

EE 474 Project 5

Irina Golub, Yifan Shao, Eric Ho

TABLE OF CONTENTS

Abstract	1
Introduction	2
Project Details	2
Design Specification	2
Software Implementation	8
Results	24
Error Analysis	27
Resolved Errors	27
Unresolved Errors	28
Test Plan	28
Test Specification	29
Test Cases	33
Summary	39
Conclusion	39
Group Member Contributions	40
Appendix A	41

ABSTRACT

The purpose of this lab is to develop a prototype for a portable medical device capable of taking vital bodily measurements on the go, processing the data, and alerting the user of abnormal measurements. For the prototype, measurements are modeled by external, controllable signals to demonstrate its functionality as well as the capability to handle abnormal situations. The prototype is also split into two modules, one responsible for connecting to additional peripheral devices in order to take measurements, and one responsible for processing data, displaying information, and overall control of the system.

The prototype underwent a number of considerations for its intended users, use cases, and how those use cases should be decomposed into functions, distributed across the two modules, and implemented. As the implementation details have changed over time throughout the prototype's

development as the requirements were expanded upon, the overall goal and purpose of the system has remained consistent.

The team ran into various problems and obstacles during the project, a number of which was unforeseen and required various components to be revised and decisions to be revisited. By the end, however, the team was able to find solutions to the issues that have come up and successfully build a prototype of a system which fulfills the original intent.

INTRODUCTION

The prospective medical device being developed will perform a number of essential measurements for people with chronic medical problems. The product will use the Arduino ATmega 2560 microcomputer for the System Control, and the Arduino Uno microcomputer for the Peripherals with an intersystem communication link between them. The system will also send measurement data to and accept commands from a remote computer running the PuTTY software. Measurements will be taken from various sensors.

A rapid prototyping approach will be used to initialize the basic, high-level architecture of the system. Instead of reading actual input values, this project will model the input signals and focus on the flow of control and communication between the software modules. Phase III of the development of this device includes the addition of respiration rate and pulse rate measurements using ISRs, blood pressure cuff control using external hardware, and extended display capabilities. Phase IV includes the addition of a remote communications system and network interface, and EKG measurement capabilities.

First, various UML diagrams will be made including a block diagram, use-case diagram, functional decomposition diagram, task/class diagrams, and activity diagrams of each task. The diagrams will help identify the static and dynamic relationships among the tasks. The system is designed in the Arduino IDE based upon the UML diagrams. The resulting software and hardware represents the Phase IV prototype of the medical device.

PROJECT DETAILS

Design Specification

The medical device prototype will be implemented using an ATmega microcomputer for the high-level system management, control, and display, and an UNO microcomputer for the measurement, alarm and other peripheral functions. The initial prototypes implementing various measurements, intersystem communication, and a user-interface for selecting measurements and viewing data has been implemented. The Phase IV prototype is a further development of the medical system including interrupt-based signal inputs, further development of the user display, hard real-time EKG signal measurement and FFT, respiration rate measurement, bidirectional communication with an external computer, a LAN interface, and ability to accept and interpret commands from a remote system.

The device inputs include blood pressure, body temperature, respiration rate, pulse rate, and EKG signals, which are being simulated in this Phase. The user will also be able to switch

between display modes and input a measurement selection and alarm acknowledgement. The system outputs include display of blood pressure, body temperature, pulse rate, respiration rate, EKG status, and device battery status, as well as visual annunciation of warnings and alarms. The keypad task will include a side effect that changes a global 'display mode' variable as well as control which measurements are taken and displayed.

First, UML diagrams should be updated for the Phase IV prototype to accurately express the static and dynamic structure of the Phase IV system:

- A use-case diagram.
- A functional decomposition diagram for the System Control block.
- A task/class diagram for the System Control block.
- Activity diagrams for the System Control block.
- A functional decomposition diagram for the Peripheral Subsystem.
- A task/class diagram for the Peripheral Subsystem.
- Activity diagrams for the Peripheral Subsystem.

The *System Control* portion of the medical monitoring and analysis system is to be designed as a collection of tasks that execute continuously, on a periodic schedule, following power on. The system tasks will be scheduled in a dynamic task queue and implemented as a doubly linked list of TCBs. Information within the system will be exchanged using a number of shared variables.

The required variables and their initial values are updated for the Phase IV prototype and included in Appendix A. Each task will be expressed a task Control Block (TCB) structure.

The TCB is implemented as a C struct. The TCB data pointer will point to a struct containing references to all data utilized by the task. The structs are updated for the Phase II prototype and included in Appendix B. Each TCB will have two members: a pointer to a function taking a void* argument and returning a void, a void* pointer referencing the data for the task, and pointers to the next and previous TCBs in a linked list data structure.

The *Peripheral Subsystem* of the system is to be designed as a collection of peripheral drivers that execute on demand.

The *Intersystem Communication* link is a bidirectional net that transports commands from *System Control* tasks to a designated device in the *Peripheral Subsystem* and returns a response to the command.

The *Remote Communications* link is a bi-directional net designed to provide remote command and control access to a medical monitoring system.

The task queue is implemented as an array of twelve elements that are pointers to variables of type TCB. Eleven of the elements include pointers to the TCBs for *Measure*, *Compute*, *Keypad*, *Display*, *Warning-Alarm*, *Status*, *Communications*, *EKG Capture*, *EKG Processing*, *Remote Communications*, and *Command*. The function of each task required for the system is specified as follows:

1. *Startup*: This task is not included in the task queue and is to run one time each time the system is started. The task performs any necessary system initialization and configures and activates the hardware-based system time base. A sequence diagram should be made for this task to show the flow of control algorithm for the system.
2. *Schedule*: This task is not included in the task queue. This task manages the execution order of the tasks in the system. This task causes the suspension of all task activity except

the warning and error annunciation for five seconds. The global counter will be incremented based on the local hardware-based system time base. The system cannot block for five seconds. This task will examine all *addTask* type flags and add or remove all flagged tasks to or from the task queue. A state chart should be created for this task.

3. *Measure*: Accepts a pointer to the struct containing all the data required for this task. Only the measurements selected by the user are to be performed. For each measurement task, a Request message, specifying the desired measurement and any relevant data, must be created and sent to the *Peripheral Subsystem*. The measurement subtasks are called in the *Peripheral Subsystem* based on the user's measurement selection. The various parameters are simulated in the *Peripheral Subsystem* as follows:

- TemperatureRaw: The patients temperature is modeled using an analog signal that is changed between 0 and 5 V using a potentiometer. The signal is read and scaled appropriately to match the human temperature range. The temperature data is stored in a circular eight reading buffer if the current reading is more than 15% different from the previously stored reading.
- SystolicPressRaw and diastolicPressRaw: The cuff is inflated and deflated manually using a pushbutton and switch. The button increments the pressure by 10% for each press when the switch is in increment position and decrements it by 10% for each press when the switch is in decrement position. The cuff and patient are modeled by a simple counter. Each button will require a parallel capacitor for button debouncing. At a blood pressure in the range of 110 to 150 mm Hg, the Uno will generate an interrupt that will shine an LED to show that the systolic measurement is to be captured. At a blood pressure in the range of 50 to 80 mm Hg, the Uno will generate an interrupt that will shine an LED to show that the diastolic measurement is to be captured.
- pulseRateRaw: The output of a pulse-rate transducer will be modeled using a function generator that sends a 0 to 3.3 V pulse wave to the Uno. The pulse rate pin will be assigned an interrupt with a corresponding ISR that increments a pulse Counter. The measured pulse rate value will be determined and stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading. The upper frequency limit of the incoming signal will be determined.
- respirationRateRaw: The output of a respiration-rate transducer will be modeled using a function generator that sends a 0 to 3.3 V square wave to the Uno. The respiration rate pin will be assigned an interrupt with a corresponding ISR that increments a respiration Counter. The measured respiration rate value will be determined and stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading. The upper frequency limit of the incoming signal will be determined.

The *Peripheral Subsystem* performs the requested operation and returns the results to the *System Control* subsystem. When a *response* message is received from the *Peripheral Subsystem*, the task will enter the value of the measured data into the

designated *MeasureData* buffer. When the *measure* task has completed a new set of measurements, the *addTask* flag for the *Compute* task is to be set.

4. *EKG Capture*: Accepts a pointer to the struct containing all the data required for this task. This task is scheduled on demand. This task will read a time-varying sinusoidal analog data signal from one transducer and isochronously collect 256 samples, convert the sampled values to digital form, then store the converted samples in a 256-measurement buffer for further processing. The samples must be read at a sampling rate of two and a half to three times the maximum specified frequency. The driver will be written at the register level. When a capture is complete, the task must signal the *EKG Processing* task.
5. *EKG Processing*: Accepts a pointer to the struct containing all the data required for this task. This task will perform an FFT on the EKG samples collected by the *EKG Capture* task using an integer algorithm developed by Brent Plump and an FFT algorithm provided by *Bwang Software, Ltd.*. The EKG data will range from 35 Hz to 3.75 kHz with a peak amplitude of 3.3 V.
6. *Compute*: Accepts a pointer to the struct containing all data required for this task. This task is only scheduled if new data is available from the *Measure* task. When scheduled, this task will take the measured data, perform the necessary transformations or corrections, enter the result into the appropriate buffer, reset the *addTask* flag for the *Compute* task, and set the *removeTask* flag for the *Compute* task. The transformations to the measured data are:
 - Corrected temperature = $5 + 0.75 * \text{raw temperature}$
 - Corrected systolic pressure = $9 + 2 * \text{raw systolic pressure}$
 - Corrected diastolic pressure = $6 + 1.5 * \text{raw diastolic pressure}$
 - Corrected pulse rate = $8 + 3 * \text{raw pulse rate}$
 - Corrected respiration rate = $7 + 3 * \text{raw respiration rate}$
7. *TFT Keypad*: Accepts a pointer to the struct containing all data required for this task. The keypad should allow the user to select between *Menu*, *Display*, and *Annunciation* mode. In 'menu', the user will be able to select a temperature, blood pressure, pulse rate, respiration rate, EKG measurement or any combination of these. In 'display' mode, the user will be able to exit to the main menu or enter annunciation mode. In 'annunciation' mode, the user will be able to acknowledge an alarm or warning. The keypad is scanned for new key presses on a quarter-second cycle.
8. *Display*: Accepts a pointer to the struct containing all data required for this task. This task has four display states: *Main Menu*, *Selection Menu*, *Display*, and *Annunciate*. In *Main Menu*, this task will display "Main menu". In the *Selection Menu*, this task will display the different measurement options and what keypad number they correspond to. In the *Display* state, retrieves the results from the *Compute* task and formats the data to be displayed on the TFT. In *Annunciation* state, the measurements will also be displayed along with battery status and warning/alarm annunciation. The default value for all measurements in *Annunciation* shall be green. The warning values will be yellow, and the alarm values will be red.

9. *Warning-Alarm*: Accepts a pointer to the struct containing the data required for this task. This task will compare the measured data with a normal range for the measurements as specified:

- Temperature: 36.1 C to 37.8 C
- Systolic Pressure: <120 mm Hg
- Diastolic pressure: <80 mm Hg
- Pulse rate: 60 to 100 beats per minute
- Respiration rate: 12 to 25 breaths per minute
- EKG: 35 Hz to 3.75 kHz
- Battery: Greater than 20% charge remaining

The values will be displayed green if the measurement/battery state is within the specified range. A *warning* value will be displayed *yellow* and flash with the specified period if any measurement is more than 5% out of range:

- Pulse rate: flash with 2-second period
- Temperature: Flash with 1-second period
- Blood Pressure: Flash with 0.5 second period

An *alarm* value is displayed *red* under the following conditions:

- If the systolic blood pressure measurement is more than 20 percent above the specified limit
- If the temperature, pulse rate, or respiration rate measurement is more than 15 percent above or below the specified limit.
- If the *acknowledge* key associated with an annunciation is pressed, the alarm shall change to a warning. However, if the signal remains out of range for more than five measurements, the alarm annunciation will resume.

The state of the battery annunciation will display *red* when the state of the battery drops below 20% remaining.

10. *Peripheral Communications Functions*: Supports communication between the *System Control* and the *Peripheral Subsystem*. To access and utilize a tool, the task must send a *Request* message to the *Peripheral Subsystem*. A *Request* message contains the following fields:

- Start of message
- End of message
- Requesting task identifier
- Function being requested
- Data required by the function

The subsystem will accept and interpret the incoming message, perform the requested action, and return the results in a *response* message. A *Response* message must contain the following fields:

- Start of message
- End of message
- Requesting task identifier
- Function being requested
- Data being returned by the function

When the *Response* message is received, a flag is set informing the *Scheduler* to schedule the appropriate task(s).

11. *Status*: Accepts a pointer to the struct containing the data for this task. This task decrements the battery state by 1 each time the *Status* task is entered. The value is no longer decremented if the battery state reaches 0.
12. *Remote Communications*: Accepts a pointer to the struct containing the data for this task. This task shall be started following power on, then will:
 - Initialize the network interface
 - Connect to and configure a local area network (LAN)
 - Set up a handler to communicate with a remote system
 - Format the data to be displayed and send the formatted data over the network for display on the remote terminal.
 - Continually update the displayed data at a five-second rate.
13. *Command*: Accepts a pointer to the struct containing the data for this task. This task is scheduled whenever a command has been received by the system or when an outgoing message must be formatted in preparation for transmission to the remote computer. This task is then deleted from the task queue.
 - Receive: When a command has been received by the system, the task must verify that the received message is valid. If valid, it is acted upon; if invalid, an error response must be sent to the Remote Communications task. The legal commands and their interpretation are specified in Appendix B.
 - Transmit: When a message is to be transmitted, the Command task must build the message body. The message body is then sent to the Remote Communications task for transmission.
14. *Remote Display*: The remote display presents the following information:
 - The name of the product
 - The patient's name
 - The doctor's name

The corrected patient data presented at the remote site must be expressed as strings showing the temperature, systolic pressure, diastolic pressure, pulse rate, respiration rate, EKG, and battery. The measured value must flash whenever a warning occurs.

Annotation must also accompany each displayed measurement indicating how many times during an eight-hour interval that a new warning has occurred.

The system inputs and outputs should be shown in a data flow diagram. The execution time of each task should be determined empirically. The tasks shall be executed in an infinite loop by calling *Schedule* in the loop() function of the Arduino IDE.

Software Implementation

A high-level block diagram for the final system is shown in Fig. 1, given in the project description document.

