
    */

    SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOF );
    GPIODirModeSet( GPIO_PORTF_BASE, (GPIO_PIN_2 | GPIO_PIN_3), GPIO_DIR_MODE_HW );
    GPIOPadConfigSet( GPIO_PORTF_BASE, (GPIO_PIN_2 | GPIO_PIN_3 ), GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD );

}


/*-----------------------------------------------------------*/

void vApplicationTickHook( void )
{
    // static xOLEDMessage xMessage = { "PASS" };
    static unsigned portLONG ulTicksSinceLastDisplay = 0;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /*
      Called from every tick interrupt.  Have enough ticks passed to make it
      time to perform our health status check again?
    */

    ulTicksSinceLastDisplay++;
    if( ulTicksSinceLastDisplay >= mainCHECK_DELAY )
```

```
    {
        ulTicksSinceLastDisplay = 0;

    }
}
```

D.2 Global Data

../code/globals.h

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

#include <String.h>
#include "inc/hw_types.h"
#include "CircularBuffer.h"
#include "stddef.h"
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include <string.h>

#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 EKG_FREQ_RLT 0

#define BATT_INIT 200

#define NUM_EKG_SAMPLES 256
#define SAMPLE_FREQ  8000 // # sample frequency to get a good measure of <
    3750 Hz
#define COMMAND_LENGTH 10 // length of command string
#define RESPONSE_LENGTH 600 // length of response string


typedef struct global_data {
  CircularBuffer temperatureRaw;
  CircularBuffer systolicPressRaw;
  CircularBuffer diastolicPressRaw;
```

```
    CircularBuffer pulseRateRaw;
    int ekgRaw[NUM_EKG_SAMPLES];
    int ekgTemp[NUM_EKG_SAMPLES];

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

    unsigned short batteryState;
    unsigned short mode;
    unsigned short measurementSelection;
    tBoolean measurementComplete;
    tBoolean ekgCaptureDone;
    tBoolean ekgProcessDone;
    tBoolean responseReady;
    tBoolean alarmAcknowledge;
    tBoolean select;
    unsigned short scroll;

    char commandStr[COMMAND_LENGTH];
    char responseStr[RESPONSE_LENGTH];
} 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

```
/*
 * globals.c
 * Author(s): Jonathan Ellington, Patrick Ma
 * 1/28/2014
 *
 * Defines global data for tasks to access
 */
#include "globals.h"
#include "utils/ustdlib.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];
static int ekgFrequencyResultArr[16];

static signed int ekgRaw[NUM_EKG_SAMPLES];  // initialize all the elements to 0
static signed int ekgTemp[NUM_EKG_SAMPLES];

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);

  memset(global.ekgRaw, 0, NUM_EKG_SAMPLES);
  memset(global.ekgTemp, 0, NUM_EKG_SAMPLES);

  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);
  global.ekgFrequencyResult = cbWrap(ekgFrequencyResultArr, sizeof(int), 16);

  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;
  int fr = EKG_FREQ_RLT;

  // Add initial values
  cbAdd(&(global.temperatureRaw), &tr);
  cbAdd(&(global.systolicPressRaw), &sr);
  cbAdd(&(global.diastolicPressRaw), &dr);
  cbAdd(&(global.pulseRateRaw), &pr);

  cbAdd(&(global.temperatureCorrected), &tc);
  cbAdd(&(global.systolicPressCorrected), &sc);
  cbAdd(&(global.diastolicPressCorrected), &dc);
  cbAdd(&(global.pulseRateCorrected), &pc);
```

```
  cbAdd(&(global.ekgFrequencyResult), &fr);

  // Set normal variables
  global.batteryState = 200;
  global.mode = 1;
  global.measurementSelection = 0;
  global.alarmAcknowledge = false;
  global.select = false;
  global.scroll = 0;

  memset(&(global.commandStr), NULL, sizeof(char) * COMMAND_LENGTH);
  memset(&(global.responseStr), NULL, sizeof(char) * RESPONSE_LENGTH);

  global.measurementComplete = false;
  global.ekgCaptureDone = false;
  global.ekgProcessDone = false;
  global.responseReady = false;
}

// debug tool
void debugPin47() {
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);          // debug
  GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
  long a = GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_2);
  if(0 == a)
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
  else
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
}
```

D.3 Timebase

../code/timebase.h

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

#include "hw_types.h"

#define MINOR_CYCLE 250       // minor cycle, in milliseconds
#define MAJOR_CYCLE 4       // major cycle, in number of minor cycles
#define PULSE_CYCLE 8        // pulse rate measurement time, in number of
                            // minor cycles
#define IS_PULSE_CYCLE (minor_cycle_ctr % (MAJOR_CYCLE * PULSE_CYCLE) == 0)
#define IS_MAJOR_CYCLE (minor_cycle_ctr % MAJOR_CYCLE == 0)

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

```c

// returns whether or not length minor cycles have happened since
// start_time
tBoolean timeHasPassed(int start_time, int length);

void initializeTimebase(void);
void TimerAIntHandler(void);
```

D.4 Modified lmi_fs.c

../code/lmi_fs.c

```c
// *****************************************************************************
//
// lmi_fs.c - File System Processing for enet_io application.
//
// Copyright (c) 2007-2012 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 9107 of the EK-LM3S8962 Firmware Package.
//
// *****************************************************************************

#include <string.h>
#include "inc/hw_types.h"
#include "utils/lwiplib.h"
#include "utils/ustdlib.h"
#include "httpserver_raw/fs.h"
#include "httpserver_raw/fsdata.h"

#include "FreeRTOS.h"
#include "task.h"
#include "globals.h"

```

```c
extern xTaskHandle commandHandle;
//#include "io.h"

//*****************************************************************************
//
// Include the file system data for this application.  This file is generated
// by the makefsfile utility, using the following command (all on one line):
//
//     makefsfile -i fs -o io_fsdata.h -r -h
//
// If any changes are made to the static content of the web pages served by the
// application, this script must be used to regenerate io_fsdata.h in order
// for those changes to be picked up by the web server.
//
//*****************************************************************************
#include "io_fsdata.h"

//*****************************************************************************
//
// Global Settings for demo page content.
//
//*****************************************************************************
static char g_cSampleTextBuffer[10] = {0};

//*****************************************************************************
//
// Open a file and return a handle to the file, if found.  Otherwise,
// return NULL.  This function also looks for special filenames used to
// provide specific status information or to control various subsystems.
// These filenames are used by the JavaScript on the "IO Control Demo 1"
// example web page.
//
//*****************************************************************************
struct fs_file *
fs_open(char *name)
{
    char *data;
    int i;
    const struct fsdata_file *ptTree;
```

```c
    struct fs_file *ptFile = NULL;

    //
    // Allocate memory for the file system structure.
    //
    ptFile = mem_malloc(sizeof(struct fs_file));
    if(NULL == ptFile)
    {
        return(NULL);
    }

    //
    // Process command request
    //
    if(strncmp(name, "/cgi-bin/send_command/value=", 28) == 0)
    {
        // Get Command String
        data = name;
        data += 28;
        i = 0;
        do
        {
            switch(data[i])
            {
                case '+':
                    global.commandStr[i] = ' ';
                    break;
                default:
                    global.commandStr[i] = data[i];
                    break;
            }
            if(global.commandStr[i] == 0)
            {
                break;
            }
            i++;
        }while(i < sizeof(global.commandStr));

        // Setup the file structure to return whatever.
        ptFile->data = NULL;
        ptFile->len = 0;
        ptFile->index = 0;
        ptFile->pextension = NULL;

        vTaskResume(commandHandle);
        return(ptFile);
    }

    //
    // Process command response
    //
    if(strncmp(name, "/cgi-bin/receive_command", 24) == 0)
    {
```

```c
    if (global.responseReady) {

        char *buf = (char *) mem_malloc(sizeof(global.responseStr));
        memcpy(buf, global.responseStr, sizeof(global.responseStr));

        // Setup the file structure to return whatever.
        ptFile->data = buf;
        ptFile->len = strlen(buf);
        ptFile->index = 0;
        ptFile->pextension = NULL;

        //
        // Return the file system pointer.
        //
        return(ptFile);
      }

        ptFile->data = NULL;
        ptFile->len = 0;
        ptFile->index = 0;
        ptFile->pextension = NULL;

      return ptFile;
    }

    //
    // If I can't find it there, look in the rest of the main file system
    //
    else
    {
        //
        // Initialize the file system tree pointer to the root of the linked list.
        //
        ptTree = FS_ROOT;

        //
        // Begin processing the linked list, looking for the requested file name.
        //
        while(NULL != ptTree)
        {
            //
            // Compare the requested file "name" to the file name in the
            // current node.
            //
            if(strncmp(name, (char *)ptTree->name, ptTree->len) == 0)
            {
                //
                // Fill in the data pointer and length values from the
                // linked list node.
                //
                ptFile->data = (char *)ptTree->data;
```

```
                ptFile->len = ptTree->len;

                //
                // For now, we setup the read index to the end of the file,
                // indicating that all data has been read.
                //
                ptFile->index = ptTree->len;

                //
                // We are not using any file system extensions in this
                // application, so set the pointer to NULL.
                //
                ptFile->pextension = NULL;

                //
                // Exit the loop and return the file system pointer.
                //
                break;
            }

            //
            // If we get here, we did not find the file at this node of the linked
            // list.  Get the next element in the list.
            //
            ptTree = ptTree->next;
        }
    }

    //
    // If we didn't find the file, ptTee will be NULL.  Make sure we
    // return a NULL pointer if this happens.
    //
    if(NULL == ptTree)
    {
        mem_free(ptFile);
        ptFile = NULL;
    }

    //
    // Return the file system pointer.
    //
    return(ptFile);
}

//*****************************************************************************
//
// Close an opened file designated by the handle.
//
//*****************************************************************************
```

```c
void
fs_close(struct fs_file *file)
{
    //
    // Free the main file system object.
    //
    mem_free(file);
}

//
    ******************************************************************************
//
// Read the next chunck of data from the file.  Return the count of data
// that was read.  Return 0 if no data is currently available.  Return
// a -1 if at the end of file.
//
//
    ******************************************************************************
int
fs_read(struct fs_file *file, char *buffer, int count)
{
    int iAvailable;

    //
    // Check to see if a command (pextension = 1).
    //
    if(file->pextension == (void *)1)
    {
        //
        // Nothting to do for this file type.
        //
        file->pextension = NULL;
        return(-1);
    }

    //
    // Check to see if more data is available.
    //
    if(file->len == file->index)
    {
        //
        // There is no remaining data.  Return a -1 for EOF indication.
        //
        return(-1);
    }

    //
    // Determine how much data we can copy.  The minimum of the 'count'
    // parameter or the available data in the file system buffer.
    //
```

```
    iAvailable = file->len - file->index;
    if(iAvailable > count)
    {
        iAvailable = count;
    }

    //
    // Copy the data.
    //
    memcpy(buffer, file->data + iAvailable, iAvailable);
    file->index += iAvailable;

