EE 474 Project 4

Spring 2018

Learning the Development Environment – The Next Step

University of Washington - Department of Electrical Engineering

Blake Hannaford, Joshua Olsen, Tom Cornelius,

James Peckol, Justin Varghese, Justin Reina, Jason Louie, Bobby Davis, Jered Aasheim, George Chang, Anh Nguyen

---

Project Objectives:

This project is the third phase in the development of a low cost, portable, medical monitoring system. In the previous phases of the project, we built a simple kernel and utilized a non-preemptive schedule to manage the selection and execution of the set of tasks comprising our system. We took the first steps towards implementing and incorporating the *Peripheral Subsystem* by moving the measurement tasks from the *System Control Subsystem* to the *Peripheral Subsystem* and replacing one of the modeled / simulated measurement capabilities with the initial design of a driver to support the specified task and developed the initial support for the *Intrasystem Communication Channel*.

We have also taken the first steps toward developing the user interface by extending the TFT keypad display. Finally, we improved the overall flow of control, and enhanced the safety of the system.

In the third phase of the design life cycle for the project, we will now,

1. Add features and capabilities to an existing product.
2. Incorporate several additional simple tasks to our system.
3. Develop drivers for the remaining peripheral devices.
4. Work with a formal communication protocol.
5. Introduce interrupts and ISRs.
6. Amend the requirements and design specifications to reflect the new features.
7. Amend existing UML diagrams to reflect the new features.
8. Utilize other UML diagrams to model new, dynamic capabilities of the system.
9. Continue to improve skills with pointers, the passing of pointers to subroutines, and manipulating them in subroutines.

The final subsystem must be capable collecting data from several different types of sensors, processing the data from those sensors, displaying it locally, and then transmitting it over a local area network to a collection management station.

Prerequisites:

Familiarity with the C language, the Arduino UNO and ATMega 2560 microcomputers and the Arduino IDE development environment. A wee bit of patience.

Background Information:

Did incredibly well on Project 3; tired and anxious to relax. Getting ready to go party in a few weeks….but don't want to go outside with the current temps and the rain.

In this project, we're going to continue to improve on the capabilities in our previous designs…this is the real world and we'll add more features to our system as well. We have to make money selling people things that we first convince them that they need…yes, we'll make modifications to Version 2.0 of our earlier system….and raise the price, of course. We have to support the continually flagging economy.

Relevant chapters from the recommended text: Chapters 5, 8, 9, 11, 12, and 16.

Cautions, Warnings, and Other Musings:

Never ask a guide in the Uffizi who is carrying rare porcelain vases directions to the Ponte Vechio. Never light up if you just purchased and donned paper pants and jacket from a street vendor to meet the Vatican's dress code.

Try to keep your ATMega and UNO boards level to prevent the machine code from collecting in one corner of the memory. This will prevent bits from sticking and causing a memory block. With a memory block, sometimes the UNO or the ATMega system will forget to download and misremember the name of the, you know, *Peripheral Subsystem* processor.

Never try to run your system with the power turned off unless it's solar powered or you have it plugged into a local current bush. Under such circumstances, the results are generally less than satisfying.

Since current is dq/dt, if you are running low on current, raise your ATMega 2560 and UNO boards to about the same level as the USB connections on the PC and use short leads. This has the effect of reducing the dt in the denominator and giving you more current. You could also hold it out the window hoping that the TFT is really a solar panel.

If the IDEs are downloading your binaries too slowly, lower your board so that it is substantially below the USB connection on the PC and put the IDE window at the top of the PC screen. This enables any downloads to get a running start before coming into your board. It will now program much faster. Be careful not to get the download process going too fast, or the code will overshoot the board and land in a pile of bits on the floor. This can be partially mitigated by downloading over a bit bucket. Note that local software stores stock several varieties of bit bucket, so make certain that you get the proper one. These are not reusable, so also please discard properly. Please note that the farther through the project that you are, the larger the bucket that you must have.… You can recycle old bits if necessary, however, watch when you are recycling that you don't get into an endless loop.