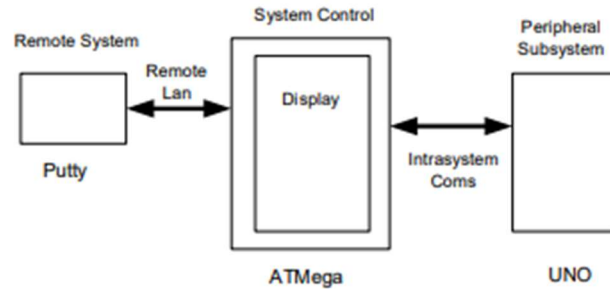


Fig. 1. High-level System Block Diagram

The Phase IV prototype of the system includes using a remote computer to select measurements to be taken and display measurements. The Uno board will still run the *Peripheral Subsystem* and read the measurement input signals while the ATmega board is used for all other functions. A UML use-case diagram for the Phase IV prototype of this medical device is shown in Fig. 2.

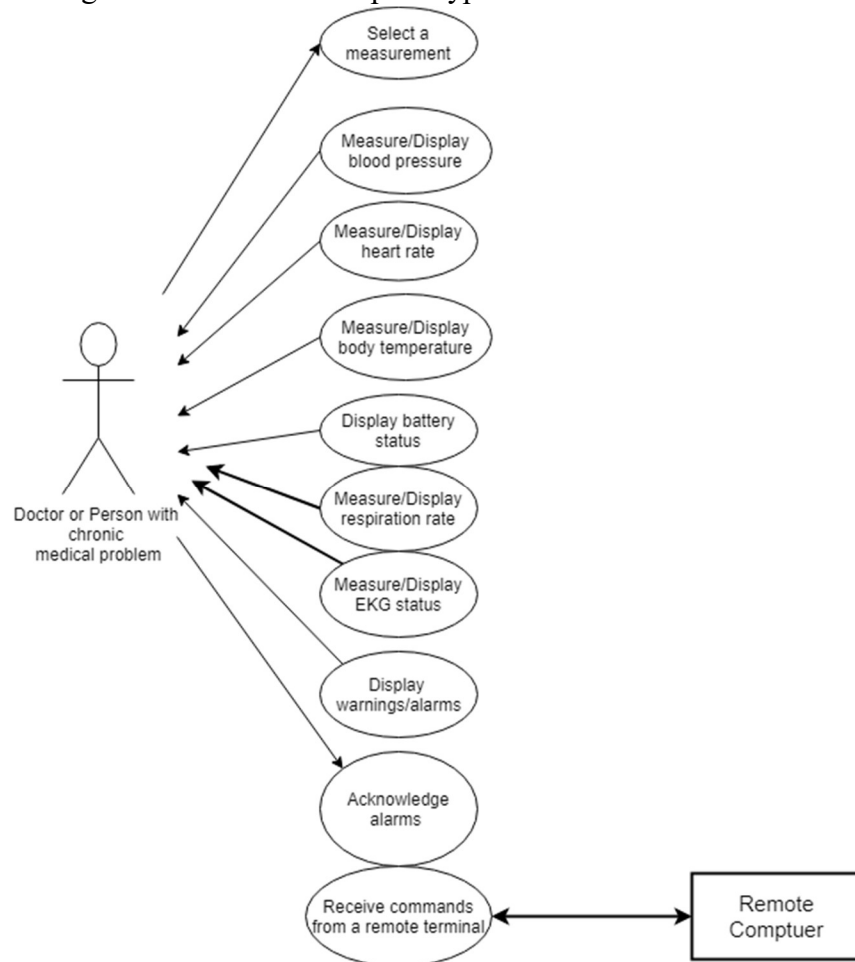


Fig. 2. UML use-case diagram for the Phase II prototype of the medical device.

Use-Case Textual Description:

Select a measurement

User will use a keypad in the menu selection mode to select one or more of three measurements to be taken

- Exceptions – power off
- Display blood pressure
- User has selected to take a blood pressure measurement and is on the display or annunciation mode. TFT display will show "blood pressure: " plus the value of the user's measured blood pressure in green. Most recent blood pressure value will also display on the remote computer if measurement has been requested.
- Exceptions – power off
- Display pulse rate
- User has selected to take a pulse rate measurement and is on the display or annunciation mode. TFT display will show "pulse rate: " plus the value of the user's measured pulse rate in green. Most recent pulse rate value will also display on the remote computer if measurement has been requested.
- Exceptions – power off
- Display body temperature
- User has selected to take a body temperature measurement and is on the display or annunciation mode. TFT display will show "temperature: " plus the value of the user's measured temperature in green. Most recent temperature value will also display on the remote computer if measurement has been requested.
- Exceptions – power off
- Display respiration rate
- User has selected to take a respiration rate measurement and is on display or annunciation mode. TFT display will show "resp rate: " plus the value of the user's measured respiration rate in green. Most recent respiration rate value will also display on the remote computer if measurement has been requested.
- Exceptions – power off
- Display EKG Status
- User has selected to take an EKG measurement and is on display or annunciation mode. The TFT display will show "EKG: " plus the measured EKG frequency range. Most recent EKG status value will also display on the remote computer if measurement has been requested.
- Exceptions – power off
- Display battery status
- User is in annunciation mode, TFT display will show "battery: " plus the value of the battery status of the device in green.
- Exceptions – power off
- Display warnings/alarms
- User is in annunciation mode. When a measured value is out of healthy range, the value will display in the color yellow. When a value is in a dangerous range, the value will show in red. Most recent alarms/warnings will also show on the remote computer if this has been requested.
- Exceptions – power off
- Acknowledge alarms
- User may acknowledge alarm annunciations in 'annunciation' mode. This will remove the alarm unless the value persists in the alarm value range for the next five measurements.
- Exceptions – power off
- Receive commands from a remote terminal
- The system will be able to recognize commands "I" (initialize network comms), "S" (start measurements), "P" (stop measurement tasks), "D" (enable local display), "M" (request most recent measurements), and "W" (request most recent warnings/alarms). The system will send "E" to the remote computer if an incorrect or non-existent command is received.
- Exceptions – power off, no communication available

The *Peripheral Subsystem* contains the following functional blocks: *Measure* and one portion of the *Intersystem Communication*. Based upon the Phase IV System Requirements and use-case diagrams, a functional decomposition diagram for the *Peripheral Subsystem* is shown in Fig. 5.

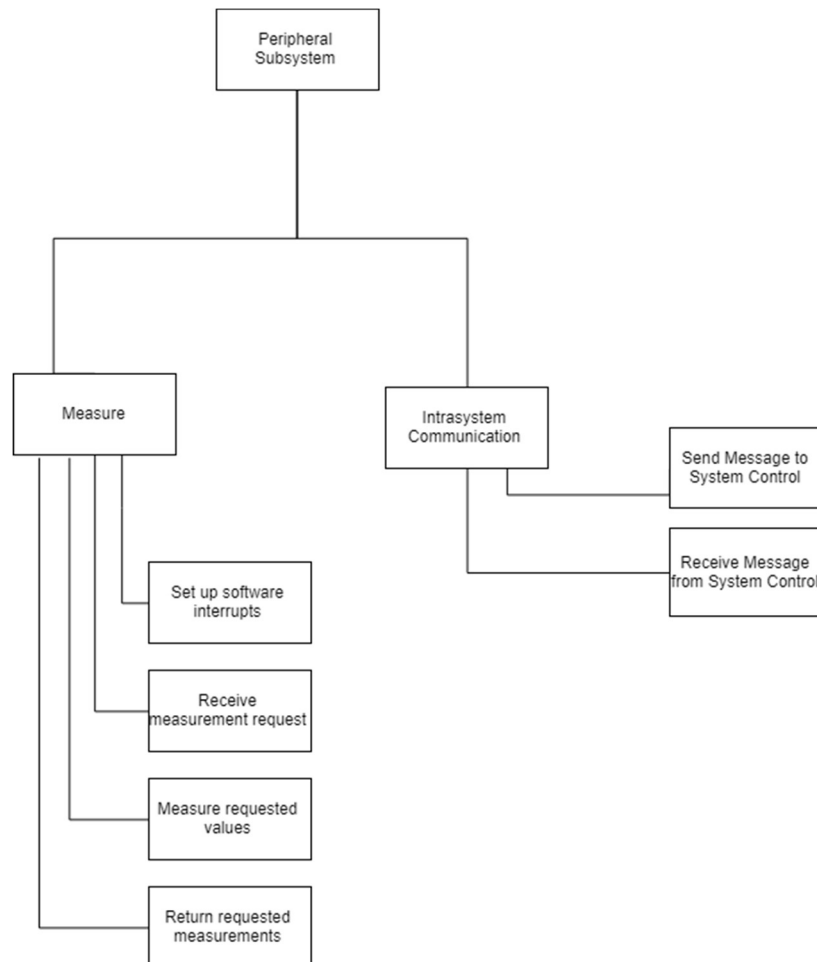


Fig. 5. Functional decomposition diagram for the Peripheral portion of the system.

These functional blocks decompose into the Phase II task diagrams shown in Fig. 6.

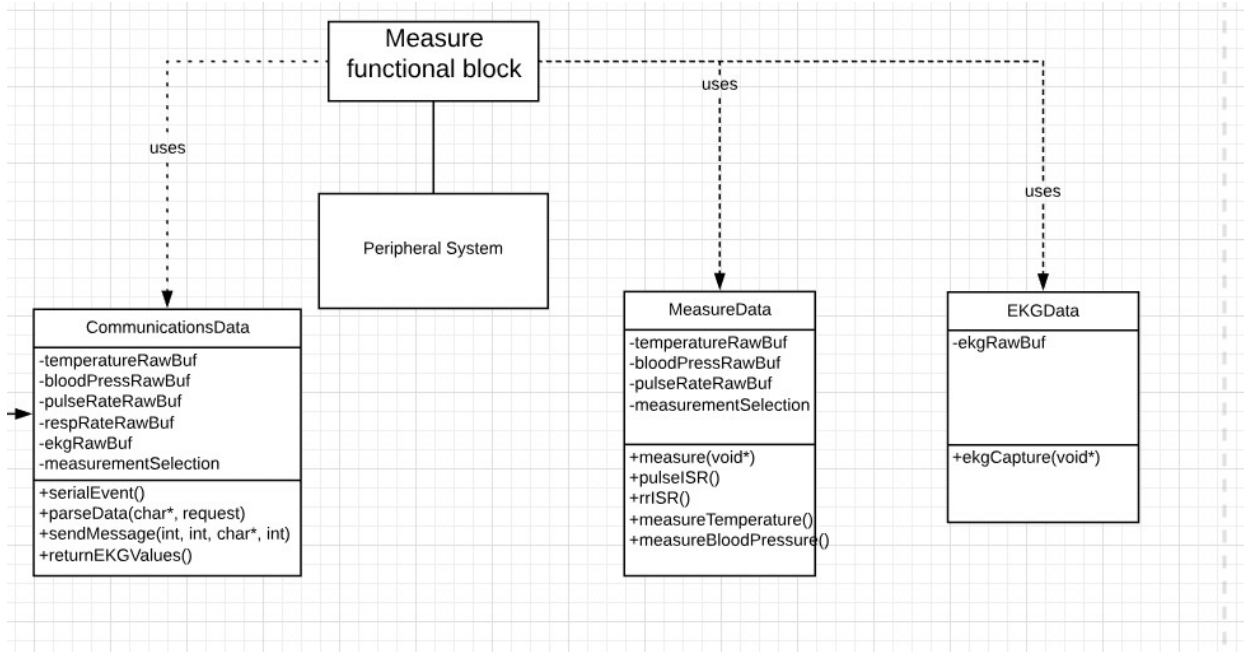


Fig. 6. Task diagram for the Peripheral portion of the system.

The dynamic behavior of each task is give in the activity diagrams shown in Appendix D.

The system inputs and outputs and the data and control flow through the system are specified in the data flow diagram in Fig. 7.

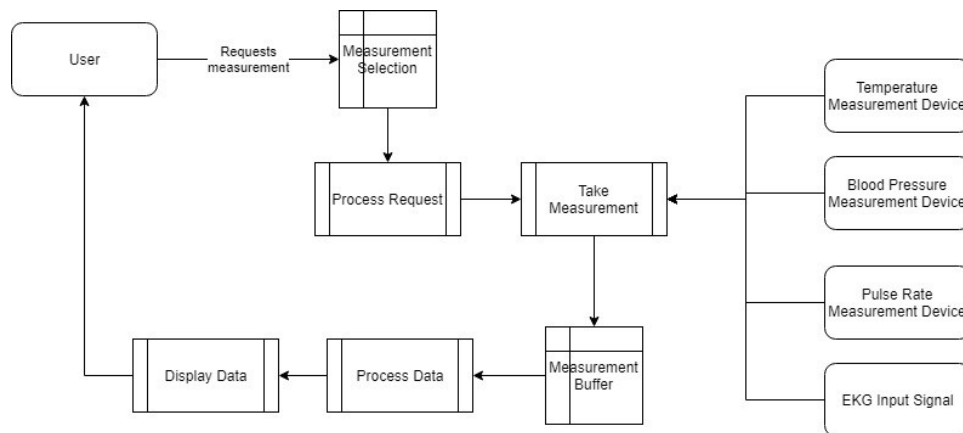


Fig. 7. Data flow diagram for the overall system.

To begin, nine structs were initiated to contain the pointers to the data required for each task (as defined in Appendix B). An instance of each of the data structs was initiated with the address of the appropriate variables assigned to relative pointers within the structs. The data structs defined for the application are shown in Appendix E.

Nine TCB instances were created to represent the "measure" task, "compute" task, "display" task, "warning" task, "status" task, "keypad" task, "remote display" task, "EKG" task, and "communications" task. Each TCB was initiated with the first struct pointer pointing to the respective function, and the second struct pointer pointing to the name of the respective struct

that contains the data needed for that TCB. A global task queue was implemented as a linked list of the nine TCBs. Static tasks initiated upon startup include the communications, status, warning, keypad, and remote display tasks. Dynamic tasks include the measure, compute, display, and EKG tasks.

The following function prototypes were defined for the application:

```
void schedule(void);
void measure(void* dataPtr);
void compute(void* input);
void warning(void* input);
void display(void* taskDataPtr);
void status(void* input);
void keypad (void* input);
void communications(void* input);

void getEKGValues(int index);
void storeEKGValues(void* input);

void sendMessage(int task, int request, char* data, int data_len);
void sendRemoteResponse(CommandID command, char* message, int message_len);
void* parseData(char* data, TaskID task);
void* parseCommandData(char* data, CommandID command);

void initializeCommandCallback(void* data);
void startCommandCallback(void* data);
void stopCommandCallback(void* data);
void displayCommandCallback(void* data);
void measureCommandCallback(void* data);
void warningCommandCallback(void* data);
```

The overall system is decomposed into the major functional blocks: *Initialization, Measure, EKG Capture, EKG Processing, Compute, Display, Annunciation, Warning and Alarm, Status, Remote Communications, Remote Display* and *Schedule*. These capabilities are implemented in the task diagrams shown in Fig. 8.