    //
    // Return the count of data that we copied.
    //
    return(iAvailable);
}
```

D.5 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);

#endif // _CIRCULAR_BUFFER_H
```

../code/CircularBuffer.c

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

#include "CircularBuffer.h"
#include <stdio.h>

CircularBuffer cbWrap(void *arr, int se, int ne) {
  CircularBuffer cb;
  cb.array = arr;
  cb.sizeElm = se;
  cb.nElm = ne;
  cb.currElm = 0;

  return cb;
}

/* Returns the current elment in the circular buffer */
void *cbGet(CircularBuffer *cb) {
  int sizeElm = cb->sizeElm;
  int index = cb->currElm;

  unsigned char *bytePtr = (unsigned char *)cb->array;
  bytePtr += sizeElm * index;

  return (void *) bytePtr;
}

```

```c
/* Returns a pointer to the wrapped array, with the oldest elment
 * at the end and the newest elment at the beginning */
void *cbGetArray(CircularBuffer *cb) {
  return NULL;
}

/* Adds elm to cb */
void cbAdd(CircularBuffer *cb, void *elm) {
  cb->currElm = (cb->currElm + 1) % cb->nElm;

  int sizeElm = cb->sizeElm;
  int index = cb->currElm;

  // copy sizeElm bytes from elm into the right spot
  unsigned char *elmPtr = (unsigned char *) elm;
  unsigned char *arrElmPtr = (unsigned char *) cb->array;
  arrElmPtr += sizeElm * index;

  for (int i = 0; i < sizeElm; i++) {
    arrElmPtr[i] = elmPtr[i];
  }
}
```

D.6 Warm up File (For Interrupt Initialization)

../code/startup_ewarm.c

```c
// *****************************************************************************
//
// startup_ewarm.c - Boot code for Stellaris.
//
// Copyright (c) 2006-2007 Luminary Micro, Inc.  All rights reserved.
//
// Software License Agreement
//
// Luminary Micro, Inc. (LMI) is supplying this software for use solely and
// exclusively on LMI's microcontroller products.
//
// The software is owned by LMI and/or its suppliers, and is protected under
// applicable copyright laws.  All rights are reserved.  Any use in violation
// of the foregoing restrictions may subject the user to criminal sanctions
// under applicable laws, as well as to civil liability for the breach of the
// terms and conditions of this license.
//
// THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
// OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
```

```
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
// LMI SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR
// CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 100 of the Stellaris Ethernet
// Applications Library.
//
//*****************************************************************************

//*****************************************************************************
//
// Enable the IAR extensions for this source file.
//
//*****************************************************************************
#pragma language=extended

//*****************************************************************************
//
// Forward declaration of the default fault handlers.
//
//*****************************************************************************
static void NmiSR(void);
static void FaultISR(void);
static void IntDefaultHandler(void);
extern void lwIPEthernetIntHandler(void);
extern void timer1IntHandler (void);

//*****************************************************************************
//
// External declaration for the interrupt handler used by the application.
//
//*****************************************************************************


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

```c
//
// The entry point for the application.
//
//*****************************************************************************
extern void __iar_program_start(void);
extern void xPortPendSVHandler(void);
extern void xPortSysTickHandler(void);
extern void vPortSVCHandler(void);
extern void vT2InterruptHandler( void );
extern void vT3InterruptHandler( void );
extern void vEMAC_ISR( void );
//extern void Timer0IntHandler( void ); not sure where this is from
extern void TimerAIntHandler( void ); // timebase.h
extern void ADC0IntHandler(void); // ADC0 -- for EKG sensor

//*****************************************************************************
//
// Reserve space for the system stack.
//
//*****************************************************************************
#ifndef STACK_SIZE
#define STACK_SIZE                              1024
#endif
static unsigned long pulStack[STACK_SIZE] @ ".noinit";

//*****************************************************************************
//
// A union that describes the entries of the vector table. The union is needed
// since the first entry is the stack pointer and the remainder are function
// pointers.
//
//*****************************************************************************
typedef union
{
    void (*pfnHandler)(void);
    unsigned long ulPtr;
}
uVectorEntry;

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

```c
//
// The minimal vector table for a Cortex-M3.  Note that the proper constructs
// must be placed on this to ensure that it ends up at physical address
// 0x0000.0000.
//
// *****************************************************************************

__root const uVectorEntry __vector_table[] @ ".intvec" =
{
    { .ulPtr = (unsigned long)pulStack + sizeof(pulStack) },
                                            // The initial stack pointer
    __iar_program_start,                    // The reset handler
    NmiSR,                                  // The NMI handler
    FaultISR,                               // The hard fault handler
    IntDefaultHandler,                      // The MPU fault handler
    IntDefaultHandler,                      // The bus fault handler
    IntDefaultHandler,                      // The usage fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    vPortSVCHandler,                        // SVCall handler
    IntDefaultHandler,                      // Debug monitor handler
    0,                                      // Reserved
    xPortPendSVHandler,                     // The PendSV handler
    xPortSysTickHandler,                    // The SysTick handler
    IntDefaultHandler,                      // GPIO Port A
    IntDefaultHandler,                      // GPIO Port B
    IntDefaultHandler,                      // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                      // UART0 Rx and Tx
    IntDefaultHandler,                      // UART1 Rx and Tx
    IntDefaultHandler,                      // SSI Rx and Tx
    IntDefaultHandler,                      // I2C Master and Slave
    IntDefaultHandler,                      // PWM Fault
    IntDefaultHandler,                      // PWM Generator 0
    IntDefaultHandler,                      // PWM Generator 1
    IntDefaultHandler,                      // PWM Generator 2
    IntDefaultHandler,                      // Quadrature Encoder
    IntDefaultHandler,                      // ADC Sequence 0
    ADC0IntHandler,                     // ADC Sequence 1
    IntDefaultHandler,                      // ADC Sequence 2
    IntDefaultHandler,                      // ADC Sequence 3
    IntDefaultHandler,                      // Watchdog timer
    TimerAIntHandler,                       // Timer 0 subtimer A
    IntDefaultHandler,                      // Timer 0 subtimer B
    IntDefaultHandler,                      // Timer 1 subtimer A
    IntDefaultHandler,                      // Timer 1 subtimer B
    IntDefaultHandler,                      // Timer 2 subtimer A
    IntDefaultHandler,                      // Timer 2 subtimer B
```

```
    IntDefaultHandler,                      // Analog Comparator 0
    IntDefaultHandler,                      // Analog Comparator 1
    IntDefaultHandler,                      // Analog Comparator 2
    IntDefaultHandler,                      // System Control (PLL, OSC, BO)
    IntDefaultHandler,                      // FLASH Control
    IntDefaultHandler,                      // GPIO Port F
    IntDefaultHandler,                      // GPIO Port G
    IntDefaultHandler,                      // GPIO Port H
    IntDefaultHandler,                      // UART2 Rx and Tx
    IntDefaultHandler,                      // SSI1 Rx and Tx
    IntDefaultHandler,                    // Timer 3 subtimer A
    IntDefaultHandler,                      // Timer 3 subtimer B
    IntDefaultHandler,                      // I2C1 Master and Slave
    IntDefaultHandler,                      // Quadrature Encoder 1
    IntDefaultHandler,                      // CAN0
    IntDefaultHandler,                      // CAN1
    IntDefaultHandler,                      // CAN2
    lwIPEthernetIntHandler,                 // Ethernet
    IntDefaultHandler,                      // Hibernate
    IntDefaultHandler,                      // USB0
    IntDefaultHandler,                      // PWM Generator 3
    IntDefaultHandler,                      // uDMA Software Transfer
    IntDefaultHandler                       // uDMA Error
};


//
    *****************************************************************************
//
// This is the code that gets called when the processor receives a NMI.
    This
// simply enters an infinite loop, preserving the system state for
    examination
// by a debugger.
//
//
    *****************************************************************************
static void
NmiSR(void)
{
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

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

```
//
// This is the code that gets called when the processor receives a fault
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
FaultISR(void)
{
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives an unexpected
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
IntDefaultHandler(void)
{
    //
    // Go into an infinite loop.
    //
    while(1)
    {
    }
}
```

D.7 Tasks

D.7.1 Task Control Blocks

../code/tcb.h

```
/*
 * task.h
```

```
 * Author(s): Jonathan Ellington
 * 1/28/2014
 *
 * Defines the interface for a task
 */

#ifndef _TASK_H
#define _TASK_H

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

#endif // _TASK_H
```

D.7.2 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();

void DisplayIPAddress();

void lwIPHostTimerHandler();
```

../code/startup.c

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

```
#include <String.h>
#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"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_nvic.h"
#include "inc/hw_types.h"
#include "driverlib/ethernet.h"
#include "driverlib/flash.h"
#include "driverlib/gpio.h"
#include "utils/locator.h"
#include "utils/lwiplib.h"
#include "utils/uartstdio.h"
#include "utils/ustdlib.h"
#include "httpserver_raw/httpd.h"
#include "drivers/rit128x96x4.h"
#include "driverlib/timer.h"



//*****************************************************************************
//
// Defines for setting up the system clock.
//
//*****************************************************************************
#define SYSTICKHZ               100
#define SYSTICKMS               (1000 / SYSTICKHZ)
#define SYSTICKUS               (1000000 / SYSTICKHZ)
#define SYSTICKNS               (1000000000 / SYSTICKHZ)

//*****************************************************************************
//
// A set of flags.  The flag bits are defined as follows:
//
//     0 -> An indicator that a SysTick interrupt has occurred.
//
//*****************************************************************************
#define FLAG_SYSTICK            0
static volatile unsigned long g_ulFlags;
```

```c

//*****************************************************************************
//
// External Application references.
//
//*****************************************************************************
extern void httpd_init(void);


//*****************************************************************************
//
// Timeout for DHCP address request (in seconds).
//
//*****************************************************************************
#ifndef DHCP_EXPIRE_TIMER_SECS
#define DHCP_EXPIRE_TIMER_SECS  45
#endif

void timer1IntHandler (void) {

    TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
    //
    // Indicate that a SysTick interrupt has occurred.
    //
    HWREGBITW(&g_ulFlags, FLAG_SYSTICK) = 1;

    //
    // Call the lwIP timer handler.
    //
    lwIPTimer(SYSTICKMS);
}


/*
 * Initializes hw counter ans system state variables
 */
void startup() {

      SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
      SYSCTL_XTAL_8MHZ);


    RIT128x96x4Init(1000000);

  // Initialize global data
```

```
 initializeGlobalData();    // from globals.h
   initializeTimebase();



//////////////////////////////////////////
//// Web server initialization stuff /////
//////////////////////////////////////////
 unsigned long ulUser0, ulUser1;
 unsigned char pucMACArray[8];

   // Enable and Reset the Ethernet Controller.
   SysCtlPeripheralEnable(SYSCTL_PERIPH_ETH);
   SysCtlPeripheralReset(SYSCTL_PERIPH_ETH);