Throwing your partially completed but malfunctioning design on the floor, stomping on it, and screaming 'why don't you work you stupid fool' is typically not the most effective debugging technique although it is perhaps one of the more satisfying. The debugging commands, *step into* or *step over*, is referring to your code, not the system you just smashed on the floor. Further, *breakpoint* is referring to a point set in your code to stop the high-level flow through your program to allow more detailed debugging…it's not referencing how many bits you can cram into the ATMega 2560 or UNO processor's memory before you destroy it.

When you are debugging your code, writing it, throwing it away, and rewriting again several dozen times does little to fix what is most likely a design error. Such an approach is not highly recommended, but can keep you entertained for hours….particularly if you can convince your partner to do it.

Sometimes - but only in the most dire of situations – sacrificing small animals to the code elf living in your ATMega 2560 board and smaller critters to your UNO board does occasionally work. However, these critters are not included in your lab kit and must be purchased separately from an outside vendor or you can ask Bill in stores if he has any spares that you can borrow. Also, be aware that most of the time, code elves are not affected by such sacrifices. They simply laugh in your face…bwa ha ha…

Alternately, blaming your lab partner can work for a short time…until everyone finds out that you are really to blame.

Always keep only a single copy of your source code. This ensures that you will always have a maximum amount of disk space available for games, email, and some interesting pictures. If a code eating gremlin or elf happens to destroy your only copy, not to worry, you can always retype and debug it again.

Always make certain that the cables connecting the PC to the ATMega 2560 or UNO board are not twisted or have no knots. If they are twisted or tangled, the compiled instructions might get reversed as they are downloaded into the target and your program will run backwards.

Do not connect the digital outputs to the digital inputs of the ATMega 2560. Doing so has the potential of introducing excess 0's or 1's into the board and causing, like an over inflated child's balloon, small popping sounds leading to potential rupture of the space-time fabric within the Mega and UNO interconnection scheme such that not even Dijkstra or the thumb mode will be able to stop the bit leaks.

In this part of the lab, when you need to upload from the ATMega 2560 system, be certain to turn the cables around.

Always practice safe software engineering…don't leave unused bits laying around the lab or as Scott Adams writes in the Dilbert strip.

©Dilbert by Scott Adams – 26 January 2013

Project:

We will use this Project to continue working with the formal development life cycle of an embedded system. Specifically, we will continue to move inside and expand the system to implement the software modules and supporting hardware (the *how* – the system internal view) that was reflected in the use cases (the *what* – the system external view) of the medical monitoring system. Inter task communication will still be implemented utilizing shared variables. We will continue to work with the ATMega IDE development tool to edit and build the software then download and debug the code in the target environment.

In the initial phase, we modeled many of the subsystems as we focused on the control flow through the system. In the second phase, we started to develop the *Intrasystem Communications* LAN and implement the detailed drivers required for the various subsystems.

Now, in the third phase, as we continue the development of the system, we will….

- ✓ Continue work with the ATMega and UNO GPIO subsystems.
- ✓ Continue work with ATMega and UNO timing functions.
- ✓ Implement and test the new features and capabilities of the system,
- ✓ Amend the formal specifications to reflect the new features.
- ✓ Amend existing UML diagrams to reflect the new features model some of the dynamic aspects of the system.
- ✓ Upgrade the hardware based system time base,
- ✓ Work with interrupts, interrupt service routines, and hardware timing functions,

This Project, Project report, and program are to be done as a team – play nice, share the equipment, and no fighting.

Software Project – Developing Basic Tasks

The economy is growing…some interesting engineering jobs are starting to appear, and you have just been given a once in a lifetime opportunity to join an exciting new start up. Some of the top venture people, working with *CrossLoop, Inc*. recent startup in the Valley have just tracked you down and are considering you for a position as an embedded systems designer on a new medical electronics device that they are funding. They have put together a set of preliminary requirements for a small medical product based on IPhone, Pre, Blackberry, and Google concepts, ideas, and technologies that is intended to serve as a peripheral to the CrossLoop system.