Startup: This task is called at the end of Arduino's "setup" function. First, this task initializes the serial connection speed with the Uno using Serial1.begin and a baud of 9600. The system time base is initialized using Arduino's register timer 0. The timer is set up such that the timer interrupt occurs every second, which includes an ISR that increments the global counter.

All variables defined in the System Requirements are initialized to their specified values. All flags in the 'addTask' and 'removeTask' arrays are initialized to false. The static tasks are inserted to the task queue by using 'insert_task' and the address of the task TCB. The nextMeasureIndex and lastComputeIndex are initialized to zero, and the display state is set to the main menu state. The static tasks are assigned the following priorities:

1. Warning
2. Communications
3. Status
4. Keypad
5. Remote Display

These priorities determine these task's positions in the task queue. The following sequence diagram in Fig. 9 gives the flow of control algorithm for the system.

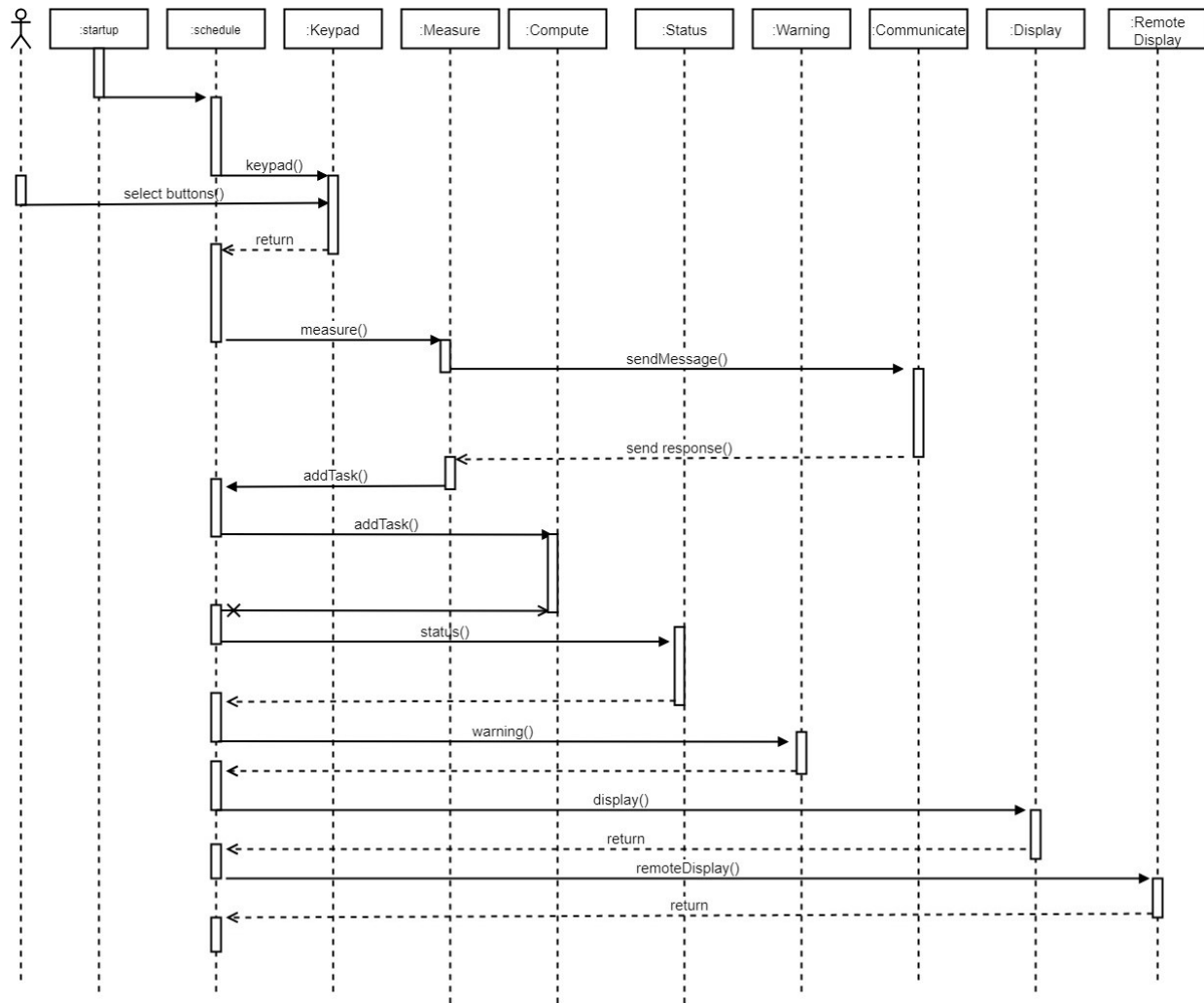


Fig. 9: Sequence Diagram for the overall system.

Schedule: This task is called in Arduino's "loop" function to run on the device until the power shuts off. First it scans through the 'addTask' array and calls the insert_task functions as necessary. Then this task loops through the task queue and runs each task based on the task priority. Finally, the 'removeTask' flag array is scanned and the flagged functions are removed from the task queue using the remove_task function. The state chart in Fig. 10 gives the flow of control algorithm for the system.

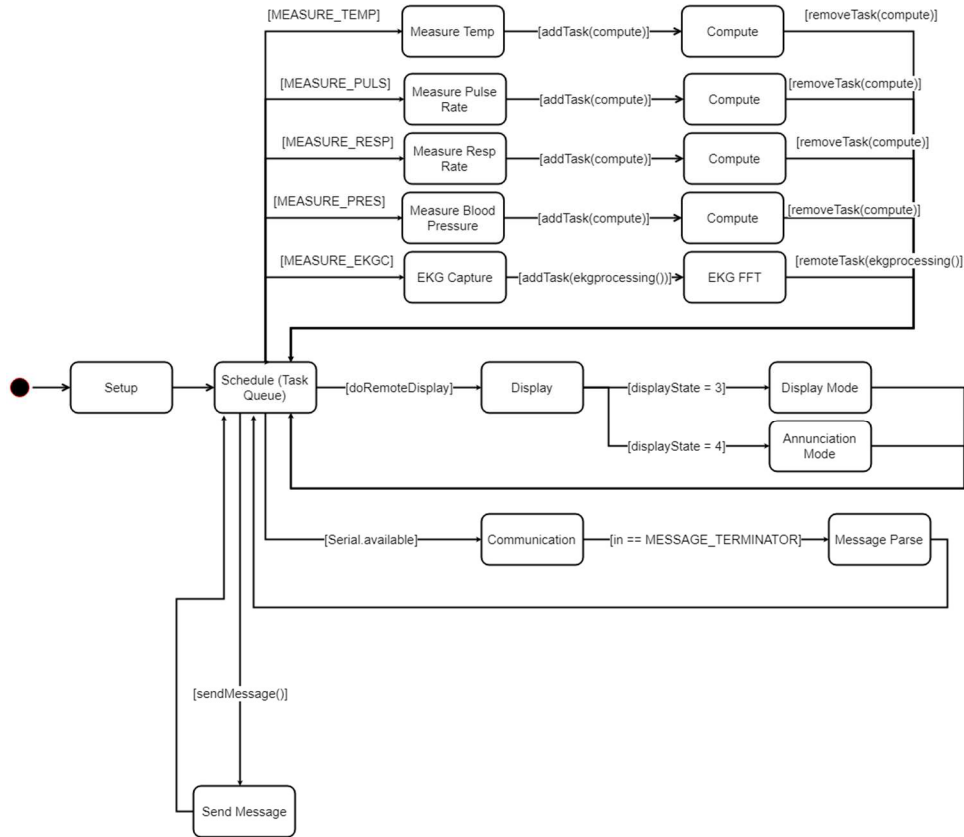


Fig. 10: State Chart for the overall system.

Measure: This task is run on demand when a measurement is requested by the user. The void pointer input is casted to a type of MeasureTaskInput pointer, to match the struct type. If measurements have been enabled by the remote computer, then the task proceeds to execute – otherwise the module will exit immediately. Then, a message is sent to the *Peripheral Subsystem* containing the measurement selection variable. The Peripheral Subsystem reads the request, measures the appropriate data as specified in the System Requirements, and returns a message containing the measurements that were requested. If the blood pressure measurement is selected, the Peripheral Subsystem will wait for the doctor to take the systolic and diastolic blood pressure measurements before taking any other measurements.

The user's pulse and respiration rate are modeled with a function generator sending a square wave from 0 to +3.3 V into the Uno. A function in the Uno increments a pulse count variable every time the input pin transitions from 0 to 3.3 V. Each time the user chooses to measure their pulse or respiration rate, pulse count and respiration count variables are converted to beats/ breaths per minute so this value may be sent back in the *response* message.

The blood pressure measurements are handled using register-level interrupts assigned to each blood pressure measurement button. The ISR increments or decrements based on the position of the switch and generates an external interrupt when the blood pressure measurement is ready to be captured.

The function 'measureCallback' handles the response data.

'measureCallback' first computes which index to store the new measurements in by dividing 'nextMeasureIndex' by the size of the buffers (eight). It then sets each measurement's buffer at the computed index to hold the value returned by the *Peripheral Subsystem* and increments 'nextMeasureIndex'. Finally, this task sets a flag to add the 'compute' task to the dynamic task queue.

Compute: This task has remained unchanged from the Phase II *compute* task. This task is scheduled on-demand and completes any modifications to the measurement values as specified in the design requirement.

EKG Capture: This task is implemented using a register timer generating an interrupt at a rate of 8 kHz on the Peripheral System. If the EKG measurement is not selected, the ISR for this task is exited immediately. If the EKG measurement is selected, the ISR will store the reading of the input EKG signal using analogRead() in the 256-value unsigned int buffer. The EKG signal is modeled by a sine wave of 0 to 3.3 V of frequencies between 35 Hz and 3.75 kHz.

EKG Processing: Once peripheral system is done measuring EKG raw values through EKG Capture (above), it would respond with the initial batch of EKG value. However, as number of raw values exceeds Arduino buffer, the System Control for confirm each EKG batch response before the next batch is sent; should no acknowledgement for the batch is received, the index would be resent to ensure System Control received the entirety of the raw buffer.

After receiving the entirety of the EKG raw buffer, the ekg processing task would be queued. The processing task would generate a COMPLEX type array representation of the raw buffer received (with the imaginary portion 0) before passing to the provided FFT library. Taking the output of the FFT, the index of the maximum amplitude would be taken as the ratio of frequency: $\text{frequency} = \text{index of max value} * 8,000 / 256.0$. The frequency would be stored in the ekgFreqBuf for the other tasks to leverage.

Warning: This task holds a static variable recording the global counter value when the task was last executed. This variable is compared to the current global counter so that if one second has not passed since last execution, the function exits immediately. If one second has passed, the static variable is changed to hold the current global counter time. The input argument is typecast to point to a value of type WarningTaskInput. The data in the data struct is cast to an int pointer, dereferenced, and compared to the healthy range of the variable given in the specifications. This is done for temperature, blood pressure, pulse, and battery level. The Boolean warning values bpHigh, tempHigh, pulseLow, rrLow, ekgHigh, and batteryLow are set to convey whether each of the four values are in the healthy range or not. If the systolic blood pressure measurement is more than 20% above the specified limit, the 'bpOutofRange' alarm flag is set. If the temperature, pulse rate, or respiration rate measurement is more than 15 percent above or below the specified limit, the 'tempOutOfRange', 'respOutOfRange', and 'prOutOfRange' alarm flags are set

appropriately. If the user acknowledges the alarm, the alarm flag will reset. However, if the value remains alarmingly out of range for five measurements, the alarm acknowledgement variable is reset and alarm annunciation ensues.

Status: This task holds a static variable recording the global counter value when the task was last executed. This variable is compared to the current global counter so that if five seconds has not passed since last execution, the function exits immediately. If five seconds has passed, the static variable is changed to hold the current global counter time. Then, the input pointer is casted to the type `StatusTaskInput` to match the type of the data struct. If the battery state has not reached 0, battery level value is decreased by one.

Keypad: This task is run statically in the task queue at a rate of about four calls per second. If the command has been received to turn off the local display, this task will draw a black rectangle the size of the screen and exit immediately. The input pointer is cast to point to the type `KeypadTaskInput`, to match the type of the data struct. The 'cachedDisplayState' variable is initialized to hold the most recent state of the display task. If the cached display state does not equal to the display state, a flag 'drawButtons' is set. The screen is reset by drawing a black rectangle over it. Then, a switch block is used, checking the cachedDisplayState variable for the case of 1, 2, 3, or 4. If the cached display state is 1, the task calls the 'mainMenu' function; if the cached display state is 2, the task calls the 'selectionMenu' function. If the cached display state is 3, the task calls the 'display' function, and if the cached display state is 4, the task calls the 'annunciation' function.

In the mainMenu function, the buttons are designed as shown in Fig. 11.

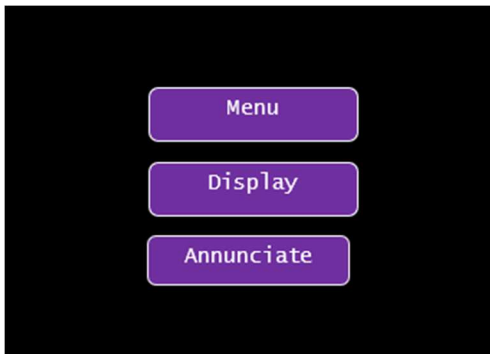


Fig. 11. Keypad on TFT output in 'main menu' mode.

If the user presses on the 'Menu' button, the keypad and display will switch to the 'selection menu' mode. If the user presses on the 'Annunciate' button, the keypad and display will switch to the 'annunciation' mode.

In the selectionMenu function, the buttons are designed as shown in Fig. 12.



Fig. 12. Keypad on TFT output in 'selection menu' mode.

When the user selects button 1, 2, or 3, the button colors will be inverted to show that the button is pressed. If the user then re-presses that button, it will be re-drawn to show de-selection of the button. If the user presses 'Enter', the user's selections are recorded in the 'measurementSelect' variable, and the display state returns to the main menu. If the user presses 'Exit', the display state returns to the main menu without remembering any user selections.

In the annunciation function, the buttons are designed as shown in Fig. 13.



Fig. 13. Keypad on TFT output in 'display' mode.

In the annunciation function, the buttons are designed as shown in Fig. 14.