   // Enable Port F for Ethernet LEDs.
   //  LED0         Bit 3    Output
   //  LED1         Bit 2    Output
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
   GPIOPinTypeEthernetLED(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);

   // Configure timer for a periodic interrupt.
   SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
 TimerDisable(TIMER1_BASE, TIMER_BOTH);
 TimerConfigure(TIMER1_BASE, TIMER_CFG_32_BIT_PER );
 TimerLoadSet(TIMER1_BASE, TIMER_A, (SysCtlClockGet() / SYSTICKHZ));
       TimerIntRegister(TIMER1_BASE,TIMER_A, timer1IntHandler);
       TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
       TimerEnable(TIMER1_BASE, TIMER_A);

 IntMasterEnable();

   FlashUserGet(&ulUser0, &ulUser1);
   if((ulUser0 == 0xffffffff) || (ulUser1 == 0xffffffff))
   {
       // We should never get here.  This is an error if the MAC address
       // has not been programmed into the device.  Exit the program.
       RIT128x96x4StringDraw("MAC Address", 0, 76, 15);
       RIT128x96x4StringDraw("Not Programmed!", 0, 86, 15);
       while(1);
   }

   pucMACArray[0] = ((ulUser0 >>  0) & 0xff);
   pucMACArray[1] = ((ulUser0 >>  8) & 0xff);
   pucMACArray[2] = ((ulUser0 >> 16) & 0xff);
   pucMACArray[3] = ((ulUser1 >>  0) & 0xff);
   pucMACArray[4] = ((ulUser1 >>  8) & 0xff);
   pucMACArray[5] = ((ulUser1 >> 16) & 0xff);

   // Initialze the lwIP library, using DHCP.
   lwIPInit(pucMACArray, 0, 0, 0, IPADDR_USE_DHCP);

   // Setup the device locator service.
```

```
    LocatorInit();
    LocatorMACAddrSet(pucMACArray);
    LocatorAppTitleSet("EK-LM3S8962 enet_io");

    // Initialize a sample httpd server.
    httpd_init();
}

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

//
// Display an lwIP type IP Address.
//
//
    *****************************************************************************

void DisplayIPAddress(unsigned long ipaddr, unsigned long ulCol,
                      unsigned long ulRow)
{
    char pucBuf[16];
    unsigned char *pucTemp = (unsigned char *)&ipaddr;

    //
    // Convert the IP Address into a string.
    //
    usprintf(pucBuf, "%d.%d.%d.%d", pucTemp[0], pucTemp[1], pucTemp[2],
             pucTemp[3]);

    //
    // Display the string.
    //
    RIT128x96x4StringDraw(pucBuf, ulCol, ulRow, 15);
}

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

//
// Required by lwIP library to support any host-related timer functions.
//
//
    *****************************************************************************

void lwIPHostTimerHandler(void)
{
    static unsigned long ulLastIPAddress = 0;
    unsigned long ulIPAddress;

    ulIPAddress = lwIPLocalIPAddrGet();

    //
```

```
    // If IP Address has not yet been assigned, update the display accordingly
    //
    if(ulIPAddress == 0)
    {
        static int iColumn = 6;

        //
        // Update status bar on the display.
        //
//          RIT128x96x4Enable(1000000);
        if(iColumn < 12)
        {
            RIT128x96x4StringDraw(" >", 114, 86, 15);
            RIT128x96x4StringDraw("< ", 0, 86, 15);
            RIT128x96x4StringDraw("*",iColumn, 86, 7);
        }
        else
        {
            RIT128x96x4StringDraw(" *",iColumn - 6, 86, 7);
        }

        iColumn += 4;
        if(iColumn > 114)
        {
            iColumn = 6;
            RIT128x96x4StringDraw(" >", 114, 86, 15);
        }
        //RIT128x96x4Disable();
    }

    //
    // Check if IP address has changed, and display if it has.
    //
    else if(ulLastIPAddress != ulIPAddress)
    {
        ulLastIPAddress = ulIPAddress;
//          RIT128x96x4Enable(1000000);
//          RIT128x96x4StringDraw("                      ", 0, 16, 15);
        RIT128x96x4StringDraw("                      ", 0, 86, 15);
        RIT128x96x4StringDraw("IP:   ", 0, 86, 15);
//          RIT128x96x4StringDraw("MASK: ", 0, 24, 15);
//          RIT128x96x4StringDraw("GW:   ", 0, 32, 15);
        DisplayIPAddress(ulIPAddress, 36, 86);
//          ulIPAddress = lwIPLocalNetMaskGet();
//          DisplayIPAddress(ulIPAddress, 36, 24);
//          ulIPAddress = lwIPLocalGWAddrGet();
//          DisplayIPAddress(ulIPAddress, 36, 32);
//          RIT128x96x4Disable();
    }
}
```

D.7.3 Measure Task

../code/measure.h

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

#include "tcb.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
 */

#define DEBUG_MEASURE 0

#include "FreeRTOS.h"
#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"
#include "driverlib/adc.h"

// Used for debug display
#if DEBUG_MEASURE
#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
```

```c
#include "compute.h"
#include "ekgCapture.h"
#endif

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

// Internal data structure
typedef struct measureData {
  CircularBuffer *temperatureRaw;
  CircularBuffer *systolicPressRaw;
  CircularBuffer *diastolicPressRaw;
  CircularBuffer *pulseRateRaw;
  unsigned short *measureSelect;
  tBoolean *ekgCaptureDone;
} MeasureData;

static int pulseRate = 0;
static MeasureData data;  // internal data
TCB measureTask = {&measureRunFunction, &data};  // task interface
extern xTaskHandle ekgCaptureHandle;

void initializeMeasureTask() {
  // Load data memory
  data.temperatureRaw = &(global.temperatureRaw);
  data.systolicPressRaw = &(global.systolicPressRaw);
  data.diastolicPressRaw = &(global.diastolicPressRaw);
  data.pulseRateRaw = &(global.pulseRateRaw);
  data.measureSelect = &(global.measurementSelection);
  data.ekgCaptureDone = &(global.ekgCaptureDone);

    //setup for temperature sensor
    ADCSequenceDisable(ADC0_BASE, 1);
    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 1);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_END | ADC_CTL_TS);

  /* Interrupt setup
   * Note: using statically registered interrupts, because they're faster
   *       this means we aren't using the dynamic GPIOPortIntRegister()
   function,
   *       instead, an entry in the interrupt table is populated with the
   address
   *       of the interrupt handler (under GPIO Port A) and this is enabled
   with
   *       IntEnable(INT_GPIOA) */

  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

  // Set PA4 as input
  GPIOPinTypeGPIOInput(GPIO_PORTA_BASE, GPIO_PIN_4);

  // Enable interrupts on PA4
```

```
  GPIOPinIntEnable(GPIO_PORTA_BASE, GPIO_PIN_4);

  // Set interrupt type
  GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_4, GPIO_RISING_EDGE);

  // Enable interrupts to the processor.
  IntMasterEnable();

  // Enable the interrupts.
  IntEnable(INT_GPIOA);
}

void setTemp(CircularBuffer *tbuf) {
  int temp = 0;
  ADCSequenceEnable(ADC0_BASE, 1);

  // Trigger the sample sequence.
  ADCProcessorTrigger(ADC0_BASE, 1);

  // Wait until the sample sequence has completed.
  while( 1 != ADCSequenceDataGet(ADC0_BASE, 1, &temp));

  ADCSequenceDisable(ADC0_BASE, 1);
  cbAdd(tbuf, &temp);
}

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

  static unsigned int i = 0;

  static tBoolean sysComplete = false;
  static tBoolean diaComplete = false;

  int syspress = *(int *)cbGet(spbuf);

  // Restart systolic measurement if diastolic is complete

  if (syspress > 100)
    sysComplete = true;

  if (! sysComplete) {
    if (i%2==0) syspress+=3;
    else syspress--;
  }

  int diapress = *(int *)cbGet(dpbuf);

  // Restart diastolic measurement if systolic is complete
```

```
  if (diapress < 40)
    diaComplete = true;

  if (!diaComplete) {
    if (i%2==0) diapress-=2;
    else diapress++;
  }

  if (diaComplete && sysComplete) {
    sysComplete = false;
    diaComplete = false;
    syspress = SYS_RAW_INIT;
    diapress = DIA_RAW_INIT;
  }

  cbAdd(spbuf, &syspress);
  cbAdd(dpbuf, &diapress);

  i++;
}

// pulse rate interrupt handler
void PulseRateISR(void) {
  pulseRate++;

  GPIOPinIntClear(GPIO_PORTA_BASE, GPIO_PIN_4);
}

void measureRunFunction(void *dataptr) {
  static tBoolean onFirstRun = true;
  static int rate;
  MeasureData *mData = (MeasureData *) dataptr;

  if (onFirstRun) {
    initializeMeasureTask();
    rate = *(int*) cbGet(mData->pulseRateRaw);
    onFirstRun = false;
  }

  // capture pulse rate
  if (IS_PULSE_CYCLE) {
    // Divide by two so raw pulse rate matches frequency
    rate = pulseRate/2;
    pulseRate = 0;
  }

  // only run on major cycle
  short measureSelect = *(mData->measureSelect);
//   if(measureSelect == 0 || measureSelect == 1)
//   {
    setTemp(mData->temperatureRaw);
//     }
//   if(measureSelect == 0 || measureSelect == 2)
```

```
//   {
     setBloodPress(mDataー>systolicPressRaw, mDataー>diastolicPressRaw);
//   }
//   if(measureSelect == 0 || measureSelect == 3)
//   {
       int prev = *(int*) cbGet(mDataー>pulseRateRaw);

       // Only save if +- 15%
       if (rate < prev*0.85 || rate > prev*1.15) {
         cbAdd(mDataー>pulseRateRaw, (void *)&rate);
       }
//   }
//   if(measureSelect == 0 || measureSelect == 4)
//   {
     *(mDataー>ekgCaptureDone) = false;
//     vTaskResume(ekgCaptureHandle);
#if DEBUG_MEASURE
     RIT128x96x4StringDraw("ekgCapture go!", 0, 50, 15);
#endif
//   }
//   else
//   {
//     vTaskSuspend(ekgCaptureHandle);
#if DEBUG_MEASURE
     RIT128x96x4StringDraw("no ekg!", 0, 50, 15);
#endif
//   }

#if DEBUG_MEASURE
     char num[30];
     int temp = *(int *)cbGet(mDataー>temperatureRaw);
     int sys = *(int *)cbGet(mDataー>systolicPressRaw);
     int dia = *(int *)cbGet(mDataー>diastolicPressRaw);
     int pulse = *(int *)cbGet(mDataー>pulseRateRaw);
     int batt = global.batteryState;

     usnprintf(num, 30, "<ーー MEASURE DEBUG ーー>");
     RIT128x96x4StringDraw(num, 0, 0, 15);

     usnprintf(num, 30, "Raw temp: %d  ", temp);
     RIT128x96x4StringDraw(num, 0, 10, 15);

     usnprintf(num, 30, "Raw Syst: %d  ", sys);
     RIT128x96x4StringDraw(num, 0, 20, 15);

     usnprintf(num, 30, "Raw Dia: %d  ", dia);
     RIT128x96x4StringDraw(num, 0, 30, 15);

     usnprintf(num, 30, "Raw Pulse: %d  ", pulse);
     RIT128x96x4StringDraw(num, 0, 40, 15);

     usnprintf(num, 30, "Raw Batt: %d  ", batt);
     RIT128x96x4StringDraw(num, 0, 50, 15);
```

```
#endif

    vTaskResume(computeHandle);   // run the compute task
}
```