The product, *Doctor at Your Fingertips*, will have the ability to perform many of the essential measurements needed in an emergency situation or to support routine basic measurements of bodily functions that people with chronic medical problems need to make. The collected data can then be sent to a doctor or hospital where it can be analyzed and appropriate actions taken.

The system must be portable, lightweight, inexpensive, and Internet enabled. It must have the ability to make such fundamental measurements as pulse rate, blood pressure, temperature, blood glucose level, perform simple computations such as trending, and log historical data, or track medication regimen and prompt for compliance. It must also issue appropriate alarms when any of the measurements or trends exceeds normal ranges or there is a failure to follow a prescribed medication regimen.

The initial deliverables for the system include the display and alarm portion of the monitoring system as well demonstrated ability to handle pulse rate, blood pressure, and temperature measurements. Other measurements and capabilities will follow in subsequent phases.

You have now successfully delivered an alpha and beta1 working prototypes of that system. Your customer has been sufficiently impressed with the work that your firm has done that they have selected and funded you to continue with the next phase of the project and has now

awarded the development contract for the third stage of the project. Similar to the second phase, the tasks during the current phase include modifications to the design to improve performance as well as to incorporate additional features and capabilities.

We will now add the Phase III features and capabilities are given in the requirements and design specifications that follow.

Phase III Modification

For clarity, the identification of most of the Phase I and Phase II modifications have been removed from this document. If specific information about these modifications is required, refer the appropriate earlier version.

Phase III requirements supersede Phase I and II specifications where ever there may be a conflict.

Phase III Additions

1. The system will support the ability to measure respiration rate.
2. The system will support control of a blood pressure cuff.
3. The system will implement drivers for all peripheral devices.
4. The display subsystem will be extended.
5. Overall system safety and reliability must be improved.

System Requirements Specification

1.0 General Description

A low cost, state of the art medical monitoring and analysis system is to be designed and developed. The following specification elaborates the requirements for enhanced measurement, display, intrasystem communications, and alarm capability.

The measurement subsystem must accept a measurement request from the user, accept inputs from a number of sensors used to collect data from various parts of the human body, perform the measurement, and return the value.

The display and alarm management subsystems must accept the measurement results from the measurement subsystem, present the results of a on the TFT display, and signal a warning if the data falls outside of pre-specified bounds. Some analysis of the data will be performed by other parts of the system to be designed later.

The outputs of the sensors, that are measuring a variety of natural phenomenon, comprise a number of different data types such as integers, strings or high precision numbers. The system must be designed to accommodate each of these different types.

2.0 Medical Monitoring System

Phase III Description Modified

Displayed messages comprise three major categories: annunciation, status, and warning / alarm. Such information is to be presented on the *System Control* TFT display and on lights on the *Peripheral Subsystem* front panel.

Sensor signals are to be continuously monitored against predetermined limits. If such limits are exceeded, a visible indication is to be given and shall continue until acknowledged.

Acknowledgement shall terminate the initial indication but an alternate visible indication shall continue until the aberrant value has returned to its proper range. If the signal value remains out of range for a specified time, the primary annunciation shall recommence.

The local display function will be fully incorporated during this phase.

1.2 Measurement and Display Subsystems

Sensors and signal values will be modeled and implemented as analogue and digital input and output signals and values to or from the *Peripheral Subsystem* GPIO ports as appropriate for each of the respective tasks,

Modeled sensor signals are to be continuously monitored against predetermined limits as required. If such limits are exceeded, a visible indication is to be given and shall continue until acknowledged.

Acknowledgement shall be indicated but a visible indication shall continue until the aberrant value has returned to its proper range. If the signal value remains out of range for a specified time, the original annunciation shall recommence.

Phase III Additions and Modifications

1. The system will support measurement of respiration rate.
2. The system will support control of a blood pressure cuff.
3. The system will support a console keypad for user data entry.
4. Drivers will be developed for the current set of measurement tools.
5. The measurement data will be modeled using analogue and digital signals.
6. Overall system safety and reliability must be improved.