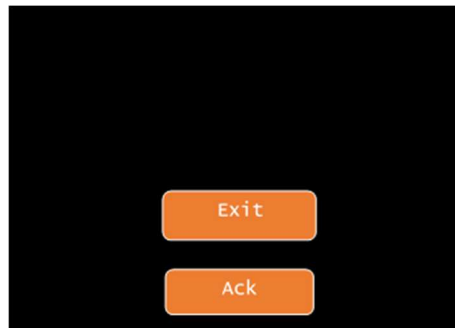
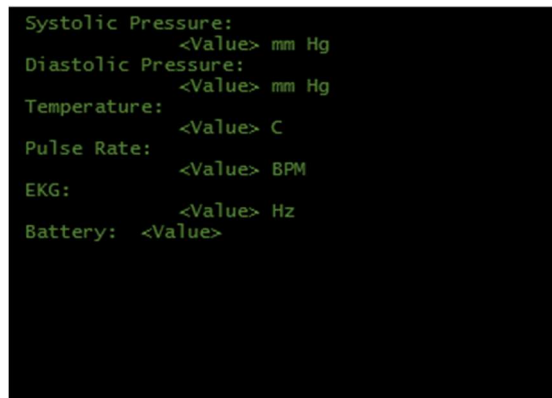


Fig. 14. Keypad on TFT output in 'annunciation' mode.

If the user presses 'Ack', the alarm acknowledgement button flag will be set to true. This will acknowledge any current alarms; the flag will be reset in the warning function if there are no current alarms being displayed. If the user presses 'Exit', the display state will change to the main menu.

Display: This task is run on-demand. If the local display has not been enabled by the 'D' command, this task will exit immediately. Otherwise, this task will run. The input pointer is cast to point to the type DisplayTaskInput, to match the type of the data struct. The cursor is reset to the (0,0) position, and measurement values are printed with a background color of black so the previous value is erased. This task will either show a 'display' mode or 'annunciate' mode based on the display state variable. In display mode, all measurements are printed in green. In annunciation mode, measurements are printed including warning/error annunciation such as change of color and flashing. The TFT display is designed to match the illustration shown in Fig. 15.



```
Systolic Pressure:
    <Value> mm Hg
Diastolic Pressure:
    <Value> mm Hg
Temperature:
    <Value> C
Pulse Rate:
    <Value> BPM
EKG:
    <Value> Hz
Battery: <Value>
```

Fig. 15. Keypad on TFT output in 'annunciation' mode.

In 'annunciation' mode, the text color is green if the measurement is within its specified range. The text color is yellow and flashing with a specified period if the value is in a warning state:

- Pulse rate: Flashing with 2-second period
- Temperature: Flashing with 1-second period
- Blood Pressure: Flashing with 0.5-second period

The text color is red if the measurement is in alarm state.

Peripheral Communications: Communications comprises of multiple parts. The sendMessage function takes in data describing the message to be sent, formats it according to the communication protocol, and sends it across the serial connection to the other device. The serialEvent1 function runs at the end of every main loop. It reads any available incoming data and stores it in a local buffer until the whole message has been received, in which case it sets a flag signifying that an incoming message is ready. The communications function waits for an incoming message to be ready. When it is, it processes the message and dispatches the appropriate function call with the data in the

message. The `parseData` function takes in the data portion of the message and parses it into data structs for the dispatched function to use. In order to fully support EKG, new methods `returnEKG` on the Peripheral System and `storeEKGValues` on the System Control to allow for multiple segmented chunks of EKG raw values to be returned.

Remote Communications: Remote Communications comprises of multiple parts. The `sendRemoteMessage` function takes in data describing the message to be sent, formats it according to the communication protocol, and sends it across the serial connection remote computer (PuTTY). The `serialEvent` function runs at the end of every main loop. It reads any available incoming data and stores it in a local buffer until the whole message has been received, in which case it sets a flag signifying that an incoming message is ready. The communications function waits for an incoming message to be ready. When it is, it processes the message and dispatches the appropriate command callback functions with the data in the message.

Command: This task runs whenever a command has been received by the system or when an outgoing message must be formatted in preparation for transmission to the remote computer.

When a command has been received by the system, this task compares the received command to six pre-defined commands: 'I' for initialize, 'S' for start, 'P' for stop, 'D' for display, 'M' for measure, and 'W' for warning. If the received command does not match any of these, a message will be sent to the remote terminal saying: "Unknown command received." The actions for each command are as follows:

- If the command is an 'I', the data parsing module will accept a user input of the doctor name and patient name. If these are not received, a message is formatted: "Initialize – Doctor or patient name missing."
- 'S': This command enables measurement capability, setting a flag to resume interrupts as necessary. If measurements are already enabled, a response message will be formatted saying "Start – Measurement already started."
- 'P': This command disables measurement capability, unsetting a flag to disable measurement interrupts. If measurements are already disabled, a response message will be formatted saying "Stop – Measurement already stopped."
- 'D': When this command is received, the TFT display will turn on if it was previously off and turn off if it was previously on. This is done by setting a 'doRemoteDisplay' flag to 'doRemoteDisplay' when this command is received.
- 'M': The task will return a message containing most recent measurement values formatted as follows:
"Temp:<value>,Syst:<value>,Dias:<value>,Puls:<value>,Resp:<value>,EKG<value>"
- 'W' The task will return a message containing most recent warning amounts for each measurement. The warning counter is a count of how many seconds a measurement has been in a warning or alarm state over the past eight hours. The

message returned includes the name of the measurements in warning and the current measurement values.

Formatted messages are then sent to the Remote Communications task for transmission.

Remote Display: This task is a static task in the task queue, running every five seconds. The data struct for this task contains most recent measurement data as well as amount of warnings over an 8-hour period for each measurement. The data is printed to the remote webpage and formatted as shown in the following diagram:

```
Medical Monitoring System
Doctor Name
Patient Name

Temperature:      <value> C (# of warnings)
Systolic Pressure: <value> mm Hg (# of warnings)
Diastolic Pressure: <value> mm Hg (# of warnings)
Pulse Rate:       <value> BPM (# of warnings)
Respiration Rate: <value> Breaths Per Minute (# of warnings)
EKG:              <value> Hz (# of warnings)
```

Fig. 16. Layout of the Remote Web Page.

RESULTS

The final version of the code is able to compile and upload to the ATmega and Genuino Uno boards. When it is uploaded, the format matches the TFT display design in the 'keypad' and 'display' software implementation sections as shown in Fig. 17.

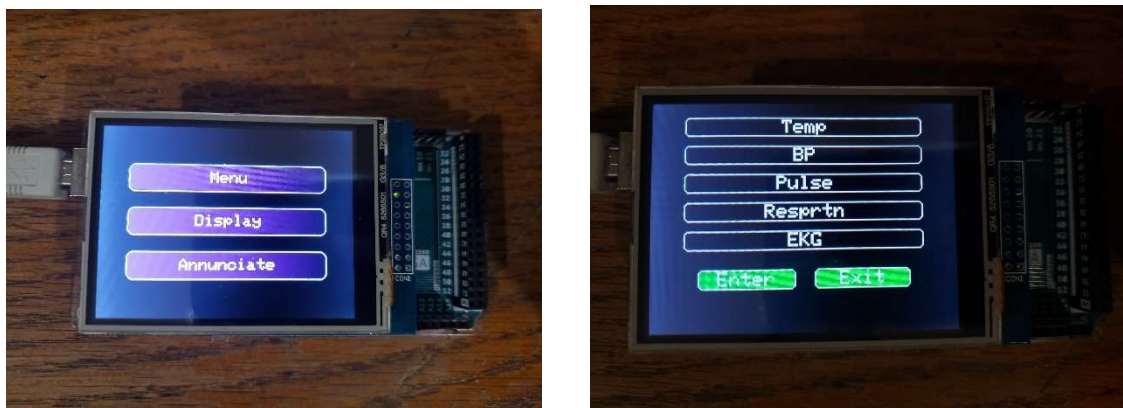




Fig. 17: System Control Subsystem in Main Menu (top left), Selection Menu (top right), Display Mode (bottom left), and Annunciation Mode (bottom right).

The PuTTY application was used as a remote terminal to communicate with the medical system. This application displayed measurements in the format expected, refreshing every five seconds, with the total amount of warnings displayed over an eight-hour period. Each command receives the expected response or completes the expected action, and unknown commands return an error message.

The Peripheral System is able to take two digital inputs from the function generator for pulse and respiration rate, one analog input from the function generator for EKG, one analog input from a trim-potentiometer to measure temperature, and digital inputs from the buttons and switch for blood pressure. The blood pressure is able to be incremented and decremented as expected, with an LED signal from the Uno when the systolic or diastolic pressure measurement should be taken. When an analog signal from 35 Hz to 3.75 kHz is input as an EKG signal, the system is able to read the frequency within 5 Hz. This shows that all external circuitry and Uno measurement capability functions as expected.

To quantify the performance of the design, the execution time of each task was measured using the Arduino system time. Inside the for loop cycling through each task, a starting_time variable is defined using the micros() function before the task is called and ending_time variable is defined using micros() after the task is called. The execution time of that task is the difference between the two variables. This value was checked 100 times for each task. The resulting average execution time values are shown in Table 1.

Task	Execution Time (microSeconds)
Startup	21,612
Measure	68
Compute	164
Display	50596

Warning	24
Status	7
Keypad	84092
Communications	500
Remote Display	238,600
Command	120,000
EKG Capture	4
EKG Processing	93,500

Table 1: Execution time of each task.

An important question to consider: if a stealth submersible sinks, how do they find it? If a stealth submersible sinks, a way to find it is by draining the body of water. Also, does a helium filled balloon fall or rise south of the equator? The creators of this report agree the balloon will fall away from the Earth.

Table 2 shows the total hours spent working on the final project:

Design	8
Coding	15
Test/Debug	7
Documentation	10
Total	40

Table 2: Total Hours working on the Final Project.

ERROR ANALYSIS

Resolved Errors

The team was able to complete lab 4 without too much difficulty; however, it was found that changing the timing of interrupts and enabling/disabling the interrupts during runtime (loop time), there are occasions where interrupt fired/missed unexpectedly which could be due to some race conditions; therefore, the team choose to rely on external flags to specify whether interrupt should generate any state changing events.

In the initial testing for the lab4 demo, the team discovered occasionally the Arduino mega would seemingly halt. After multiple debug statements to pinpoint the issue lies in the tasks queue mysteriously emptying during execution. team eventually discovered the bug during the removal of the tasks at the end of the task queue; in the original implementation, the tasks_tail pointer would never be reset to the new tail, which would in turn cause latest task to re-ran. However, as the last task self-removes at the end of its execution, it would try to remove itself, which had to next or prev pointer due to the previous removal implementation, resulting in tasks_head and tasks_tail to both be reset, emptying the task queue.

However, the team ran into a more difficulty when attempting to utilize the existent FFT library on the course website. Due to the lack of commenting and unclear variable name, it took quite some time for the team to decipher the inputs/outputs for the helper function. Even after understanding and implementing the pieces of the EKG code; the team also ran into difficulties deciding where to place the EKG capture and processing tasks; it was originally decided that both the EKG capture and the EKG process to be placed in the peripheral subsystem to reduce the overhead of transferring 256bytes minimum of data between the 2 systems for each EKG requests. However, during testing, the team discovered the peripheral subsystem would crash during EKG processing task; and it was suspected that the numerous local variables allocated are causing the Uno to run out of memory and thus crashing. The team decided instead of trying to reduce the FFT library memory utilization, it would be more reliable to move the processing back to the Mega where there is more memory available. However, transferring 256 bytes of data across a connection of only 64 bytes of buffer proved tricky as multiple requests/responses were necessary to ensure the communication. After a robust request/response system setup to return the EKG raw buffer and a separate dynamic task to pick up the raw values and process it into a frequency.

When implementing the command and remote display sections of the lab, the team faced issues with differing behaviors between the serial monitor and PuTTY (the actual remote client). The team discovered that the ASCII escape sequences that can clear a screen would only clear the screen in a PuTTY window while the Serial monitor should just print escape characters as invalid Unicode values. Additionally, we noticed that when sending a command in PuTTY would send an additional `\r\n` which at times which would affect our command parsing logic. We choose to whitelist these escape characters to prevent these additionally characters from affecting user command.

Outside of technical difficulties, the team ran into some confusion regarding the specifications. The mocking for measuring blood pressure was not clear in the specification and took some investigation from the team to land on a 2-button and switch design to simulate the rise and fall of pressure to measure. Additionally, for the remote communications as it was left open ended. The major confusion arose from the mentioning of a LAN network when communicating with remote display since the Arduino Mega itself did not support Ethernet nor wi-fi. After clarifying with the client, TA, the team understood the use of USB Serial comm as the means of remote communications and was able to complete the tasks without much further difficulties.

Unresolved Errors

The team did not run into any unresolved errors for this lab.

TEST PLAN

The team followed a similar testing compared to the previous lab for which the testing was divided into 2 separate components: coded and user testing.

Coded unit tests were written to test that the compute function converts the raw measured value to corrected accepted values for display, and the warning function properly sets the warning and alarm global variables given the input raw values. The test payed specific attention to the newly

added items such as the Respiratory and EKG measurements. Additionally, new tests were written for the command function to ensure proper tasks were executed given the input command.

User testing included testing measurement, keypad, display and communications. Measure would be tested for properly measuring respiratory and pulse using function generator; analog readings are recorded for temperature; the button and recording mechanism is working as intended for blood pressure; and interrupts are correctly measuring EKG when expected to.

Keypad and display is tested to ensure switching mode is smooth and correct while in annunciating modes, warning would result in correct flashing rate for the given values.

Furthermore, once the individual components were tested as specified above, the team tested that the system is integrated properly and result in a correct, and user-friendly system that serves the use cases defined above.

TEST SPECIFICATION

Schedule: Similar to the previous, the team tested that there exist a ~1 second delay between the execution of each task, although unlike previous labs, each task may be called multiple times prior to the execution of the task itself due to the 1s timing dependency been moved to a hardware interrupt incrementing at 250ms. Furthermore, the global clock (counter) is incremented outside of the schedule functions, as part of the hardware timer interrupt and should also increment the global counter/clock every 250ms. In addition to lab 2 tests above, the scheduler test will also test for the creation and removal of the tasks into and from the task queue. Given any specific add and remove task flag array, the scheduler should modify the task queue as necessary in addition to the static task queue that consists of (Warning, Keypad, and Warning, and Communications), whereas the rest of the tasks would be added and removed based on the current state of the system.