D.7.4 Compute Task

../code/compute.h

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

#include "tcb.h"

/* Points to the TCB for compute */
extern TCB computeTask;
```

../code/compute.c

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

#define DEBUG_COMPUTE 0

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

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

// computeData structure internal to compute task
typedef struct computeData {
  // raw data pointers
  CircularBuffer *temperatureRaw;
```

```
  CircularBuffer *systolicPressRaw;
  CircularBuffer *diastolicPressRaw;
  CircularBuffer *pulseRateRaw;

  //corrected data pointers
  CircularBuffer *temperatureCorrected;
  CircularBuffer *systolicPressCorrected;
  CircularBuffer *diastolicPressCorrected;
  CircularBuffer *pulseRateCorrected;

  tBoolean *measurementComplete;
  unsigned short *measurementSelect;
  tBoolean *ekgCaptureDone;
  tBoolean *ekgProcessDone;
} ComputeData;

void computeRunFunction(void *computeData);

static ComputeData data;    // the internal data
TCB computeTask = {&computeRunFunction, &data};  // task interface

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

  data.temperatureCorrected = &(global.temperatureCorrected);
  data.systolicPressCorrected = &(global.systolicPressCorrected);
  data.diastolicPressCorrected = &(global.diastolicPressCorrected);
  data.pulseRateCorrected = &(global.pulseRateCorrected);

  data.measurementComplete = &(global.measurementComplete);
  data.measurementSelect = &(global.measurementSelection);
  data.ekgCaptureDone = &(global.ekgCaptureDone);
  data.ekgProcessDone = &(global.ekgProcessDone);
}

/*
 * Linearizes the raw data measurement and converts value into human
 * readable format
 */
void computeRunFunction(void *computeData) {
  static tBoolean onFirstRun = true;

  if (onFirstRun) {
    initializeComputeTask();
    onFirstRun = false;
  }

```

```
  ComputeData *cData = (ComputeData *) computeData;

  float temp = (5 + 0.75 * (*(int*)cbGet(cData->temperatureRaw)))/10;
  float systolic = 9 + 2 * (*(int*)cbGet(cData->systolicPressRaw));
  float diastolic = 6 + 1.5 * (*(int*)cbGet(cData->diastolicPressRaw));
  float pulseRate = 8 + 3 * (*(int*)cbGet(cData->pulseRateRaw));

  cbAdd(cData->temperatureCorrected, &temp);
  cbAdd(cData->systolicPressCorrected, &systolic);
  cbAdd(cData->diastolicPressCorrected, &diastolic);
  cbAdd(cData->pulseRateCorrected, &pulseRate);

//  if (0 == *(cData->measurementSelect) || 4 == *(cData->measurementSelect)
    ) {
//    while(!(cData->ekgCaptureDone)) { // wait until ekg captured
//    }
//  vTaskResume(ekgProcessHandle);
//
//    RIT128x96x4StringDraw("go ekgProcess", 0, 60, 15);
//    *(cData->ekgProcessDone) = false;
//    while (!*(cData->ekgProcessDone)) { // wait until ekg computed
//    }
//  }
  *(cData->measurementComplete) = true;

  vTaskSuspend(NULL);  // suspend self

#if DEBUG_COMPUTE
  char num[30];

  usnprintf(num, 30, "<-- COMPUTE DEBUG -->");
  RIT128x96x4StringDraw(num, 0, 0, 15);

  usnprintf(num, 30, "Corrected temp: %d", (unsigned int) temp);
  RIT128x96x4StringDraw(num, 0, 10, 15);

  usnprintf(num, 30, "Corrected Syst: %d", (unsigned int) systolic);
  RIT128x96x4StringDraw(num, 0, 20, 15);

  usnprintf(num, 30, "Corrected Dia: %d", (unsigned int) diastolic);
  RIT128x96x4StringDraw(num, 0, 30, 15);

  usnprintf(num, 30, "Corrected Pulse: %d", (unsigned int) pulseRate);
  RIT128x96x4StringDraw(num, 0, 40, 15);
#endif
}
```

D.7.5 Keypad Task

../code/keypad.h

```
/*
```

```
 * keyPad.h
 * Author(s): Jarrett Gaddy
 * 2/10/2014
 *
 * Defines the public interface for the keyPad task
 *
 * initializeKeyPadTask() should be called once before performing runkeyPad
 * functions
 */

#include "tcb.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);



   GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD_WPU);
   GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_DIR_MODE_IN);

   GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_4, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD);
   GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_4, GPIO_DIR_MODE_IN);

   GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_5, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD);
   GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_5, GPIO_DIR_MODE_IN);

   GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD);
   GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_DIR_MODE_IN);

   GPIOPadConfigSet(GPIO_PORTD_BASE, GPIO_PIN_7, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD);
   GPIODirModeSet(GPIO_PORTD_BASE, GPIO_PIN_7, GPIO_DIR_MODE_IN);

   //for ack switch
   /* GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_STRENGTH_2MA,
     GPIO_PIN_TYPE_STD_WPU);
   GPIODirModeSet(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_DIR_MODE_IN);*/



  // Load data
  data.mode = &(global.mode);
  data.measurementSelection = &(global.measurementSelection);
  data.scroll = &(global.scroll);
  data.alarmAcknowledge = &(global.alarmAcknowledge);
  data.select = &(global.select);

  // Load TCB
  keyPadTask.runTaskFunction = &keyPadRunFunction;
  keyPadTask.taskDataPtr = &data;
}

/* Perform status tasks */
void keyPadRunFunction(void *keyPadData){
```

```

  static tBoolean onFirstRun = true;

  if (onFirstRun) {
    initializeKeyPadTask();
    onFirstRun = false;
  }
  KeyPadData *kdata = (KeyPadData *) keyPadData;
  if( 0 == *(kdata->mode))
  {

    if(false == *(kdata->select))
    {

      if(UP_SW)
        *(kdata->scroll) = *(kdata->scroll) +1;
      else if(DOWN_SW)
        *(kdata->scroll) = *(kdata->scroll) -1;
      else if(RIGHT_SW)
      {
        *(kdata->select) = true;
        *(kdata->measurementSelection) = (*(kdata->scroll) %5) + 1;
      }

            else if(LEFT_SW)
            {
        *(kdata->mode) = 1;
            }

    }
    else
    {
            if(LEFT_SW)
      {
        *(kdata->select) = false;
        *(kdata->measurementSelection) = 0;
      }
    }
  }
  else
  {
    if(RIGHT_SW)
    {
      *(kdata->mode) = 0;
    }
  }
  if(1 == ACK_SW)
  {
    *(kdata->alarmAcknowledge) = true;
  }
  else
    *(kdata->alarmAcknowledge) = false;
}
```

D.7.6 Display Task

../code/display.h

```c
/*
 * display.h
 * Author(s): Jarrett Gaddy
 * 1/28/2014, updated 2/10/2014
 *
 * Defines the interface for the displayTask.
 * initializeDisplayData() should be called before running displayTask()
 */

#include "tcb.h"

/* Initialize DisplayData, must be done before running displayTask() */
void initializeDisplayTask();

/* Points to the TCB for display */
extern TCB displayTask;
```