System Inputs

The measurement component of the system in the third prototype must track and support the measurement of the following signals:

Phase III Addition

Measurements

Blood Pressure

Body temperature

Respiration rate

Pulse rate

Keypad Data

Measurement Selection

Alarm Acknowledge

System Outputs

The display component of the system in the third prototype must track and support the display of the following signals and data:

Phase III Addition

Measurements

Blood Pressure

Body temperature

Respiration rate

Pulse rate

The status, alarm, and warning management portion of the system must monitor and annunciate the following signals:

Status

Battery state

Warning

Temperature, blood pressure, respiration rate, or pulse rate in dangerous range

Alarms

Temperature, blood pressure, respiration rate, or pulse rate out of range

1.3 Use Cases

The following use cases express the external view of the system.

Phase III

(To be updated as necessary)

Software Design Specifications

1.0 Software Overview

A state of the art medical monitoring and analysis system is to be designed and developed.

The high-level design is to be implemented as a set of tasks that are executed, in turn, forever.

The display and alarm management subsystem must accept inputs from a number of sensors that can be used to collect data from various parts of the human body and signal a warning if the data falls outside of pre-specified bounds. Some analysis of the data will be performed by other parts of the system to be designed later.

The outputs of the sensors that are measuring a variety of natural phenomenon comprise a number of different data types such as integers, strings or high precision numbers. The system must be designed to accommodate each of these different types.

The prototype will be implemented using an ATMega development board, a UNO development board, and a TFT display. The ATMega microcomputer will perform the high-level system management, control, and display for the system. The UNO microcomputer will provide the measurement, alarm, and other peripheral functions. The peripheral drivers and measurements will be implemented; sensor signals will be modeled in this phase.

The diagram in Figure 1.1 provides a high-level block diagram for the system.

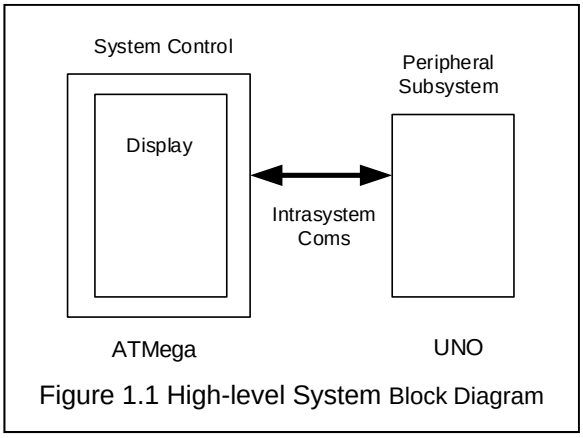


Figure 1.1 High-level System Block Diagram

The following elaborates the specifications for the Phase III measurement, display, communication, and alarm portions of the system.

In addition, you must determine the execution time of each task empirically.

2.0 Functional Decomposition – Task Diagrams

As shown in Figure 1.1 above, the system comprises two top-level blocks. The *System Control* block contains the following functional blocks: *Initialization, Schedule, Measure, Compute, Display, Data Collection, Communication,* and *Status* and additional features for the *Intrasystem Communication.*

Based upon the System Requirements and the Use Case diagrams, a functional decomposition diagram for the *System Control* is given as,

Phase III

(To be updated as necessary)

<u>These functional blocks decompose into the following task/class diagrams,</u>

Phase III

(To be updated as necessary)

The dynamic behaviour of each task is given, as appropriate, in the following activity diagrams:

Phase III

(To be updated as necessary)

The *Peripheral Subsystem* contains the following functional blocks: *Measure, Compute, Annunciate, Status,* and additional capability for the *Intrasystem Communication*.

Based upon the System Requirements and the Use Case diagrams, a functional decomposition diagram for the *Peripheral Subsystem* is given as:

Phase III

(To be updated as necessary)

These functional blocks decompose into the following task/class diagrams,

Phase III

(To be updated as necessary)

The dynamic behaviour of each task is given, as appropriate, in the following activity diagrams:

Phase III

(To be updated as necessary)

2.1 System Software Architecture

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 all have equal priority and will be not be preemptable. Information within the system will be exchanged utilizing a number of shared variables.

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

The *Intrasystem 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.