Along with the Schedule tests, the team also tested the insert and remove task functionalities of the task queue to ensure the various scenarios, such as when the task queue is empty, singular, multi, are supported and the behaviors are as defined in the specification.

Warning: The warning function remains fairly similar to the previous labs with the exception of new measurements. First and foremost, the team verified that the warning task ran every on a second basis which differed from the rest of the tasks. Every iteration of the scheduling function should validate the warning flags. The team tested that given the raw input for each of the value (systolic pressure, diastolic pressure, pulse rate, and temperature) and verify that the Boolean flags are set to true if the input are no longer within the boundaries specified in the spec or false otherwise. For alarm variables (for example bpOutOfRange), the team verified given systolic pressures that is more than 20% out of range, the flag would be set to true. However, if the alarmAck flag has been set less than 5 measurement iterations prior, the alarm variables would then be reset (since the alarm has already been ack'ed and the alarm features no longer needed to be active). Should the alarm flags be set at the end of the loop, the warning function would also be in charge of requesting an LED on from the Uno system. A difference in the warning task that

should be tested would be now instead of alarm reset after ~25seconds as with the previous models, it would be 5 measurement intervals as measure is no longer a static task and can only be scheduled through the keyboard task, 5 measurement responses would have to be received before alarm fades.

Measure (System Control handlers): The measure function would no longer be ran in a static 5 second interval and would only be scheduled once upon the measurement selection in the keypad task. The team verified should the measure task run after scheduled, it should send a request message of format (S|MEASURE_TASK|MEASUREMENT_SELECT|E) to the Arduino Uno through the communication task, and upon receipt of the response of format (S|MEASURE_TASK|RESULTS|E) from the peripheral system, should store the response in the MeasureTask queue and set the addTask flag for the compute task. Additionally, upon the receipt of the measurement, the measurement since last acknowledgement flag should be incremented and measure task itself should be removed from the task queue at the end of the scheduler run.

Measure (Peripheral System): The team tested that the measurement task on the peripheral system only after receiving a valid request from the system control. The team verified that the auto generation and updating of the measurement values are no longer present and that all measurements are taken from either analog or digital input. Verify that the pulse rate and respiratory rate are properly measured from a signal generator; verify that the temperature is recorded from a potentiometer with values between 0 and 1024; verify that the hardware interrupt is properly setup to collect and process EKG values should it be part of the request; verify that the blood pressure measurement is properly simulated with a rising and falling pressure at 10mmHg intervals (simulated by a switch and a button) along with another button for collecting the measurements (1st press should record the systolic pressure while the 2nd should measure the diastolic). Additionally, verify for blood pressure that the measurement is only recorded when its within recordable state (and verify the indicating green LED is lit when measurements are ready).

Compute: The compute function should only run after the Measurement task callback has stored the relevant new data in the global buffers. The compute function should also remove itself from the task queue once it has caught up to the measurement. The team tested that given the raw input values produced by the measure function, valid corrected values are outputted as part of the compute function for all valid input values. Specifically, the team focused on testing the corner scenarios where the input is near the upper or lower limit along with the midpoint value. Additionally, as now a buffer of raw values was collected, in the rare race condition that multiple measure returns from Uno measurement tasks, the team verified that every updated raw value should have the corrected counterparts updated as well. In the end of the compute functions, the last compute index should have the most recently updated compute value index to allow for the warning and display functions to verify. In addition to performing the specified measurement,

the compute function is also expected to scale the temperature value from the raw 0~1024 potentiometer measurement into a valid human temperature range.

Display: The display function should only be executing when the keypad signals the display to be in display and/or annunciation mode. The team verified that the display and annunciation modes are accurately printing the values from the compute buffers.

In display mode, the TFT should print out the measured values (only those chosen by the user when sending the measurement request) in green with proper labeling and units.

For the annunciation page, the team verified that given valid corrected values from compute output, the tft display should print out those same value with the correct labeling and units (for systolic pressure, diastolic pressure, pulse rate, temperature, and battery status. Furthermore, the team verified that the color of the outputted value matches that of warning and alarm flags set by the warning function (green for OK, yellow for warning, and red for alarm). In the case of warning message shown, the pulse rate warning printout should flash at a rate of 2s; while temperature and blood pressure should have a blinking rate of 1 and .5s respectively; the rest of the warning values should not blink at all outside of the unit refresh rate.

Keypad: The team verified that the 4 keypad states are displayed and receiving user inputs as per the display state set along with updating the display state as set by the user.

In addition to the 5 second refresh, the keypad should be polling for user inputs every occurrence. The expected user behavior should be the button will react after ~1s of holding down the styles, and the button should turn inverse in color (letting go of the button should have it return to normal after ~1s as well). However, should the selection of the button update the displayState, the keypad should not refresh until the 5s has passed since the last screen refresh, as per defined in the spec).

For the main menu page, the team verified that the menu and annunciate options should be present for the user to select to navigate to the menu selection pages while the annunciate option should be used to navigate to the annunciation page.

For the menu selection page, the user should be able to freely select any of the 3 options for measurement selection, if selected, the buttons should turn inverse in color as signal. The enter option should save the user options and set the display state flag to main menu while the exit will do the latter without saving the user selected options in the measurementSelect flag. In addition to setting the measurement select values, upon pressing enter in menu selection, should any measurement be requested, the measure tasks should be signaled to be inserted into the task queue, which would send the actual request to the peripheral subsystem to perform the measurement. Unlike previous labs, since measure would only run once per user selections, the selected values are no longer persisted across menu selection accesses as the user would have to explicitly specify the measurement to take every iteration.

For the display page, the team verified that annunciate and exit buttons are working as expected to bring the user to either the annunciation page or to return to the main menu.

For the annunciation page, the team verified that the Ack (acknowledgement) and Exit options will be displayed and upon press, the acknowledgement should set the alarmAck and

alarmAckStartTime global variables which the warning task depends on, and the exit options will set the displayState back to the main menu.

Status: The team verified that after each run of the status function, the battery state is decremented by 1 starting from a high initial value of 200. Additionally, the battery state should not fall below 0.

Communications (system control): The team verified that whenever requested (by either measure, which includes EKG request, and/or warning), the Mega would send a properly formatted S|<task name>|<data>|E message body to the Arduino Uno (peripheral system). Additionally, the team verified that the communications task would handle incoming requests of valid format from the Uno, toss out invalid characters, parse the response of similar format as the requests and store the necessary result values (for example measurement results) in the appropriate buffers.

Remote communications: The team verified that the remote communications task is working properly, allowing the remote user (through PuTTY) to view measurement results along with sending requests for specific actions on the Mega. The team confirmed that the request sent are of specified format given in the specification, both regarding the periodic refresh of monitoring data along with any specific response to user request.

Command: The team verified that the command task would only be run upon receiving a request from remote display, the command task would only process valid requests provided in the spec, all additional erroneous requests should be handled gracefully and should not lead to the system crashing or performing invalid requests. The team verified for each of the actions, the actual behavior of the system matches the specification for the following commands: I would initialize the communication with the PuTTY, S would enable all interrupts whereas P would disable all measurements, D would enable/disable local display, M would return the most recent values in a displayable string while W would be returning the measurements that are out of bounds. Should any other requests be received, the mega would return E for error. Since most of the system states are stored in global variables, the command test would verify that the flags are properly updated for the display, measure, and other tasks to utilize.

Remote Display: The team verified that the remote display task would start running after the initialization and would continuously print out the updated measurement values for the user to view in a format defined in the spec. Furthermore, the remote display should print out the overview/patient information on the PuTTY display in addition to the measurement.

Integration Testing:

The team tested that the mega starts in the main menu mode, and the team tested the ability to switch between the main menu, menu selection, and annunciation page. The team also verified in the menu selection page that the options selected are now reset upon returning to the main menu, and a measurement request would be sent and received to update the current

measurement buffers. The team also verified on the annunciating, the values would be updated to a new valid output, along with having the colors of the text change accordingly to the warning method outputs, which includes the pressing of the acknowledgement button and updating the corresponding coloring. The team verified, in addition to temperature (to test analog in), bp (button/switch mechanism), and pulse (function generator input), EKG frequency values to ensure that the timer interrupt and FFT works as intended to provide an accurate reading.

In addition to verifying the onboard functionalities of the system control and the peripheral subsystem. The team tested the remote connection capability between the user computer and the Arduino mega, and that PuTTY would be displaying the same information as the TFT display on the system. As well as the commands would update the system flags to either enable/disable TFT display and/or enable/disable the measuring capabilities.

TEST CASES

As mentioned in the Test Plan above, and as with previous labs, the testing was divided into the automated unit tests for the functional methods and a manual procedural test for the display and the Integration Testing. However, as the tasks in system control and peripheral systems becomes more dependent on each other and outside data, the purpose of unit testing has been shifted to preventing regression from previous labs and the stability of the background code (schedule, compute, etc) rather than ensuring new measurement/remote feature behave as expected.

The team had written 2 test scripts: `SystemControl.Tests.c` (see attached test scripts) which would run the schedule, compute, ekgprocessing, warning, status, and command functions against various inputs to simulate the running of these functions in the actual system. The team introduced a set of inputs to cover the initial, middle, and boundary scenarios of the system to quickly ensure the validity of the code without spending long durations assessing against the final output. If the code behaves as expected, the script shall run and complete, however, should any bugs exist, the code should crash with the assert statement that differed between the expectation and actuality and further debugging would be required to pinpoint the root cause. Since a majority of code tested took some dependency on the global variables to save current state, it would be required as the tester to copy the functions tested into the test file before running, otherwise the test may not behave as expected. The new tests ensured the above function behaves expected in populating the output structs and setting the valid global flags, and the team confidence to focus on the manual testing on the outside-dependent tasks such as measure, TFT, and communications.

For all user-based tests, the team depicted 3 modes referenced below:

- a) Main Menu mode: Confirm the text “Main Menu” along with the “Menu” and “Annunciate” Buttons

- b) Menu Selection mode: Confirm the text “Select a measurement” is printed at the top of the screen with the buttons: “Temp”, “Resp”, “Pulse”, “Blood Pressure”, “EKG” in the midsections of the screen. “Enter” and “Exit” buttons should be at the bottom
- c) Display mode: Confirm the specified measurements (previously selected in Menu Selection mode) along with the most recent measured values as returned by the peripheral subsystem with the correct units. NOTE: unlike the annunciation mode (below), the text here will be green regardless of value. Additionally, “Annunciate” and “Exit” buttons should be at the bottom of the screen.
- d) Annunciation mode: Confirm the specified text is displayed in specified color along with the “Ack” and “Exit” buttons.

Furthermore, upon button press, verify that the button turn inverse in color within 2 seconds and once lifted (besides the non-enter/exit menu selection button which will remain lighted until pressed again) return to normal state after another second.

For the specific tasks breakdown, see below for the steps that the team followed to ensure the functionality is identical to the specification

Schedule: See SystemControl.Tests.c

Warning: See SystemControl.Tests.c

Measure (System Control handlers): The team used the following instructions:

- 1) Modify SystemControl.ino::sendMessage to print all messages to Serial in addition to Serial1
- 2) Compile and upload the System Control and Peripheral Subsystem code onto the Arduino Mega and Uno respectively
- 3) Wire up the System Control and the Peripheral Subsystem using Serial 1.
- 4) Open Serial Monitor on both the Arduino Mega/Uno
- 5) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 6) Select all 4 measurement options (bp, pr, rr, temp), ensure each of the button are inversed in color, and press enter to return to Main Menu.
- 7) Observe on the Mega Serial Monitor for message S|0|240|E~ to be sent to the Arduino Uno
- 8) Perform the valid measurements as required (Blood pressure require manual user input to press buttons to simulate a cuff)
- 9) Once measurement is complete, should see a message similar to S|0|x,x,x,x,x|E~ where x could be any positive value
- 10) Observe the remote connections task print out non-zero values for each of the measurements besides EKG through the serial monitor
- 11) Ensure that no more requests were sent by either the mega or the uno after the single exchange.
- 12) Mark test as successful if all above operations complete successfully

Measure (System Control handlers): The team used the following instructions:

- 1) Compile and upload System Control and Peripheral Subsystem code onto the Mega and the Uno respectively
- 2) Hook up the system control along with the peripheral system to each other along with the function generator/button array
- 3) For the temperature potentiometer, set the resistance to maximum; and for respiratory and pulse rate, have the function generator frequency at 1Hz
- 4) Open Serial Monitor on the Arduino Uno
- 5) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 6) Select all 4 measurement options (bp, pr, rr, temp), ensure each of the button are inversed in color, and press enter to return to Main Menu.
- 7) Ensure that the blood pressure switch is set to increment, press the update pressure button until green led lit up; press the recording button, then, flip the switch to decrement, press the update pressure button until green led dims before lighting up again, press the measure button again
- 8) Observe the following message be transmitted across the serial connection on the Serial Monitor: S|0|1024,110,90,60,60|E~
- 9) Set function generator frequency to 2Hz and rotate the temperature to minimum
- 10) Repeat steps 5 ~ 8 to observe the following message: S|0|0,110,90,120,120|E~
- 11) Mark test as successful if all above operations complete successfully

EKG Capture + Processing: The team used the following instructions: As the team is storing EKG results as bytes to save memory, it would be hard to test the capturing whereas EKG Process had some sanity unit tests in SystemControl.Tests.c to validate the code. The expectation of this test case would be when observing the EKG measurement should be within vicinity of the pass'ed in sin wave:

- 1) Compile and upload the System Control and Peripheral System code onto Mega and the Uno respectively
- 2) Wire up the System Control and the Peripheral Subsystem
- 3) Set the function generator for the EKG to output a sin wave of 300Hz
- 4) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 5) Select the EKG measurement option (the last one), ensure the button is inversed in color and press enter to return to the main menu.
- 6) Click on the display menu to go to the display mode
- 7) Observe on the TFT display that the EKG frequency is displayed and between 200Hz and 400Hz
- 8) Change function generator to output a frequency of 600Hz
- 9) Repeat steps 4 – 7 and ensure the newly displayed EKG frequency is between 500Hz and 700Hz

Compute: See SystemControl.Tests.c

Display: The team used the following instructions:

- 1) Add a new line Serial.println(“Running display function”); at the beginning of the display function.