../code/display.c

```c
/*
 * display.c
 * Author(s): jarrett Gaddy
 * 2/10/2014
 *
 * Implements display.h
 */

#include "globals.h"
#include "timebase.h"
#include "display.h"
#include "inc/hw_types.h"
#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
#include <stdlib.h>

#define DISPLAY_OFF 0

// Internal data structure
typedef struct oledDisplayData {
  CircularBuffer *temperatureCorrected;
  CircularBuffer *systolicPressCorrected;
  CircularBuffer *diastolicPressCorrected;
  CircularBuffer *pulseRateCorrected;
  CircularBuffer *ekgFrequencyResult;
  unsigned short *batteryState;
  unsigned short *mode;
  unsigned short *measurementSelection;
```

```
  unsigned short *scroll;
  tBoolean *alarmAcknowledge;
  tBoolean *select;
  unsigned short *selection;
} DisplayData;

void displayRunFunction(void *dataptr);  // prototype for compiler

static DisplayData data;  // internal data
TCB displayTask = {&displayRunFunction, &data};          // task interface

void initializeDisplayTask() {
  // Load data
  data.temperatureCorrected = &(global.temperatureCorrected);
  data.systolicPressCorrected = &(global.systolicPressCorrected);
  data.diastolicPressCorrected = &(global.diastolicPressCorrected);
  data.pulseRateCorrected = &(global.pulseRateCorrected);
  data.batteryState = &(global.batteryState);
  data.ekgFrequencyResult = &(global.ekgFrequencyResult);

  data.mode = &(global.mode);
  data.measurementSelection = &(global.measurementSelection);
  data.scroll = &(global.scroll);
  data.alarmAcknowledge = &(global.alarmAcknowledge);
  data.select = &(global.select);
}


void displayRunFunction(void *dataptr) {
  static tBoolean onFirstRun = true;

  if (onFirstRun) {
    initializeDisplayTask();
    onFirstRun = false;
  }

  DisplayData *dData = (DisplayData *) dataptr;

  tBoolean selection = *(dData->select);
  int scroll = *(dData->scroll);

#if !DISPLAY_OFF
  char num[40];
  //char buf1[30];
  char buf2[30];

  if(0 == *(dData->mode))
  {
    if(false == selection)
    {
      RIT128x96x4StringDraw("Make Selection                    ", 0, 0,
       15);
```

```
    RIT128x96x4StringDraw("   Blood Pressure                         ", 0, 10, 15);
    RIT128x96x4StringDraw("   Temperature                            ", 0, 20, 15);
    RIT128x96x4StringDraw("   Pulse Rate                             ", 0, 30, 15);
    RIT128x96x4StringDraw("   EKG                                    ", 0, 40, 15);
    RIT128x96x4StringDraw("   Battery                                ", 0, 50, 15);
    RIT128x96x4StringDraw("                                          ", 0, 60, 15);
  RIT128x96x4StringDraw("                                          ", 0, 70, 15);
    RIT128x96x4StringDraw("->", 0, 10*((scroll%5+1)), 15);
  }
  else
  {
    RIT128x96x4StringDraw("                                          ", 0, 0, 15);
    if(0 == scroll%5)
      RIT128x96x4StringDraw("Blood Pressure:", 0, 0, 15);
    else if(1 == scroll%5)
      RIT128x96x4StringDraw("Temperature:", 0, 0, 15);
    else if(2 == scroll%5)
      RIT128x96x4StringDraw("Pulse Rate:", 0, 0, 15);
    else if(3 == scroll%5)
      RIT128x96x4StringDraw("EKG:", 0, 0, 15);
  else if(4 == scroll%5)
      RIT128x96x4StringDraw("Battery:", 0, 0, 15);

    else RIT128x96x4StringDraw("oops", 0, 0, 15);//just in case



    if(0 == scroll%5)
      usnprintf(buf2,30, "Systolic: %d mm Hg ", (int) *( (float*) cbGet(dData->systolicPressCorrected)));
    else if(1 == scroll%5)
      usnprintf(buf2,30,"%d C ", (int) *( (float*) cbGet(dData->temperatureCorrected)));
    else if(2 == scroll%5)
      usnprintf(buf2,30, "%d BPM ", (int) *( (float*) cbGet(dData->pulseRateCorrected)));
  else if(3 == scroll%5)
      usnprintf(buf2,30, "%d Hz ", (int) *( (float*) cbGet(dData->ekgFrequencyResult)));
    else if(4 == scroll%5)
      usnprintf(buf2,30, "%d %%  ", (int) *(dData->batteryState)/2);
    //else buf2 = "oops"; //just in case

    RIT128x96x4StringDraw("                                          ", 0, 10, 15);
```

```
    RIT128x96x4StringDraw(buf2, 0, 10, 15);
    if(0 == scroll%5)
    {
      usnprintf(buf2,30, "Diastolic: %d mm Hg        ", (int)*( (float*) cbGet(dData->diastolicPressCorrected)));

      RIT128x96x4StringDraw(buf2, 0, 20, 15);
    }
    else RIT128x96x4StringDraw("                    ", 0, 20, 15);
    RIT128x96x4StringDraw("                    ", 0, 30, 15);
    RIT128x96x4StringDraw("                    ", 0, 40, 15);
    RIT128x96x4StringDraw("                    ", 0, 50, 15);
    RIT128x96x4StringDraw("                    ", 0, 60, 15);
  RIT128x96x4StringDraw("                    ", 0, 70, 15);

  }
}
else//(1 == *(data->mode)
{
  usnprintf(num,40,"Temperature: %d C          ", (int) *( (float*) cbGet(dData->temperatureCorrected)));
  RIT128x96x4StringDraw(num, 0, 0, 15);

  usnprintf(num,40, "Systolic Pressure:         ");
  RIT128x96x4StringDraw(num, 0, 10, 15);

  usnprintf(num,40, "%d mm Hg                   ", (int) *( (float*) cbGet(dData->systolicPressCorrected)));
  RIT128x96x4StringDraw(num, 0, 20, 15);

  usnprintf(num,40, "Diastolic Pressure:        ");
  RIT128x96x4StringDraw(num, 0, 30, 15);

  usnprintf(num,40, "%d mm Hg                   ",(int) *( (float*) cbGet(dData->diastolicPressCorrected)));
  RIT128x96x4StringDraw(num, 0, 40, 15);

  usnprintf(num,40, "Pulse rate: %d BPM         ",(int) *( (float*) cbGet(dData->pulseRateCorrected)));
  RIT128x96x4StringDraw(num, 0, 50, 15);

usnprintf(num,40, "EKG: %d Hz                 ", *((int *) cbGet(dData->ekgFrequencyResult)));
  RIT128x96x4StringDraw(num, 0, 60, 15);

  usnprintf(num,40, "Battery: %d %%             ",(int) *(dData->batteryState)/2);
```

```
    RIT128x96x4StringDraw(num,0, 70,15);
  }
#endif
}
```

D.7.7 Warning/Alarm Task

../code/warning.h

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

#include "tcb.h"

#define WARN_LOW 0.95 //warn at 5% below min range value
#define WARN_HIGH 1.05 //warn at 5% above max range value
#define ALARM_LOW 0.90 //alarm at 10% below min range value
#define ALARM_HIGH 1.20 //alarm at 20% above max range value


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

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

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

/* The warning task */
extern TCB warningTask;
```

../code/warning.c

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

```
 */

#define DEBUG_WARNING 0
#define ALARM_OFF 0

#include "globals.h"
#include "timebase.h"
#include "warning.h"
#include "schedule.h"
#include "inc/hw_types.h"
#include <stdlib.h>
#include <stdio.h>

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

#if DEBUG_WARNING
#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
#endif

// alarm cycle period (on/off) in millisecond
#define ALARM_PERIOD 2000
#define ALARM_CYCLE_RATE     (ALARM_PERIOD / MINOR_CYCLE / 2)

// duration to sleep in terms of minor cycles
#define ALARM_SLEEP_PERIOD    (5 * MAJOR_CYCLE)

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

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

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

// compiler prototypes
void warningRunFunction(void *dataptr);  // prototype for compiler

// Internal data structure
typedef struct WarningData {
  CircularBuffer *temperatureCorrected;
  CircularBuffer *systolicPressCorrected;
  CircularBuffer *diastolicPressCorrected;
  CircularBuffer *pulseRateCorrected;
  unsigned short *batteryState;
} WarningData;
```

```c

extern tBoolean serialActive;
static unsigned long ulPeriod; // sets the alarm tone period
static WarningData data;         // internal data

TCB warningTask = {&warningRunFunction, &data};  // task interface


/*
 * initializes task variables
 */
void initializeWarningTask() {
  //
  // Enable the peripherals used by this code. I.e enable the use of pin banks, etc.
  //
  SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);        // bank C
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);        // bank E
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);        // bank F
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);        // bank G


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

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

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

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


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

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

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

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

  //
  // Compute the PWM period based on the system clock.
  //
  ulPeriod = SysCtlClockGet() / 65;

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

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

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

  // initialize the warning data pointers
  data.temperatureCorrected = &(global.temperatureCorrected);
  data.systolicPressCorrected = &(global.systolicPressCorrected);
  data.diastolicPressCorrected = &(global.diastolicPressCorrected);
  data.pulseRateCorrected = &(global.pulseRateCorrected);
  data.batteryState = &(global.batteryState);

}

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

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

  static alarmState aState = OFF;
```

```
static warningState wState = NONE;
static batteryState bState = NORMAL;

static warningState prevState;
prevState = wState;

static int wakeUpAlarmAt = 0;

static tBoolean onFirstRun = true;

if (onFirstRun){
  initializeWarningTask();
  onFirstRun = false;
}


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

// Alarm condition
if ( (sysPress > SYS_MAX*ALARM_HIGH) ) { //|| // Commented lines = lab2
    // (temp < TEMP_MIN*ALARM_LOW || temp > (TEMP_MAX*ALARM_HIGH)) ||
    // (diaPress > DIA_MAX*ALARM_HIGH) ||
    // (pulse < PULSE_MIN*ALARM_LOW || pulse > PULSE_MAX*ALARM_HIGH) ) {

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

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

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

// Handle speaker, based on alarm state
static tBoolean pwmEnable = false;
```

```
  switch (aState) {
    case ON:

#if !ALARM_OFF
      if (0 == (minor_cycle_ctr % ALARM_CYCLE_RATE)) { // toggle between on/off
        if (pwmEnable)
          PWMGenEnable(PWM0_BASE, PWM_GEN_0);
        else
          PWMGenDisable(PWM0_BASE, PWM_GEN_0);
        pwmEnable = !pwmEnable;
      }
#endif
      break;
    case ASLEEP:
      PWMGenDisable(PWM0_BASE, PWM_GEN_0);
      break;
    default:  // OFF
      PWMGenDisable(PWM0_BASE, PWM_GEN_0);

      break;
  }

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

      if (wState != prevState) {
        GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
        toggletime = WARN_RATE_PRESS;
      }
      else if (0 == minor_cycle_ctr%toggletime) {
        if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
        else
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00);
      }

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

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

```
      }
      break;
    case WARN_PULSE:
      GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0X00);

      if (wState != prevState) {
        GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
        toggletime = WARN_RATE_PULSE;
      }
      else if (0==minor_cycle_ctr%toggletime) {
        if (GPIOPinRead(GPIO_PORTC_BASE, LED_RED) == 0)
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0XFF);
        else
          GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0X00);
      }
      break;
    default:  // NORMAL
      GPIOPinWrite(GPIO_PORTC_BASE, LED_GREEN, 0xFF);
      GPIOPinWrite(GPIO_PORTC_BASE, LED_RED, 0x00);
      break;
  }

  // activate the remote terminal task if in ANY warn or alarm state
  if (NONE == wState && OFF == aState)
    vTaskSuspend(serialHandle);
  else
    vTaskResume(serialHandle);

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

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

  // Check whether to resound alarm
  if (minor_cycle_ctr == wakeUpAlarmAt && aState == ASLEEP) {
```

```
    aState = ON;
    //GPIOPinWrite(GPIO_PORTC_BASE, LED_YELLOW, 0X00);  // for debug, kills
    led
  }

#if DEBUG_WARNING
    char num[30];

    usnprintf(num, 30, "<-- WARNING DEBUG -->");
    RIT128x96x4StringDraw(num, 0, 0, 15);

    usnprintf(num, 30, "Cor temp: %d  ", (int) temp);
    RIT128x96x4StringDraw(num, 0, 10, 15);

    usnprintf(num, 30, "Cor Syst: %d  ", (int) sysPress);
    RIT128x96x4StringDraw(num, 0, 20, 15);

    usnprintf(num, 30, "Cor Dia: %d  ", (int) diaPress);
    RIT128x96x4StringDraw(num, 0, 30, 15);

    usnprintf(num, 30, "Cor Pulse: %d  ", (int) pulse);
    RIT128x96x4StringDraw(num, 0, 40, 15);

    usnprintf(num, 30, "Cor Batt: %d  ", (unsigned short) battery);
    RIT128x96x4StringDraw(num, 0, 50, 15);


    usnprintf(num, 30, "aState: %d  ", aState);
    RIT128x96x4StringDraw(num, 0, 60, 15);

    usnprintf(num, 30, "alarmAck: %d  ", global.alarmAcknowledge);
    RIT128x96x4StringDraw(num, 0, 70, 15);

    usnprintf(num, 30, "pwmEn: %d  ", pwmEnable);
    RIT128x96x4StringDraw(num, 0, 80, 15);
#endif

}
```

D.7.8 Serial Task

../code/serial.h

```
/*
 * serial.h
 * Author(s): Jonathan Ellington
 * 2/05/2014
 *
 * Defines the serial communications task.  This sends various data over
 * RS-232.
 *
 * This function should only run if there is a warning (in other words, the
```

```
 * scheduler should add it to the task queue in the event of a warning), and
 * should subsequently delete itself from the task queue.
 */

#include "tcb.h"  // for TCBs

/* The status Task */
extern TCB serialTask;
```