Phase III Additions and Modifications

To implement the required additions and modifications to the product, the following new capabilities must be incorporated.

a. ***Blood Pressure Cuff Control:*** Include capability to control the inflation and deflation of the blood pressure cuff.
b. ***Extended Blood Pressure Measurement:*** Control the times when measurements are made.
c. ***Respiration Rate:*** Include the ability to measure respiration rate.
d. ***Measurement Process***: The system is to accept and measure sensor inputs as analogue and digital signals. These signals shall be modeled and the results stored in the appropriate buffers.
e. ***Display***: The system will provide the ability to display critical data on a TFT console and more detailed data on a local terminal attached to the system

f. ***Safety***: Overall system safety must be improved.

2.1.1 Tasks and Task Control Blocks

The design and implementation of the *System Control* subsystem will comprise a number of tasks that execute continuously following power ON. Each task will be expressed as a TCB (Task Control Block) structure.

The TCB is implemented as a C struct; there shall be a separate struct for each task.

Each TCB will have four members:

i. The first member is a pointer to a function taking a void* argument and returning a void.
ii. The second member is a pointer to void (void* pointer) used to reference the data for the task.
iii. The third and fourth members are pointers to the next and previous TCBs in a linked list data structure.

Such a structure allows various tasks to be handled using function pointers.

The following gives a C declaration for such a TCB.

```
struct MyStruct
{
    void (*myTask)(void*);
    void* taskDataPtr;
    struct MyStruct* next;
    struct MyStruct* prev;
};
typedef struct MyStruct TCB;
```

The following function prototypes are given for the tasks and defined for the application

Phase III

(To be updated as necessary)

2.1.2 Intertask Data Exchange

Intertask data exchange will be supported through shared variables. All system shared variables will have global scope within the respective microcontroller. Based upon the requirements specification, the following shared variables are defined to hold the measurement data, status, and alarm information in the *System Control* subsystem.

The initial state of each of the variables is specified as follows:

Phase III Addiition

Measurements

Data

Type unsigned int – the buffers are not initialized

- temperatureRawBuf[8] Declare as an 8 measurement

temperature buffer, initial raw value 75

- bloodPressRawBuf[16][1] Declare as a 16 measurement blood pressure buffer, initial raw value 80
- pulseRateRawBuf[8] Declare as an 8 measurement pulse rate buffer, initial raw value 0
- respirationRateRawBuf[8] Declare as an 8 measurement respiration rate buffer, initial raw value 0

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 III Addition

Display

Type unsigned int

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

TFT Keypad

Type unsigned short

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

Phase III Addition

Command Management:

(To be supplied by engineering)

Status

Type unsigned short

- batteryState — initial value 200

Alarms

Type unsigned char

- bpOutOfRange — initial value 0
- tempOutOfRange — initial value 0
- pulseOutOfRange — initial value 0
- rrOutOfRange — initial value 0

Warning

Type Bool[2]

- bpHigh — initial value FALSE
- tempHigh — initial value FALSE
- pulseLow — initial value FALSE
- rrLow — initial value FALSE
- rrHigh — initial value FALSE

2. Although an explicit Boolean type was added to the ANSI standard in March 2000, the compiler we're using doesn't recognize it as an intrinsic or native type. (See http://en.wikipedia.org/wiki/C_programming_language#C99 if interested)

We can emulate the Boolean type as follows:

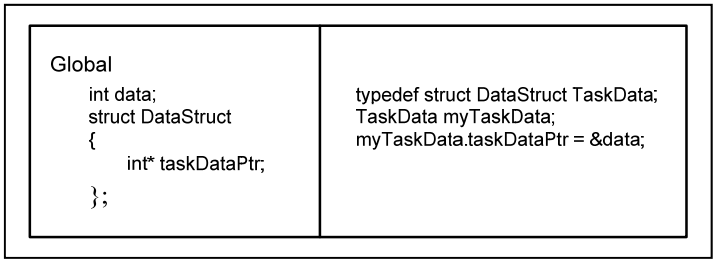
```
enum _myBool { FALSE = 0, TRUE = 1 };

typedef enum _myBool Bool;
```

Put the code snippet in an include file and include it as necessary.