- 2) Compile and upload the System Control and Peripheral System code onto Mega and the Uno respectively
- 3) Wire up the System Control and the Peripheral Subsystem
- 4) Open up the Serial Monitor on the Arduino Mega
- 5) Ensure that no message “Running display function” is being printed (display task is not being run)
- 6) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 7) Select all 4 measurement options (bp, pr, rr, temp), ensure each of the button are inversed in color, and press enter to return to Main Menu.
- 8) Perform the necessary measurements (blood pressure) on the peripheral subsystem
- 9) Select the display option to navigate to the display menu
- 10) On the serial monitor, should now see “Running display function” being printed every 5 seconds.
- 11) Additionally, on the serial monitor should see the following values being printed (Note the number may vary due to the measurements taken):

“Temperature:

60 C

Systolic Pressure:

80 mm HG

Diastolic Pressure:

70 mm HG

Pulse Rate:

75 BPM

Battery:

200”

- 12) Verify that the screen refreshes every 5 seconds
- 13) Press exit to return to the main menu
- 14) Ensure that no more “Running display function” is being printed post return.
- 15) Press annunciate button to navigate to the Annunciate page
- 16) Verify that the “Running display function” is again being printed on 5 second intervals
- 17) Verify the following text is being printed on the TFT display:

“Temperature:

75 C

Systolic Pressure:

80 mm HG

Diastolic Pressure:

70 mm HG

Pulse Rate:

75 BPM

Battery:

199”

- 18) Press the “Acknowledge” button to ack the alarms

19) Ensure all the red text above turn yellow

20) Additionally, verify all yellow (warning) text are flashing at the valid intervals

Keypad: The team used the following instructions: (Every time a button is press/display mode changes, verify against the general testing steps at the beginning of this section)

- 1) Modify SystemControl.ino::sendMessage to print all messages to Serial in addition to Serial1
- 2) Compile and upload the System Control and Peripheral Subsystem code onto the Arduino Mega and Uno respectively
- 3) Wire up the System Control and the Peripheral Subsystem using Serial 1.
- 4) Open Serial Monitor on both the Arduino Mega/Uno
- 5) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 6) Select all 5 measurement options (bp, pr, rr, temp, ekg), ensure each of the button are inversed in color, and press enter to return to Main Menu.
- 7) Observe on the Mega Serial Monitor for message S|0|248|E~ to be sent to the Arduino Uno
- 8) Repeat steps 4~7 with the following buttons pressed and the expected output to message:
only bp -> S|0|32|E~
only pr -> S|0|64|E~
only temp -> S|0|16|E~
only rr -> S|0|128|E~
only ekg -> S|0|8|E~Press on the display button to navigate to display mode
- 9) Press on Annunciate to navigate to annunciate mode
- 10) Press exit to return to main menu
- 11) Press display button to navigate to display mode
- 12) Press exit button to return to main menu

Status: See SystemControl.Tests.c

Communications: The communications between System Control and Peripheral System for measurement has already been tested in the measurement caller/callback.

- 1) Modify SystemControl.ino::sendMessage to print all messages to Serial in addition to Serial1
- 2) Compile and upload the System Control and Peripheral Subsystem code onto the Arduino Mega and Uno respectively
- 3) Wire up the System Control and the Peripheral Subsystem using Serial 1.
- 4) Open Serial Monitor on both the Arduino Mega/Uno
- 5) From the main menu, click on the “Menu Selection” button to navigate to the Menu Selection mode.
- 6) Select all 5 measurement options (bp, pr, rr, temp, ekg), ensure each of the button are inversed in color, and press enter to return to Main Menu.
- 7) Perform the valid measurements on the Peripheral Subsystem, ensure that at least 1 value is extreme out of range
- 8) Ensure the following request is sent to the Peripheral Subsystem: S|1|0|E~
- 9) Verify that a warning LED on the peripheral subsystem turns on.

Remote communications: Since remote communications is not a standalone task and is a means for which the command task and remote display tasks, it would be hard to test independently and will be tested as part of the below functionalities

Command: The team used the following instructions:

- 1) Compile and upload the System Control and Peripheral Subsystem code onto the Arduino Mega and Uno respectively
- 2) Wire up the System Control and the Peripheral Subsystem using Serial 1.
- 3) Open PuTTY and connect to the COM port connecting to the Arduino Mega
NOTE: need to force on local echo in the terminal settings in order to see user typing as well
- 4) Type "I|Doctor, Patient|E~" and press enter
- 5) Wait and should expect the Putty Screen to refresh with the new doctor and patient name:
"Medical Monitoring System
Doctor
Patient"
- 6) Type "D~" and press enter
- 7) Expect to see the TFT display turn black
- 8) Type "D~" and press enter
- 9) Expect to see the TFT display resume
- 10) Repeat steps 6~10 in the "menu selection", "display", and "annunciate" mode
- 11) Type "P~" and press enter
- 12) Press Menu Selection to enter menu selection mode
- 13) Select any measurement of choice and press enter to return to main menu
- 14) No measurements are expected to be made, any measurement made and responded by the peripheral subsystem should be discarded
- 15) Type "S~" and press enter
- 16) Press Menu Selection to enter menu selection mode
- 17) Select any measurement of choice and press enter to return to main menu
- 18) Expect measurements to resume
- 19) Type "M~" and press enter
- 20) Should expect the following text to be printed on the PuTTY display:
Temp:75,Syst:80,Dias:70,Puls:75,Resp:75,EKG:300
- 21) Type "W~" and press enter
- 22) Should expect the following text to be printed on the PuTTY display:
Temp:75,Syst:80,Dias:70,Puls:75,Resp:75
- 23) Mark test as completed successfully

Remote Display: The team used the following instructions:

- 1) Modify SystemControl.ino::sendMessage to print all messages to Serial in addition to Serial1
- 2) Compile and upload the System Control and Peripheral Subsystem code onto the Arduino Mega and Uno respectively
- 3) Wire up the System Control and the Peripheral Subsystem using Serial 1.
- 4) Open PuTTY and connect to the COM port connecting to the Arduino Mega

NOTE: need to force on local echo in the terminal settings in order to see user typing as well

- 5) Expect the following message to be printed on the PuTTY display

“Medical Monitoring System

Iam Doctor

Iam Patient

Temperature:	75 C
Systolic Pressure:	80 mmHg
Diastolic Pressure:	70 mmHg
Pulse Rate:	75 BPM
Respiration Rate:	75 Breaths per minute
EKG:	300 Hz

- 6) Expect the above measurements to change as more measurements are made, and the above value should replace the existing screen content
- 7) Pass the remote display test case

SUMMARY

In summary, this project focused on adding new measurement capabilities (temperature, blood pressure, and respiration rate) to reinforce the digital/analog input for the medical monitoring system, thus alleviating the dependency on measure model used in previous projects. Additionally, the introduction of EKG measurement brought a hard time constraint on the system for which measurement has to be taken when the timer arose. In addition to local improvements, new functionality, such as that to allow remote communication enables new user scenarios for which the doctor can remote overview the patient’s measurements and perform actions on the system.

First, the group wrote the design specification for the Phase III/IV prototype of the medical device, evaluating its use cases and breaking them down into various functional parts and individual tasks and classes. Then, the group further specified the responsibility and expected behavior of each of these tasks, which then serves as an outline for the implementation of the system, as well as tests to ensure that an implementation of the system works as expected. Afterwards, the system was implemented according to the outline: The System Control portion was uploaded to the Arduino ATmega while the Peripheral Subsystem was uploaded to the Arduino Uno. The system was tested and confirmed to follow the specifications.

Compared to the previous prototypes, this provided much more use for the user as the doctor can easily hook up real sensors (for digital/analog inputs) and read values, although blood pressure is still simulated by a virtual circuit and the computations are not yet flushed out to match real world scenarios. It was designed with the addition of peripheral measurement devices and further development of user interface in mind. The results of this project, both the prototype and the system design, will be extremely useful as development for the medical device continues.

Furthermore, the initial steps have been taken to establish a remote connection and allow for user to remote manage the system and view the measurement status.

CONCLUSION

Overall, the team is satisfied with the outcome of the project. A functioning prototype compliant with the goals and requirements was finished without any major unresolved issues. Although there are small things that could be improved upon, such as the efficiency and organization of the code, the system functions well enough as is. There was also good coordination between the team members in terms of deciding which portion of the code each member should work on and troubleshooting issues that came up together.

However, that is not to say that there was no problems or frustration with the lab. The specifications and requirements for the design was not very well defined. Although this provided a large amount of freedom and decision making for the system, at times it also felt overly difficult to discern what is actually required and what was up to discretion. There were also various parts containing spelling mistakes or inconsistencies that could be improved upon.

GROUP MEMBER CONTRIBUTIONS

Irina Golub	<ul style="list-style-type: none">• Updated 'display' and 'keypad' task, designed 'remote display' and 'EKG capture' task• Assembled required circuitry for some input signals• Helped debug project code• Created use-case, functional decomposition, activity, sequence, state chart, and data flow diagrams.• Wrote rough draft of report sections: Introduction, Design Specification, Software Implementation, Results, and Appendices
Eric Ho	<ul style="list-style-type: none">• Designed and implemented the command tasks• Added functions necessary for communication of EKG data between system control and peripheral systems• Set up register level hardware timers for various tasks• Helped debug and provide solutions various problems• Wrote rough draft of report sections: Abstract, Conclusion
Yifan Shao	<ul style="list-style-type: none">• Updated compute, warning, schedule, and startup function to allow for the new tasks• Introduce EKG processing task using the provided FFT library• Helped debug project code• Worked on the Task/Class diagrams• Wrote rough draft of report sections: Error Analysis, Test Plan, Test Specifications, Test Case, and Summary

I certify that this assignment/report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication.

Signed:

Irina Golub

Yifan Shao

Eric Ho

APPENDIX A: The initial state of each variable

Type unsigned int – the buffers are not initialized

- temperatureRawBuf[8] Declare as an 8 measurement temperature buffer, initial raw variable value 75
- bloodPressRawBuf[16]¹ Declare as a 16 measurement blood pressure buffer, initial raw variable value 80
- pulseRateRawBuf[8] Declare as an 8 measurement pulse rate buffer, initial raw variable value 0
- respirationRateRawBuf[8] Declare as an 8 measurement respiration rate buffer, initial raw variable value 0
- EKGRawBuf[256] Declare a 256 measurement EKG buffer

Type unsigned int – the buffers are not initialized

- tempCorrected Buf[8] Declare as an 8 measurement temperature buffer
- bloodPressCorrectedBuf[16]¹ Declare as a 16 measurement blood pressure buffer
- pulseRateCorrectedBuf[8] Declare as an 8 measurement pulse rate buffer

1. The systolic pressure measurements are to be stored in the first half (positions 0..7) of the blood pressure buffer and the diastolic stored in the second half of the buffer (positions 8..15).

Phase IV addition

Type unsigned int

- EKGFreqBuf[16] Declare a 16 measurement EKG result buffer

Display

Type unsigned int

- tempCorrected Buf[8] Declare as an 8 measurement temperature buffer
- bloodPressCorrectedBuf[16] Declare as a 16 measurement blood pressure buffer
- pulseRateRawBuf[8] Declare as an 8 measurement pulse rate buffer
- respirationRateCorrectedBuf[8] Declare as an 8 measurement respiration rate buffer
- EKGFreqBuf[16] Declare a 16 measurement EKG result buffer

TFT Keypad

Type unsigned short

- Function Select initial value 0
- Measurement Selection initial value 0
- Alarm Acknowledge initial value 0

Remote Display

Type unsigned long

- TempWarningTime initial value 0
- PrWarningTime initial value 0
- RrWarningTime initial value 0
- BpWarningTime initial value 0
- ekgWarningTime initial value 0

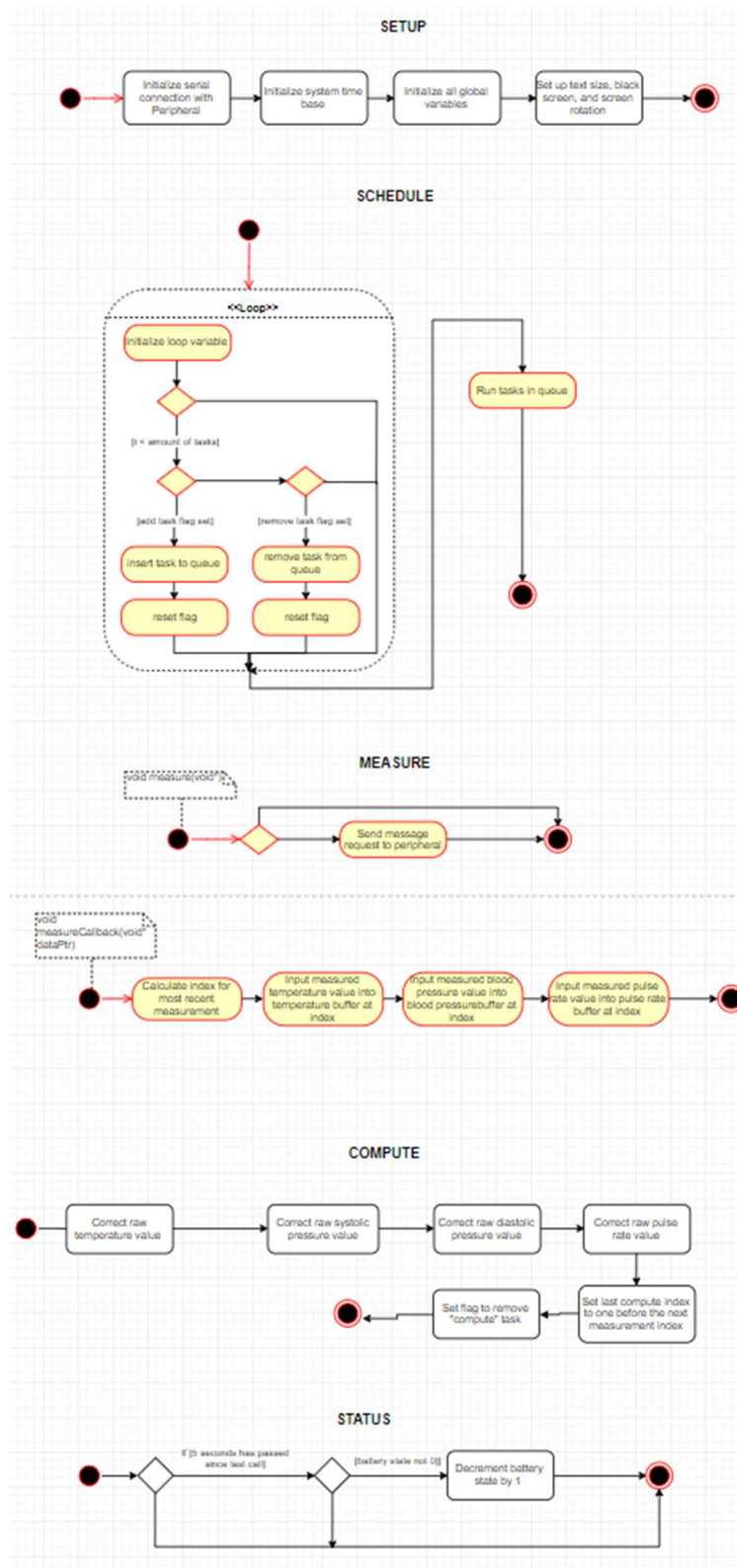
APPENDIX B: Supported Commands and Responses