../code/serial.c

```
/*
 * serial.c
 * Author(s): Jonathan Ellington
 * 2/05/2014
 *
 * Implements serial.c
 */

#include "globals.h"
#include "timebase.h"
#include "serial.h"
#include "schedule.h"
#include "CircularBuffer.h"
#include "inc/hw_types.h"
#include "driverlib/uart.h"
#include "driverlib/gpio.h"
#include "inc/hw_memmap.h"
#include "utils/ustdlib.h"
#include <string.h>


// Internal data structure
typedef struct serialData {
  CircularBuffer *temperatureCorrected;
  CircularBuffer *systolicPressCorrected;
  CircularBuffer *diastolicPressCorrected;
  CircularBuffer *pulseRateCorrected;
  CircularBuffer *ekgFrequencyResult;
  unsigned short *batteryState;
} SerialData;

// Prototype
void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount);
void serialRunFunction(void *dataptr);

static SerialData data;  // internal data
TCB serialTask = {&serialRunFunction, &data};          // task interface

void initializeSerialTask() {
  // Initialize Data
  data.temperatureCorrected = &(global.temperatureCorrected);
  data.systolicPressCorrected = &(global.systolicPressCorrected);
```

```
  data.diastolicPressCorrected = &(global.diastolicPressCorrected);
  data.pulseRateCorrected = &(global.pulseRateCorrected);
  data.batteryState = &(global.batteryState);
  data.ekgFrequencyResult = &(global.ekgFrequencyResult);
  // UART Stuff
  // Enable the peripherals used by this example.
  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

  // Set GPIO A0 (UART RX)
  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_1);

  // Configure the UART for 115,200, 8-N-1 operation.
  UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
      (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
       UART_CONFIG_PAR_NONE));

  UARTEnable(UART0_BASE);
}

void serialRunFunction(void *dataptr) {
  static tBoolean onFirstRun = true;
  SerialData *sData = (SerialData *) dataptr;
  if (onFirstRun) {
    initializeSerialTask();
    onFirstRun = false;
  }


  int temp = (int) *((float *)cbGet(sData->temperatureCorrected));
  int sys = (int) *((float *)cbGet(sData->systolicPressCorrected));
  int dia = (int) *((float *)cbGet(sData->diastolicPressCorrected));
  int pulse = (int) (*(float *)cbGet(sData->pulseRateCorrected));
  int EKG = *((int *)cbGet(sData->ekgFrequencyResult));
  int batt = *(data.batteryState);

  char buf[40];
  usnprintf(buf, 40, "\f1. Temperature:\t\t%d C\n\n\r", temp);
  UARTSend( (unsigned char *) buf, strlen(buf));

  usnprintf(buf, 40, "2. Systolic pressure:\t%d mm Hg\n\n\r", sys);
  UARTSend( (unsigned char *) buf, strlen(buf));

  usnprintf(buf, 40, "3. Diastolic pressure:\t%d mm Hg\n\n\r", dia);
  UARTSend( (unsigned char *) buf, strlen(buf));

  usnprintf(buf, 40, "4. Pulse rate:\t\t%d BPM\n\n\r", pulse);
  UARTSend( (unsigned char *) buf, strlen(buf));

  usnprintf(buf, 40, "5. EKG:\t\t\t%d Hz\n\n\r", EKG);
  UARTSend( (unsigned char *) buf, strlen(buf));

  usnprintf(buf, 40, "6. Battery:\t\t%d%%\n\n\r", batt/2);
```

```
  UARTSend( (unsigned char *) buf, strlen(buf));

  vTaskSuspend(NULL);    // suspend self
}

void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount) {
  // Loop while there are more characters to send.
  while(ulCount--)
  {
    // Write the next character to the UART.
    // This blocks while the FIFO queue is full
    UARTCharPut(UART0_BASE, *pucBuffer++);
  }
}
```

D.7.9 EKG Capture

../code/ekgCapture.c

```
/* Author: patrick Ma
 * 2/21/14
 *
 * ekgcapture.c
 *
 * Reads and stores data from ekg sensor via ADC. Stores data into a memory
 * buffer.
 */

#define DEBUG_EKG 0 // ekg task debug

#include "FreeRTOS.h"
#include "task.h"

#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/adc.h"   // for ADC use
#include "driverlib/timer.h"   // for hw timer use
#include "driverlib/interrupt.h"
#include "inc/hw_ints.h"

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

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

#if DEBUG_EKG
```

```
static char num[30];  // used for display
#endif

static tBoolean firstRun = true;
static tBoolean ekgComplete = false;
extern xTaskHandle ekgProcessHandle;

// ekgCapture data structure. Internal to the task
typedef struct ekgCaptureData {
  signed int *ekgRawDataAddr; // raw output array address
  tBoolean *ekgCaptureDone;
}EKGCaptureData;

static EKGCaptureData data;    // the interal data

void ekgCaptureRunFunction(void *ekgCaptureData);      // Compiler function prototypes
TCB ekgCaptureTask = {&ekgCaptureRunFunction, &data}; // task interface
void ADC0IntHandler();

/*
 * sets up task specific variables, etc
 */
void initializeEKGTask() {
#if DEBUG_EKG
  RIT128x96x4Init(1000000);
  RIT128x96x4StringDraw("* EKGCapture Debug *", 0, 0, 15);
#endif

  data.ekgRawDataAddr = global.ekgRaw;
  data.ekgCaptureDone = &(global.ekgCaptureDone);

  // Enable read from GPIO pin (move this and below to startup?)
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
  GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_7);

  // Setup ADC0 for EKG capture
        SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
        SysCtlADCSpeedSet(SYSCTL_ADCSPEED_500KSPS);
  ADCSequenceDisable(ADC0_BASE, EKG_SEQ);
  ADCSequenceConfigure(ADC0_BASE, EKG_SEQ, ADC_TRIGGER_PROCESSOR, 1);
  ADCSequenceStepConfigure(ADC0_BASE, EKG_SEQ, 0, ADC_CTL_IE | ADC_CTL_END |
    EKG_CH);
  ADCIntRegister(ADC0_BASE, EKG_SEQ, ADC0IntHandler);
        ADCIntEnable(ADC0_BASE, EKG_SEQ);

  // configure timer (uses both 16-bit timers) for periodic timing
//   SysCtlPeripheralEnable(EKG_TIMER_PERIPH);
//   TimerDisable(EKG_TIMER_BASE, EKG_TIMER);
//   TimerConfigure(EKG_TIMER_BASE, EKG_CFG );
//   TimerControlTrigger(EKG_TIMER_BASE, EKG_TIMER, true);
//   TimerLoadSet(EKG_TIMER_BASE, EKG_TIMER, (SysCtlClockGet() / SAMPLE_FREQ)
    );
```

```c
//   TimerEnable(EKG_TIMER_BASE, EKG_TIMER);

#if DEBUG_EKG
        long clk = SysCtlClockGet();
  long timeLoad =  TimerLoadGet(EKG_TIMER_BASE, EKG_TIMER);
  usnprintf(num, 30, "load: %ul", timeLoad);
  RIT128x96x4StringDraw(num, 0, 10, 15);
        usnprintf(num, 30, "clock: %ul", clk);
  RIT128x96x4StringDraw(num, 0, 20, 15);
#endif
}

void delay_in_ms(int ms) {
  for (volatile int i = 0; i < ms; i++)
    for (volatile int j = 0; j < 500; j++);
}

/*
 * captures a sequence of samples (given by NUM_EKG_SAMPLES) via the ADC and
 * stores the results into a buffer
 */
void ekgCaptureRunFunction(void *ekgCaptureData) {
  if (firstRun) {
    firstRun = false;
    initializeEKGTask();
  }

  EKGCaptureData *eData = (EKGCaptureData *) ekgCaptureData;
  ekgComplete = false;  // reset the adc counters

  ADCSequenceEnable(ADC0_BASE, EKG_SEQ);
  while (!ekgComplete) {
        ADCProcessorTrigger(ADC0_BASE, EKG_SEQ);
        delay_in_ms(1);
  }
  ADCSequenceDisable(ADC0_BASE, EKG_SEQ);

  *(eData->ekgCaptureDone) = true;  // we want to process our measurement

#if DEBUG_EKG
  usnprintf(num, 30, "end ADC get");
  RIT128x96x4StringDraw(num, 0, 40, 15);

  // let's check our values

  int a = global.ekgRaw[0];
  int b = global.ekgRaw[50];
  int c = (int) *(data.ekgRawDataAddr + 255);
  usnprintf(num, 30, "%d, %d, %d", a, b, c);
  RIT128x96x4StringDraw(num, 3, 50, 15);
#endif

        vTaskResume(ekgProcessHandle);
```

```
}

void ADC0IntHandler() {
  unsigned long value;
  static int i = 0;
      //debugPin47();
  // Read the value from the ADC.

  while (1 != ADCSequenceDataGet(ADC0_BASE, EKG_SEQ, &value));
  (data.ekgRawDataAddr)[i++] = (signed int) value;

  // Done sampling if we've taken enough samples
  if (i >= NUM_EKG_SAMPLES) {
    i = 0;
    ekgComplete = true;
  }
ADCIntClear(ADC0_BASE, EKG_SEQ);
}
```

../code/ekgCapture.h

```
/*
 * EkgCapture.h
 * Author: Patrick Ma
 * 2/22/2014
 *
 * Defines the public interface for the EKG data capture task.
 *
 */

#include "tcb.h"

#define EKG_TIMER_BASE TIMER0_BASE  // the timer base used for sample collection
#define EKG_TIMER TIMER_B           // the timer used for EKG sample collection
#define EKG_CFG TIMER_CFG_B_PERIODIC
#define EKG_TIMER_PERIPH SYSCTL_PERIPH_TIMER0

#define EKG_SEQ 0 // The sequence number assigned to the ekg sensor
#define EKG_CH ADC_CTL_CH0  // EKG analog input channel (ch0: pinE7, others: pinE4-6?)
#define EKG_PRIORITY 1  // the ekg sequence capture priority

/* Points to the TCB for the task */
extern TCB ekgCaptureTask;
```

D.7.10 EKG Process

../code/ekgProcess.c

```
/*
```

```c
 * ekgProcess.c
 * Author: Patrick Ma
 * 2/23/2014
 *
 * Processes the raw ekg samples and produces a frequency value
 *
 */

#define DEBUG_PROC 0

#include "FreeRTOS.h"
#include "task.h"

#include "globals.h"
#include "ekgProcess.h"
#include "CircularBuffer.h"
#include "optfft.h"
#include <stdio.h>
#include <string.h>

// Used for debug display
#if DEBUG_PROC
#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
static char num[30];
#endif

static tBoolean firstRun = true;

// ekgProcess data structure. Internal to this task
typedef struct egkProcessData {
  signed int *ekgRawData;
  signed int *ekgImgData;
  CircularBuffer *ekgFreqResult;
} EKGProcessData;

static EKGProcessData data; // internal data object

void ekgProcessRunFunction(void *ekgProcessData);

TCB ekgProcessTask = {&ekgProcessRunFunction, &data}; // task interface

void initializeEKGProcess() {
  data.ekgRawData = (global.ekgRaw);
  data.ekgImgData = (global.ekgTemp);
  data.ekgFreqResult = &(global.ekgFrequencyResult);

#if DEBUG_PROC
  RIT128x96x4Init(1000000);
  RIT128x96x4StringDraw("* EKGProcess Debug *", 0, 0, 15);
#endif
}