2.1.3 Data Structures

The TCB member, taskDataPtr, will reference a struct containing references to all data utilized by task. Each data struct will contain pointers to data required/modified by the target task as given in the following representative example,



where “data” would be an integer required by myTask.

The data that will be held in the structs associated with each task are given as follows.

MeasureData – Holds pointers to the variables:

- temperatureRawBuf
- bloodPressRawBuf
- pulseRateRawBuf
- measurementSelection
- respirationRateRawBuf

> 3. 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).

ComputeData – Holds pointers to the variables:

- temperatureRawBuf
- bloodPressRawBuf
- pulseRateRawBuf
- respirationRateRawBuf
- tempCorrectedBuf

bloodPressCorrectedBuf
prCorrectedBuf
respirationRateCorrectedBuf
measurementSelection

DisplayData – Holds pointers to the variables:
Mode
tempCorrectedBuf
bloodPressCorrectedBuf
prCorrectedBuf
respirationRateCorrectedBuf
batteryState

WarningAlarmData – Holds pointers to the variables:
temperatureRawBuf
bloodPressRawBuf
pulseRateRawBuf
respirationRateRawBuf
batteryState

Status – Holds pointers to the variables:
batteryState

TFTKeypadData – Holds pointer to the variables:
mode
measurementSelection
alarmAcknowledge

CommunicationsData – Holds pointer to the variables:
tempCorrectedBuf
bloodPressCorrectedBuf
prCorrectedBuf
respirationRateCorrectedBuf

2.1.4 Task Queue

The tasks comprising the application will be held in a task queue. Tasks will be selected from the queue in round robyn fashion and executed. Tasks will not be pre-emptable; each task will run to completion.

If a task has nothing to do, it will exit immediately.

The task queue is implemented as an array of 8 elements that are pointers to variables of type TCB. Seven of the TCB elements in the queue correspond to tasks identified in Section 2.2. The eighth element provides space for future capabilities.

The function pointer of each element should be initialized to point to the proper task. For example, TCB element zero should have its function pointer initialized to point to the *Measure* function.

The data pointer of each TCB should be initialized to point to the proper data structure used by that task. For example, if "*MeasureData*" is the data structure for the *Measure* task, then the data pointer of the TCB should point to *MeasureData*.

2.2 Task Definitions

Phase III *Modify the task definition*

As identified in the functional decomposition in Section 2.0 and system architecture in Section 2.1, the system is decomposed into the major functional blocks: *Initialization, Measure, Compute, Display, Annunciation, Warning and Alarm, Status*, and *Schedule,* and *Communications*.

Such capabilities are implemented in the following class (task) diagrams,

(To be updated as necessary)

The dynamic behaviour of each task is given, as appropriate, in the following activity diagrams

(To be updated as necessary)

2.2.1 Task Functionality

The functionality of each of the tasks is specified as follows:

Phase III Modification

Startup

The *startup* task is not contained in the task queue and is to run one time each time the system is started.

The task shall perform any necessary system initialization, configure and activate the system time base, then suspend itself. The time base will utilize a hardware timer as the basis for scheduling the execution of the remaining tasks (as necessary) every five seconds.

- ✓ Create and initialize all statically scheduled tasks.
- ✓ Enable all necessary interrupts.
- ✓ Start the system.
- ✓ Exit.

The following sequence diagram gives the flow of control algorithm for the system,

(To be updated as necessary)

The static tasks are to be assigned the following priorities:

(To be supplied by engineering)

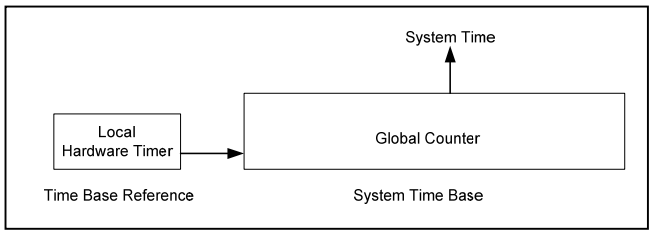
Schedule

The *schedule* task manages the execution order of the tasks in the system.