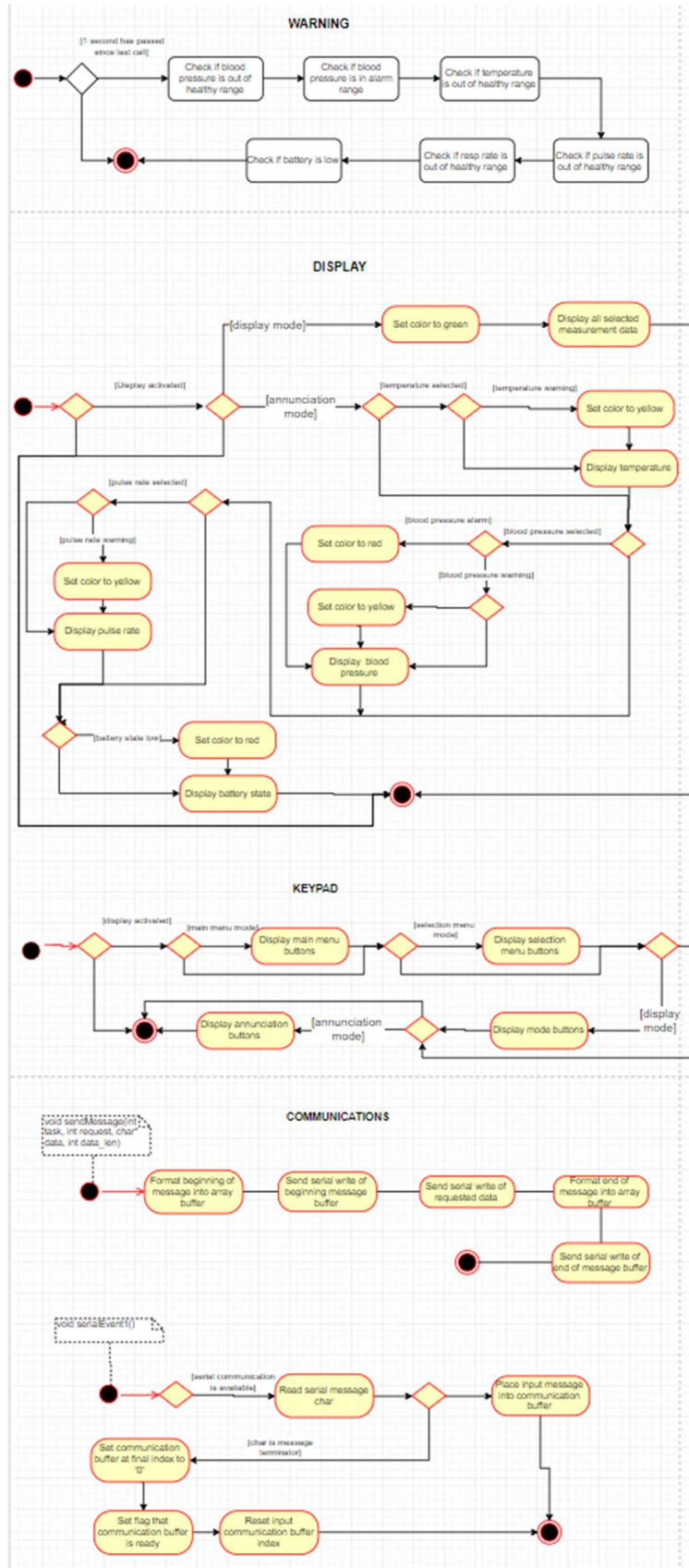
The Commands and Responses for the embedded application task are given as follows.

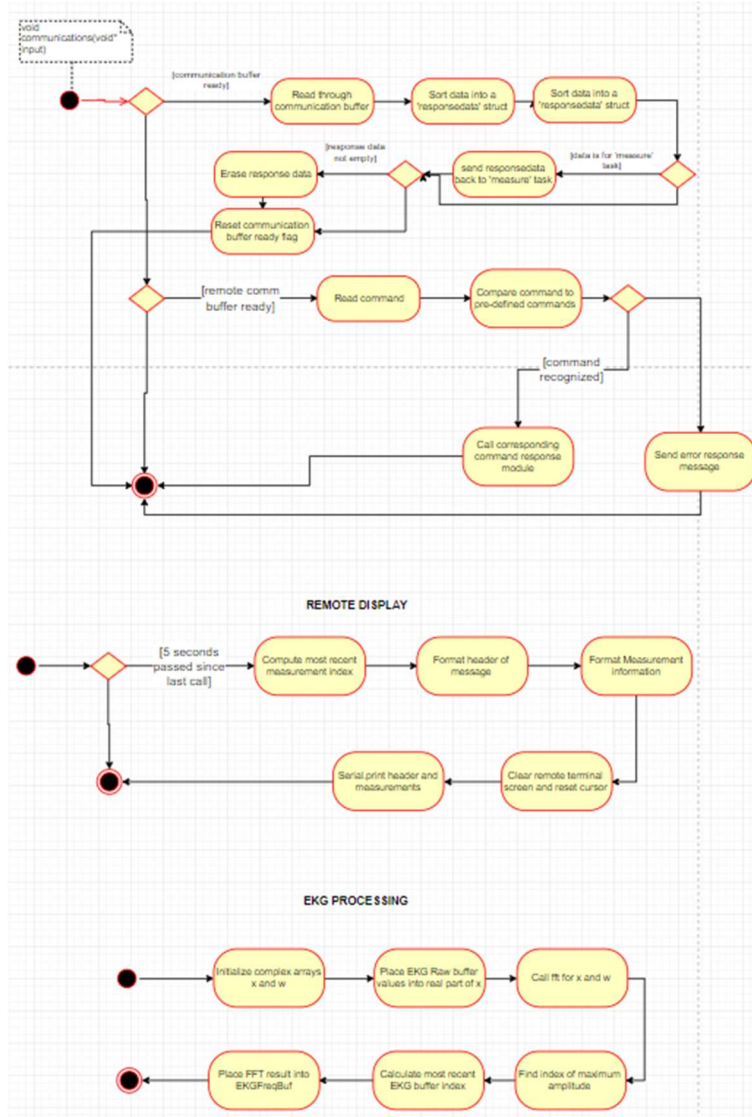
- I** The I command initializes the network communications between your system and the browser page.
- E** The E error response is given for incorrect commands or non-existent commands.
- S** The S command indicates START mode. The command shall start the embedded tasks by directing the hardware to initiate the measurement tasks. In doing so, the command shall also 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 local display.
- M** The M command. The M command requests the return of the most recent value(s) of all measurement data.
- W** The W command. The W command requests the return of the most recent value(s) of all warning and alarm data.

APPENDIX C:

The following activity diagrams show the dynamic behavior of the System Control subsystem:

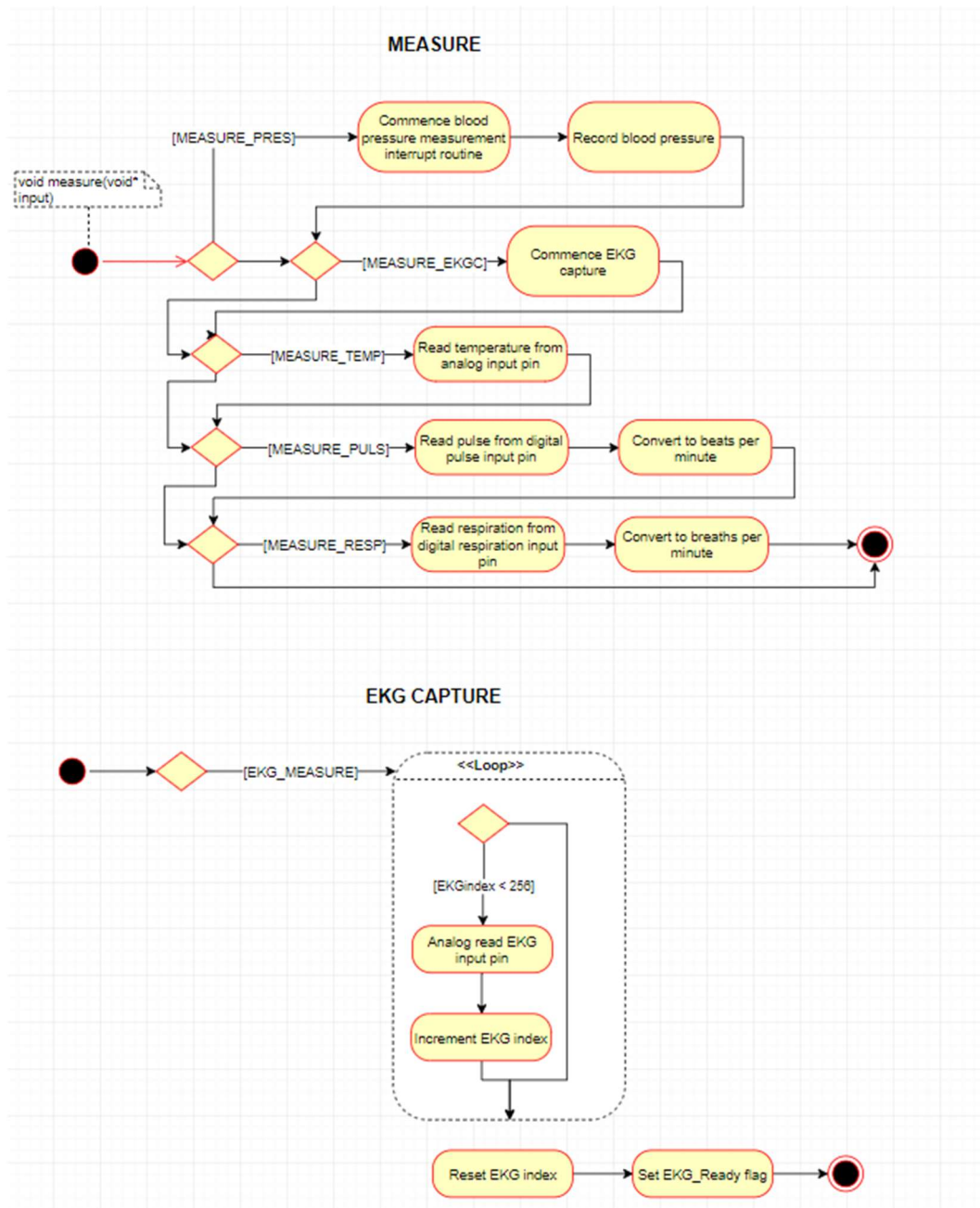




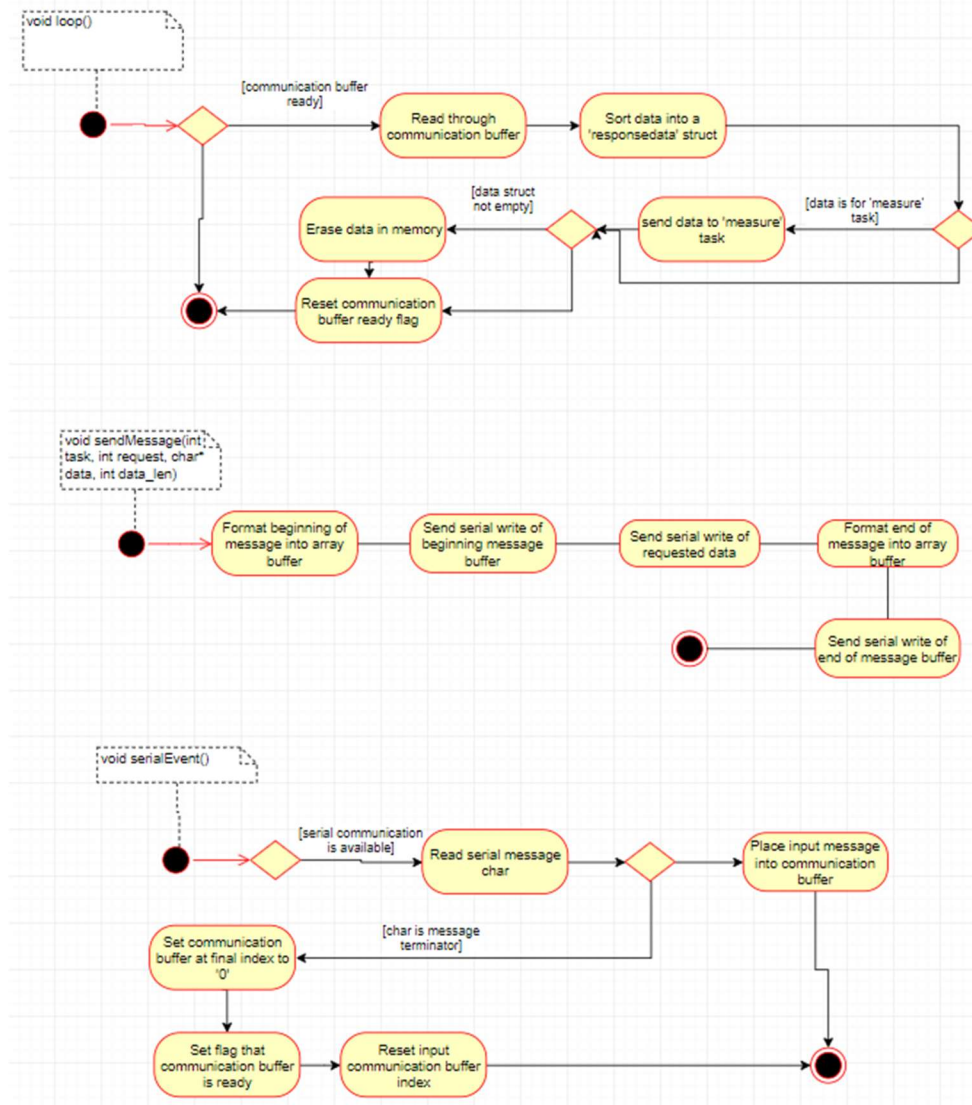


APPENDIX D

The following activity diagrams show the dynamic behavior of the Peripheral subsystem:



COMMUNICATIONS



APPENDIX E

```
// Defines TCB struct
typedef struct TaskStruct {
    void(*taskPtr)(void*);
    void* taskInputPtr;
    struct TaskStruct* nextTask;
    struct TaskStruct* prevTask;
    int id;
-} TCB; //Define this struct as a "TCB"

// Defines structs for TCB Inputs
typedef struct MeasureData {
    unsigned int* temperatureRawBufPtr;
    unsigned int* bloodPressRawBufPtr;
    unsigned int* pulseRateRawBufPtr;
    unsigned int* respirationRateRawBufPtr;
    unsigned short* measurementSelectPtr;
-} MeasureTaskInput;

typedef struct EKGData {
    unsigned int* ekgRawBufPtr;
    unsigned int* ekgFreqBufPtr;
-} EKGTaskInput;

typedef struct ComputeData {
    unsigned int* temperatureRawBufPtr;
    unsigned int* bloodPressRawBufPtr;
    unsigned int* pulseRateRawBufPtr;
    unsigned int* respirationRateRawBufPtr;
    unsigned int* tempCorrectedBufPtr;
    unsigned int* bloodPressCorrectedBufPtr;
    unsigned int* prCorrectedBufPtr;
    unsigned int* respirationRateCorrectedBufPtr;
    unsigned short* measurementSelectPtr;
-} ComputeTaskInput;

typedef struct WarningData {
    unsigned int* temperatureRawBufPtr;
    unsigned int* bloodPressRawBufPtr;
    unsigned int* pulseRateRawBufPtr;
    unsigned int* respirationRateRawBufPtr;
    unsigned int* ekgFreqBufPtr;
    unsigned short* batteryStatePtr;
-} WarningTaskInput;

typedef struct DisplayData {
    unsigned short* displayStatePtr;
    unsigned int* tempCorrectedBufPtr;
    unsigned int* bloodPressCorrectedBufPtr;
    unsigned int* prCorrectedBufPtr;
    unsigned int* respirationRateRawBufPtr;
    unsigned int* ekgFreqBufPtr;
    unsigned short* batteryStatePtr;
-} DisplayTaskInput;

typedef struct StatusData {
    unsigned short* batteryStatePtr;
-} StatusTaskInput;

typedef struct KeypadData {
    unsigned short* displayStatePtr;
    unsigned short* measurementSelectPtr;
    unsigned short* alarmAckPtr;
-} KeypadTaskInput;

typedef struct CommunicationsData {
    unsigned short* measurementSelectPtr;
    unsigned int* tempCorrectedBufPtr;
    unsigned int* bloodPressCorrectedBufPtr;
    unsigned int* prCorrectedBufPtr;
-} CommunicationsTaskInput;
```

```

typedef struct RemoteDisplayData {
    unsigned int* tempCorrectedBufPtr;
    unsigned int* bloodPressCorrectedBufPtr;
    unsigned int* prCorrectedBufPtr;
    unsigned int* respirationRateCorrectedBufPtr;
    unsigned int* ekgFreqBufPtr;
    unsigned long* tempWarningAmtPtr;
    unsigned long* bpWarningAmtPtr;
    unsigned long* prWarningAmtPtr;
    unsigned long* rrWarningAmtPtr;
    unsigned long* ekgWarningAmtPtr;
} RemoteDisplayTaskInput;

```

The structs are instantiated as follows:

```

MeasureTaskInput measureTaskInput = { temperatureRawBuf, bloodPressRawBuf,
pulseRateRawBuf, respirationRateRawBuf, &measurementSelect };
TCB measureTask = { &measure, &measureTaskInput, NULL, NULL, 0 };

```

```

ComputeTaskInput computeTaskInput = { temperatureRawBuf, bloodPressRawBuf,
pulseRateRawBuf, respirationRateRawBuf, tempCorrectedBuf, bloodPressCorrectedBuf,
pulseRateCorrectedBuf, respirationRateCorrectedBuf, &measurementSelect };
TCB computeTask = { &compute, &computeTaskInput, NULL, NULL, 1 };

```

```

DisplayTaskInput displayTaskInput = { &displayState, tempCorrectedBuf,
bloodPressCorrectedBuf, pulseRateCorrectedBuf, respirationRateCorrectedBuf, ekgFreqBuf,
&batteryState };
TCB displayTask = { &display, &displayTaskInput, NULL, NULL, 2 };

```

```

WarningTaskInput warningTaskInput = { temperatureRawBuf, bloodPressRawBuf,
pulseRateRawBuf, respirationRateRawBuf, ekgFreqBuf, &batteryState };
TCB warningTask = { &warning, &warningTaskInput, NULL, NULL, 3 };

```

```

StatusTaskInput statusTaskInput = { &batteryState };

```

```

TCB statusTask = { &status, &statusTaskInput, NULL, NULL, 4 };

```

```

KeypadTaskInput keypadTaskInput = { &displayState, &measurementSelect, &alarmAck };
TCB keypadTask = { &keypad, &keypadTaskInput, NULL, NULL, 5 };

```

```

RemoteDisplayTaskInput remoteDisplayTaskInput = { tempCorrectedBuf,
bloodPressCorrectedBuf, pulseRateCorrectedBuf, respirationRateCorrectedBuf, ekgFreqBuf,
&tempWarningTime, &bpWarningTime, &prWarningTime, &rrWarningTime, &ekgWarningTime
};
TCB remoteDisplayTask = { &remoteDisplaySend, &remoteDisplayTaskInput, NULL, NULL, 8 };
CommunicationsTaskInput communicationsTaskInput = { &measurementSelect,
tempCorrectedBuf, bloodPressCorrectedBuf, pulseRateCorrectedBuf };
TCB communicationsTask = { &communications, &communicationsTaskInput, NULL, NULL, 6 };

```


APPENDIX F: Pseudo Code for the *Startup, Schedule, Measure, Compute, Warning, Display, Keypad, Communication, Command, Remote Display and Status* tasks:

Startup Task

Startup:

- Initialize serial connection speed with peripheral
- Initialize hardware-based system time base
- Initialize all global variables
- Insert the communication, status, warning, keypad, and remote display tasks to the task queue
- Set text size to 2
- Fill screen with black
- Set rotation to 3

Schedule Task

Schedule:

- Check if any tasks need to be added
 - Add those tasks by calling insert_task
 - Reset appropriate flags
- Loop through task queue
 - Call function with the given inputs
- Check if any tasks need to be removed
 - Remove those tasks by calling remove_task
 - Reset appropriate flags

Measure Task

Measure:

- If measurements have been enabled:
 - Send message to peripheral subsystem to request data
 - Set flag to remove measurement task from task queue

MeasureCallback:

- Calculate index to store data at
- Set the measurement buffer at that index to the appropriate response data
- Increment next measure index
- Set flag to add compute task to task queue

GetEKGValues:

- Create request with current index of EKG frequency
- Send request to peripheral

EKG Capture Task

EKG Capture ISR:

- If EKG measurement has been selected
 - If EKG index is less than 256
 - Set EKGRawBuf at EKGindex to be analogRead value of EKG input pin
 - Increment EKG index
 - Else
 - Reset EKG index
 - Unset flag to measure EKG

Set flag that EKG measurement has been taken
StoreEKGValues:
Copy EKG response from response data into buffer
If EKG raw buffer is not yet full:
 request next batch of data
Call EKG processing task

EKG Processing Task

EKG Processing:
Pseudo code for EKG processing here

Compute Task

Compute:
Calculate index to be the last measured index divided by the buffer length
Correct temperature and input to corrected temperature buffer
Correct systolic blood pressure and input to corrected blood pressure buffer
Correct diastolic blood pressure and input to second half of corrected blood pressure buffer
Correct pulse rate and input to corrected pulse rate buffer
Set last compute index variable
Set flag to remove 'compute' task from task queue

Warning Task

Warning:
Check if one second has passed since last execution
Calculate index for most recent raw data
Check if systolic and diastolic blood pressure measurements are out of range
 Update blood pressure warning boolean
Check if systolic blood pressure measurement is in 'alarm' range
 Update 'bpOutOfRange' to "1" if yes, "0" otherwise
Check if temperature measurement is out of range
 Update 'tempHigh' boolean
Check if temperature is in 'alarm' range
 Update 'tempOutOfRange' variable to "1" if yes, "0" otherwise
Check if pulse rate is out of range
 Update 'pulseLow' boolean
Check if pulse rate is in 'alarm' range
 Update 'pulseOutOfRange' variable to "1" if yes, "0" otherwise
Check if respiration rate is out of range
 Update 'rrLow' boolean
Check if respiration rate is in 'alarm' range
 Update 'rrOutOfRange' variable to "1" if yes, "0" otherwise
Check if EKG is out of range
 Update 'ekgHigh' boolean
If there are no alarms, reset the 'alarmAck' variable

If alarm has been acknowledged and 25 seconds has not passed since alarm acknowledgement,
reset all alarm variables

If any alarms exist

Send alarm data to Peripheral

Check if battery has less than 20% charge left

Update 'batteryLow' boolean

Display Task

Display:

If display is turned off

Fill screen with black

Else

If display state is 'display'

Set text size to 1

Print all measured values selected by user in the same color

If display state is 'annunciation'

If user had selected a temperature measurement:

Print the temperature in green if normal, in yellow if warning, red if alarm

Blink at a 2-second rate if warning

If user had selected a blood pressure measurement:

Print the blood pressure in green if normal, yellow if warning, and red if alarm

Blink at a 1-second rate if warning

If user had selected a pulse rate measurement:

Print the pulse rate in green if normal, yellow if warning, red if alarm

Blink at 0.5-second rate if warning

If user had selected a respiration rate measurement:

Print respiration rate in green if normal, yellow if warning, red if alarm

If user had selected an EKG reading:

Print EKG reading in green if normal, yellow if warning

Print the battery state in green if normal, red if low battery

Status Task

Status:

Verify at least 5 seconds has passed since last execution:

Return otherwise

Check if battery state is not zero

Decrement battery state if not zero

Communications Task

Communications:

If there is a message in remote communication buffer

Read command

Compare received command to predefined values and call appropriate CommandCallBack
module

If no matching command found

Format error response message

- Send to remote communication for transmission
- Free command data variable from memory
- Reset 'remote com buffer ready' boolean
- If there is a message in peripheral communication buffer
 - Read and parse message
 - If the message contains measurement results, send message to measureCallBack module
 - If the message contains EKG results, send message to storeEKGValue module
 - Free responseData from memory
- SendRemoteResponse:
 - Format message into a buffer using snprintf
 - Use serial.write to send data to remote terminal
- SendMessage:
 - Format the input data
 - Send message to the other device
- SerialEvent:
 - Check if incoming data is available
 - If so, store incoming data in local buffers
 - If message is complete,
 - Set flag that message is ready
 - Switch to the next local buffer
- Communications:
 - Check if any message is ready
 - If so, extract the requesting task, requested function, and data
 - Parse the data into the appropriate struct
 - Dispatch the appropriate function with the data
- ParseData:
 - Create the appropriate data struct for the function
 - Fill it with data extracted from the message
 - Return the struct

Command Task

- ParseCommandData:
 - If command is 'T'
 - Read doctor name and patient name
- InitializeCommandCallBack:
 - If doctor name and patient name has been entered
 - Save doctor and patient name to be displayed on remote terminal
 - Else
 - Format error response
 - Send to remote comm task for transmission
- StartCommandCallBack:
 - If measurements have not been enabled
 - Enable measurements by setting enableMeasurement variable = 1
 - Else

Format error response "Start-Measurement already started"
 Sent to remote comm task for transmission
 StopCommandCallBack:
 If measurements have been enabled
 Disable measurements by setting enableMeasurement variable = 0
 Else
 Format error response "Stop – Measurement already stopped."
 Send to remote comm task for transmission
 DisplayCommandCallBack:
 Set doRemoteDisplay variable to the inverse of itself
 MeasureCommandCallBack:
 Compute index of most recent measurement (last compute index / buffer length)
 Format Message containing most recent measurement values
 Send to remote comm task for transmission
 WarningCommandCallBack:
 Check through all warning variables
 If warning variable is true, add the corresponding measurement to the formatted message
 Send message to remote comm for transmission

Remote Display Task

RemoteDisplaySend:
 If 5 seconds has passed since last execution time
 Compute index of most recent measurement values (last compute index % buffer length)
 Set up message header with device name, doctor name, and patient name
 Set up formatting of temperature data
 Set up formatting of systolic pressure data
 Set up formatting of diastolic pressure data
 Set up formatting of pulse rate data
 Set up formatting of respiration rate data
 Set up formatting of EKG data
 Clear screen of remote terminal using Esc[2J and Esc[H
 Use serial.print to send message header and formatted data to remote terminal

Keypad Task

Keypad:
 If display state is 'menu', display menu buttons
 If display state is 'selection menu', display selection menu buttons
 If display state is 'annunciation', display annunciation buttons
 MainMenu:
 Set up buttons labeled "Menu", "Display" and "Annunciate"
 Draw the buttons if display state has been recently changed
 Set touchscreen pins
 Check pressure at any point on TFT screen
 If pressure is located where button is, tell the button it is pressed
 Otherwise tell button it is not pressed
 If button is pressed, draw the button inverted

If button is just released, re-draw the button
 If 'menu' is pressed, change display state variable to 2
 If 'display' is pressed, change display state variable to 3
 If 'annunciate' is pressed, change display state variable to 4
 selectionMenu:
 Set up buttons labeled "Temperature", "Pulse Rate", "Resp Rate", "Blood Pressure", "EKG",
 "Enter", and "Exit"
 Draw the buttons if display state has been recently changed
 Set touchscreen pins
 Check pressure at any point on TFT screen
 If pressure is located where button is, tell the button it is pressed
 Otherwise tell button it is not pressed
 If button is pressed, draw the button inverted
 If any measurement selection button is pressed, store the button press in a separate array
 If button is just released, re-draw the button
 If 'enter' is pressed, place button presses in measurementSelect variable and set display state
 variable to 1.
 If 'exit' is pressed, change display state variable to 1
 Display:
 Set up buttons labeled "Annunciate" and "Exit"
 Draw the buttons if display state has been recently changed
 Set touchscreen pins
 Check pressure at any point on TFT screen
 If pressure is located where button is, tell the button it is pressed
 Otherwise tell button it is not pressed
 If button is pressed, draw the button inverted
 If button is just released, re-draw the button
 If 'exit' is pressed, change display state variable to 1
 If 'annunciate' is pressed, change display state variable to 4
 Annunciate:
 Set up buttons labeled "Ack" and "Exit"
 Draw the buttons if display state has been recently changed
 Set touchscreen pins
 Check pressure at any point on TFT screen
 If pressure is located where button is, tell the button it is pressed
 Otherwise tell button it is not pressed
 If button is pressed, draw the button inverted
 If button is just released, re-draw the button
 If 'ack' is pressed, update alarm Ack variable
 If 'exit' is pressed, change display state variable to 1