```

```

/*
 * Reads in the EKG samples and performs a fast Fourier transform (FFT) on
    the
 * data to extract the primary frequency of the signal
 */
void ekgProcessRunFunction(void *ekgData) {
  EKGProcessData *eData = (EKGProcessData *) ekgData;
  if (firstRun) {
    firstRun = false;
    initializeEKGProcess();
  }

  // need to bit shift >> 4 (divide 16) then subtract 32
  int i = 0;
  int t = 0;   // debug
  for (i = 0; i < NUM_EKG_SAMPLES; i++) {
#if DEBUG_PROC
    usnprintf(num, 30, "%d \n", eData->ekgRawData[i]);
    RIT128x96x4StringDraw(num, 0, 10, 15);
#endif
    int d = ((eData->ekgRawData)[i] >> 4) - 31;
    eData->ekgRawData[i] = d;
#if DEBUG_PROC
    if (eData->ekgRawData[i] > eData->ekgRawData[t])
      t = i;
    usnprintf(num, 30, "%d : %d \n", eData->ekgRawData[i], eData->ekgRawData
   [t]);
    RIT128x96x4StringDraw(num, 0, 20, 15);
#endif
  }

                    // reset Imaginary array
  memset(eData->ekgImgData, 0, sizeof(signed int) * NUM_EKG_SAMPLES);
//  memset(eData->ekgRawData, 0, sizeof(signed int) * NUM_EKG_SAMPLES);

  signed int max_index = optfft( eData->ekgRawData, eData->ekgImgData );
  //post processing
  int freq = ((9166) * max_index) / NUM_EKG_SAMPLES;

#if DEBUG_PROC
  usnprintf(num, 30, "%d : %d  ", max_index, freq);
  RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
  cbAdd(eData->ekgFreqResult, &freq);

         vTaskSuspend(NULL);
}
```

../code/ekgProcess.h

```
/*
 * ekgProcess.h
```

```
 * Author: Patrick Ma
 * 2/23/2014
 *
 * Header and public interface of the ekgProcess task
 *
 */

#include "tcb.h"

/* Points to the TCB for the task */

extern TCB ekgProcessTask;
```

D.7.11 Command Task

../code/commandTask.c

```
/*
 * commandTask.c
 * Author(s); Patrick Ma
 *
 * 3/10/2014
 *
 * Interprets the text commands from the remote connection or system,
   performs
 * the actions requested, and sends a reply signal.
 */

#define DEBUG_COMMAND 0

#include "FreeRTOS.h"
#include "task.h"
#include "globals.h"
#include "warning.h"
#include "CircularBuffer.h"
#include "commandTask.h"
#include <string.h>
#include "hw_ints.h"
#include "driverlib/interrupt.h"
#include "driverlib/adc.h"


#include "drivers/rit128x96x4.h"
#include "utils/ustdlib.h"
char num[30];


#define TOKEN_DELIM " \t" // token delimiter values
#define TEMP_BUFFER_LEN 55

// compiler prototypes
void commandRunFunction(void *commandDataPtr);
```

```

// internal command data structure
typedef struct commandData
{
  char *commandStr;
  char *responseStr;
  unsigned short *measureSelect;
  tBoolean *measureComplete;
  tBoolean *responseReady;
  CircularBuffer *temperature;
  CircularBuffer *systPress;
  CircularBuffer *diasPress;
  CircularBuffer *pulse;
  CircularBuffer *ekg;
  unsigned short *battery;
} CommandData;

static CommandData data; // version of data exposed to outside
TCB commandTask = {&commandRunFunction, &data}; // set up task interface
extern xTaskHandle measureHandle;
extern xTaskHandle displayHandle;
extern xTaskHandle ekgCaptureHandle;

/*
 * local private variables
 */
static tBoolean initialized = false;
static tBoolean measureOn;
static tBoolean displayOn = true;
static int value;
static char *cmd;
static char *sensor;

// communication arrays
static char parseArr[COMMAND_LENGTH];
static char temporaryBuffer[TEMP_BUFFER_LEN];
static char roughString[TEMP_BUFFER_LEN];
static char *formattedStr = temporaryBuffer;


/*
 * initializes task variables
 */
void initializeCommandTask(){
  data.commandStr = (global.commandStr);
  data.responseStr = (global.responseStr);
  data.measureSelect = &(global.measurementSelection);
  data.measureComplete = &(global.measurementComplete);
  data.responseReady = &(global.responseReady);

  data.temperature = &(global.temperatureCorrected);
  data.systPress = &(global.systolicPressCorrected);
  data.diasPress = &(global.diastolicPressCorrected);
```

```c
  data.pulse = &(global.pulseRateCorrected);
  data.ekg = &(global.ekgFrequencyResult);
  data.battery = &(global.batteryState);
}

/*
 *  parses the command string for command parameters, writing those values to
 *  variables
 */
void parse(CommandData *cData) {
  char delim[2] = TOKEN_DELIM;

  strncpy(parseArr, cData->commandStr, COMMAND_LENGTH - 1);

  cmd = strtok(parseArr, delim);
  sensor = strtok(NULL, delim);
}

/*
 * Adds an acknowledge or not acknowledge to the response buffer
 */
void ackNack(CommandData *cData, tBoolean stat) {
  if (stat)
    strncat(cData->responseStr, "<p>A</p>", RESPONSE_LENGTH - 1);
  else
    strncat(cData->responseStr, "<p>E</p>", RESPONSE_LENGTH - 1);
}

/*
 * Formats given string with appropriate html tags. Returns pointer to
 * formmated string. If statusOK is false, <blink> or </blink> flags are added.
 *
 * CAUTION! addTags does not guard against buffer overflow. Make sure your
 * string is not too long. The tags add up to 22 characters.
 */
char* addTags(char* string, tBoolean statusOK) {
  memset(temporaryBuffer, '\0', TEMP_BUFFER_LEN - 1);
  strncpy(temporaryBuffer, "<p>", 3); // first tag

  if (!statusOK) {
    strcat(temporaryBuffer, "<blink>"); // include warning maybe
  }
  strcat(temporaryBuffer, string);
  if (!statusOK) {
    strcat(temporaryBuffer, "</blink>");
  }
  strcat(temporaryBuffer, "</p>");

  return temporaryBuffer;
}
```

```
/*
 * measures the data from a sensor
 */
void measureFromSensor(CommandData* cData) {
  tBoolean meas = true;
  switch (*sensor) {
    case 'A' :
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("take A", 2, 30, 15);
#endif
      *(cData->measureSelect) = 0;
      break;
    case 'T':
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("take T", 2, 30, 15);
#endif
      *(cData->measureSelect) = 1;
      break;
    case 'B':
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("take B", 2, 30, 15);
#endif
      *(cData->measureSelect) = 2;
      break;
    case 'P' :
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("take P", 2, 30, 15);
#endif
      *(cData->measureSelect) = 3;
      break;
    case 'E' :
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("take E", 2, 30, 15);
#endif
      *(cData->measureSelect) = 4;
      break;
    case 'S' :
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("unsupported", 0, 30, 15);
#endif
      *(cData->measureSelect) = '%';
      break;
    default :
#if DEBUG_COMMAND
      RIT128x96x4StringDraw("invalid command", 0, 30, 15);
#endif
      meas = false;
  }

  if (meas) {
    ackNack(cData, true);
#if DEBUG_COMMAND
    RIT128x96x4StringDraw("MeasureTask go!", 0, 40, 15);
```

```
#endif
  } else {
#if DEBUG_COMMAND
    RIT128x96x4StringDraw("no Measure", 0, 40, 15);
#endif
    ackNack(cData, false);
  }

//  while (!(cData->measureComplete)) { // wait until measurement is
    finished
//  }
}

/*
 * gets & formats string
 */
void printTemp(CommandData *cData, tBoolean statusOK) {
  value = (int) *(float *)cbGet(cData->temperature);
  usnprintf(roughString, TEMP_BUFFER_LEN, "Temperature: %d C", value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->
    responseStr) - 1);
}

/*
 * gets & formats string
 */
void printPressure(CommandData *cData, tBoolean statusOK) {
  value = (int) *(float *)cbGet(cData->systPress);
  usnprintf(roughString, TEMP_BUFFER_LEN, "Systolic Pressure: %d mmHg",
    value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->
    responseStr) - 1);
  value = (int) *(float *)cbGet(cData->diasPress);
  usnprintf(roughString, TEMP_BUFFER_LEN, "Diastolic Pressure: %d mmHg",
    value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->
    responseStr) - 1);
}

/*
 * gets & formats string
 */
void printPulse(CommandData *cData, tBoolean statusOK) {
  value = (int) *(float *)cbGet(cData->pulse);
  usnprintf(roughString, TEMP_BUFFER_LEN, "Pulse Rate: %d BPM", value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->
    responseStr) - 1);
}

```

```
/*
 * gets & formats string
 */
void printEKG(CommandData *cData, tBoolean statusOK) {
  value = *(int *)cbGet(cData->ekg);
  usnprintf(roughString, TEMP_BUFFER_LEN, "EKG Frequency: %d Hz", value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->responseStr) - 1);
}

/*
 * gets & formats string
 */
void printBattery(CommandData *cData, tBoolean statusOK) {
  value = (int) *(cData->battery) / 2;
  usnprintf(roughString, TEMP_BUFFER_LEN, "Battery Remaining: %d %%", value);
  addTags(roughString, statusOK);
  strncat(cData->responseStr, formattedStr, RESPONSE_LENGTH - strlen(cData->responseStr) - 1);
}

/*
 * Format the outgoing response string based on the specified measurement
 */
void formatResponseStr(CommandData *cData){

        int temp = *(int *)cbGet(cData->temperature);
        int sysPress = *(int *)cbGet(cData->systPress);
        int diaPress = *(int *)cbGet(cData->diasPress);
        int pulse = *(int *)cbGet(cData->pulse);

    tBoolean bpWarn = false;
    tBoolean tempWarn = false;
    tBoolean pulseWarn = false;
    tBoolean battWarn = false;

    if ( sysPress > SYS_MAX*ALARM_HIGH || diaPress > DIA_MAX*ALARM_HIGH )
      bpWarn = true;
    if ( temp < TEMP_MIN*WARN_LOW || temp > TEMP_MAX*WARN_HIGH )
      tempWarn = true;
    if ( pulse < PULSE_MIN*WARN_LOW || pulse > PULSE_MAX*WARN_HIGH )
      pulseWarn = true;
    if (*(cData->battery) < BATTERY_MIN)
      battWarn = true;

  switch (*sensor) {
    case 'A' : //take all measurements
      printTemp(cData, !tempWarn);
      printPressure(cData, !bpWarn);
      printPulse(cData, !pulseWarn);
      printEKG(cData, true);
```

```
      printBattery(cData, !battWarn);
      break;
    case 'T' : // get temperature measurementk
      printTemp(cData, tempWarn);
      break;
    case 'B' : // get syst
      printPressure(cData, bpWarn);
      break;
    case 'P' : //pulse rate
      printPulse(cData, pulseWarn);
      break;
    case 'E' : //ekg frequency
      printEKG(cData, true);
      break;
    case 'S' : // battery state
      printBattery(cData, battWarn);
      break;
  }
}

/*
 * runs the command task
 *
 * NB: the response string needs html formatting:
 * ex: <p> Temperature: 50 C </p> <p> <blink> Blood Pressure: 120 </blink>
    </p>
 */
void commandRunFunction(void *commandDataPtr) {
    CommandData *cData = (CommandData *) commandDataPtr;
        *(cData->responseReady) = false;
  if (!initialized) {
    initialized = true;
    initializeCommandTask();
  }

#if DEBUG_COMMAND
  usnprintf(num, 30, "Initialize cmd function");
  RIT128x96x4StringDraw(num, 0, 10, 15);
#endif

  parse(cData);

#if DEBUG_COMMAND
  usnprintf(num, 30, "%s %s", cmd, sensor);
  RIT128x96x4StringDraw(num, 0, 20, 15);
#endif

  memset(cData->responseStr, '\0', RESPONSE_LENGTH);
  switch(*cmd) {
    case 'D' : // toggle display on/off
      if (displayOn) {
        vTaskSuspend(displayHandle);
        RIT128x96x4Clear();
```

```
      } else {
        vTaskResume(displayHandle);
      }
      displayOn = !displayOn;
      ackNack(cData, true);

#if DEBUG_COMMAND
      usnprintf(num, 30, "%d %s", displayOn, cData->responseStr);
      RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
      break;
    case 'S' :  // start measurements
            vTaskResume(measureHandle);
            vTaskResume(ekgCaptureHandle);

      // enable the interrupts used for measurement
      //      IntEnable(INT_GPIOA); // for pulse
      ADCSequenceEnable(ADC0_BASE, 0);//      IntEnable(INT_ADC0SS0); // for
    ekg
      //      IntEnable(INT_ADC0SS1); // for temperature
      measureOn = true;
      ackNack(cData, true);
      break;
    case 'P' :  // stop
            vTaskSuspend(measureHandle);
            vTaskSuspend(ekgCaptureHandle);

      // disable the interrupts used for measurement
      //      IntDisable(INT_GPIOA);  // for pulse
      ADCSequenceDisable(ADC0_BASE, 0);//      IntDisable(INT_ADC0SS0);  //
    for ekg
      //      IntDisable(INT_ADC0SS1);  // for temperature
      measureOn = false;
      ackNack(cData, true);
      break;
    case 'M' : // measure a sensor
      measureFromSensor(cData);
      formatResponseStr(cData);
      break;
    case 'G' :  // Commands for DEBUG mode
      switch (*sensor) {
        case 'D' :
          ackNack(cData, true);
          if (displayOn)
            strncat(cData->responseStr, "<p>On</p>", RESPONSE_LENGTH - 1);
          else
            strncat(cData->responseStr, "<p>Off</p>", RESPONSE_LENGTH - 1);

#if DEBUG_COMMAND
      usnprintf(num, 30, "%d %s", displayOn, cData->responseStr);
      RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
          break;
```

```
        case 'M' :
          ackNack(cData, true);
          addTags((measureOn ? "ON" : "OFF"), false); //TODO fix the false vlaue
          strncat(cData->responseStr, temporaryBuffer, RESPONSE_LENGTH - 1);

#if DEBUG_COMMAND
          usnprintf(num, 30, "%d %s", measureOn, cData->responseStr);
          RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
          break;
      }
      ackNack(cData, false);
      break;
    default : // send error to remoteStr
      ackNack(cData, false);
#if DEBUG_COMMAND
      usnprintf(num, 30, "invalid command");
      RIT128x96x4StringDraw(num, 0, 30, 15);
#endif
  }
  *(cData->responseReady) = true;
  vTaskSuspend(NULL);
}
```

../code/commandTask.h

```
/*
 * commandTask.h
 * Author(s): Patrick Ma
 *
 * 3/10/2014
 *
 * Header for the command task
 *
 * Command supported by the commandTask are of the form:
 * <X> <Y> <Z>
 *
 * These placeholders are explained below. If an invalid command sequence is
 * given, commandTask responds with an 'E' character. Otherwise, the immediate
 * response is 'A', followed by any specified return value(s).
 *

 * <X> : Primary Command affecting major system functions
 *
 *    S - Indicates START mode. Causes the hardware to initiate the
 *    previously assigned measurement task.
 *
 *    P - Indicates STOP mode. Causes the hardware to suspend any current
 *    measurements.
 *
 *    D - Toggles the local OLED display on and off
```

```
 *
 *    M - Initiate a MEASURE function. Takes additional commands to specify the
 *    type and number of measurements
 *
 *    G - Causes the commandTask to enter DEBUG mode. Additional modifiers
 *    specify which state to debug.
 *
 *
 * <Y> : Modifier specifying the sublevel of behavior
 *
 *    A - Instruct the system to measure All sensors
 *
 *    T - Instruct system to measure Temperature. Will return the temperature
 *    in degrees Celsius
 *
 *    B - Instruct system to measure the patient Blood pressure. Will return
 *    the systolic and diastolic blood pressure in mmHg
 *
 *    P - Instruct system to measure patient Pulse rate. Will return the heart
 *    rate in beats per minute
 *
 *    E - Instruct the system to perform an EKG measurement. Will return the
 *    primary EKG frequency
 *
 *    S - Query the system battery Status. Returns the percentage left.
 *
 *    D - DEBUG mode only. Returns the local Display status
 *
 *    M - DEBUG mode only. Returns system Measurement status. Returns boolean
 *    true if measurement is enabled, false otherwise
 *
 *    T - DEBUG mode only. Returns the Type of the last measurement performed.
 *
 *
 * <Z> : Modifier specifying the number of measurements to perform. The
 *       measurements are sent to a buffer for usage one at at time.
 *
 *    <no value> - Scan continuously
 *
 *    <integer value> - perform the specified number of measurements.
 */

#include "tcb.h"

/* points to the TCB for commandTask */
extern TCB commandTask;
```

D.7.12 Status

../code/status.h

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

#include "tcb.h"  // for TCBs

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

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

../code/status.c

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

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

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

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

static StatusData data;  // the internal data
TCB statusTask = {&statusRunFunction, &data}; // task interface

/* Initialize the StatusData task values */
void initializeStatusTask() {
  // Load data
  data.batteryState = &(global.batteryState);

  // Load TCB
  statusTask.runTaskFunction = &statusRunFunction;
```

```
  statusTask.taskDataPtr = &data;
}

/* Perform status tasks */
void statusRunFunction(void *data){
  static tBoolean onFirstRun = true;

  if (onFirstRun) {
    initializeStatusTask();
    onFirstRun = false;
  }

  if (IS_MAJOR_CYCLE) {
    StatusData *sData = (StatusData *) data;
      if (*(sData->batteryState) > 0)
        *(sData->batteryState) = *(sData->batteryState) - 1;
  }
}
```

D.8 Website Source

../code/fs/index.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>
<meta http-equiv="content-type" content="text/html;charset=ISO-8869-1"><title>Doctor at Your Fingertips - Web Interface</title>

<script language="JavaScript">
<!--
function send_command()
{
  var send = false;
  var command = document.getElementById("terminal");

  // Create a send request (don't bother with ActiveX Objects)
  send = new XMLHttpRequest();

  if(send)
  {
    // User entered non-blank command
    if(command.value != "")
    {
      var res = command.value.split(" ");
      var uri = encodeURI("/cgi-bin/send_command/value=" + res.join("+"));
      // Send request to special url for server processing
      send.open("GET", uri, true);
      send.send(null);
      setTimeout(receive_command, 500);
    }
  }
```

```
}

function receive_command() {
  var receive = false;

  receive = new XMLHttpRequest();

  // Send a request for the current command response
  if(receive)
  {
    receive.open("GET", "/cgi-bin/receive_command", true);

    // Register receive_complete() with this XMLHTTPRequest so it is called
     when a response is received
    receive.onreadystatechange = receive_complete;
    receive.send(null);
  }

  function receive_complete()
  {
    if(receive.readyState == 4)
    {
      if(receive.status == 200)
      {
        document.getElementById("response").innerHTML = receive.responseText
    ;
      }
    }
  }
}

function blink() {
  var blinks = document.getElementsByTagName('blink');
  for (var i = blinks.length - 1; i >= 0; i--) {
    var s = blinks[i];
    s.style.visibility = (s.style.visibility === 'visible') ? 'hidden' : '
    visible';
  }
  window.setTimeout(blink, 1000);
}
if (document.addEventListener) document.addEventListener("DOMContentLoaded",
     blink, false);
else if (window.addEventListener) window.addEventListener("load", blink,
    false);
else if (window.attachEvent) window.attachEvent("onload", blink);
else window.onload = blink;
//-->
</script>
<style type="text/css">
body
{
  font-family: Arial;
  background-color: white;
```

```
  margin: 20px;
  padding: 0px
}
h1
{
  color: #000000;
  font-family: Arial;
  font-size: 24pt;
}

</style>
</head>
<body>

<div id="container" style="float:right; right:50%; position:relative">
  <div id="uicontainer" style="float:right; right:-50%; position:relative">
    <div id="title"> <h1> Doctor at Your Fingertips </h1> </div>
    <div id="ui" style="padding:10px;float:left;border: 1px solid;">
      <div id="enter"> Enter a command: </div>
      <div id="command" style="margin-bottom:10px">
        <input id="terminal" type="text" onkeydown="if (event.keyCode == 13) send_command()"/>
        <input id="send" value="Send" type="button" onclick="send_command()" />
      </div>
      <div id="response"> </div>
    </div>

    <div id="validcommands" style="margin-left:20;float:left;border:1px solid;">
      <table>
        <tr>
          <td> <b> Command </b> </td>
          <td> <b> Effect/Usage </b> </td>
        </tr>
        <tr>
          <td> S </td>
          <td> Starts the device.  Enables measurement devices and interrupts. </td>
        </tr>
        <tr>
          <td> P </td>
          <td> Stops the device.  Disables measurement devices and interrupts. </td>
        </tr>
        <tr>
          <td> D </td>
          <td> Enables/disables the onboard OLED. </td>
        </tr>
        <tr>
          <td> M &lt;sensor(s)&gt; </td>
          <td> Measure the specified sensor(s). Sensors are specified by... </td>
```

```
            </tr>
          </table>
        </div>
    </div>
</div>

</div>

</body>
</html>
```