The task will cause the suspension of all task activity, except for the operation of the warning and error annunciation, for five seconds. The *schedule task* is not held in the task queue.

The following block diagram illustrates the design of the system time base. The Global Counter is incremented every time the Local Delay expires. If the Local Delay is 100 ms, for example, then 10 counts on the Global Counter represent 1 sec.

The Software Delay must be replaced by a Hardware Timer



All tasks have access to the System Time Base and, thus, can utilize it as a reference upon which to base their timing.

Note, all timing in the system must be derived from the System Time Base. The individual tasks cannot implement their own delay functions. Further, the system cannot block for five seconds.

The schedule task will examine all *addTask* type flags and add or remove all flagged tasks to or from the task queue.

The following state chart gives the flow of control algorithm for the system

(To be supplied – by engineering)

Measure

The Measure function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Measure* task's data structure type before it can be dereferenced.

During task execution, only the measurements selected by the user are to be performed.

The various parameters must be modeled and simulated because the necessary sensors are unavailable.

To simulate the parameters, the following operations are to be performed on each of the raw data variables specified in *MeasureData*.

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 *Peripheral Subsystem* must perform the requested operation and return 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* variable.

Phase III Modification

temperatureRaw

Phase III - The following task description is deprecated.

Increment the variable by 2 every even numbered time the function is called and decrement by 1 every odd numbered time the function is called until the value of the variable exceeds 50. The number 0 is considered to be even. Thereafter, reverse the process until the value of the variable falls below 15. Then, once again reverse the process.

The temperature data is to be held in a circular eight reading buffer.

Phase III - The temperature measurement task is modified as follows.

In the final product, temperature data will be collected from an external temperature sensor. For the current prototype, the patient's temperature will be modeled using an analogue signal source then scaling the measured value appropriately to the human temperature range.

The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note this is above or below the previously stored reading.

When the temperature measurements are complete, the scheduler shall be signaled to remove the temperature measurement task from the task queue.

Phase III - The following systolic and diastolic measurement task descriptions are deprecated.

systolicPressRaw

Increment the variable by 3 every even numbered time the function is called and decrement it by 1 every odd numbered time the function is called until the value of the variable exceeds 180. At such time, reverse the process. Decrement the variable by 3 every even numbered time the function is called and increment it by 1 every odd numbered time the function is called until the value of the variable drops below 90. At such time, reverse the process again. The number 0 is considered to be even.

Set a variable indicating that the systolic pressure measurement is complete.

When the diastolic measurement is complete, repeat the process.

diastolicPressRaw

Decrement the variable by 2 every even numbered time the function is called and incremented by 1 every odd numbered time the function is called until the value of the variable drops below 40. At such time, reverse the process. Increment the variable by 2 every even numbered time the function is called and decrement it by 1 every odd numbered time the function is called until the value of the variable exceeds 90. At such time, reverse the process again. The number 0 is considered to be even.

At such time, set a variable indicating that the diastolic pressure measurement is complete.

When the diastolic measurement is complete, repeat the full measurement process.

The systolic and diastolic pressures are to be held in a circular sixteen reading buffer. Positions 0 - 7 hold the systolic pressure measurements and positions 8 -15 hold the corresponding diastolic pressure readings.

Phase III - The systolic and diastolic measurement tasks are modified as follows:

systolicPressRaw and diastolicPressRaw

The analog signals for the systolic and diastolic pressures are to be read at a fixed time interval of 5 ms following an externally generated signal from the blood pressure cuff. When the signal occurs, the blood pressure measurement *addTask* flag is to be set to signal the scheduler to add the blood pressure measurement tasks to the task queue.

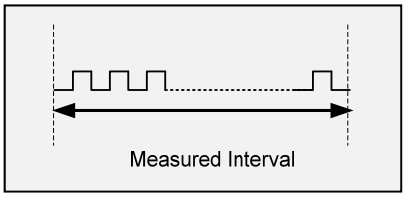
For the current prototype, the cuff is to be inflated and deflated manually using a pushbutton and a switch. The button will increment the pressure by 10% for each press when the switch is in the increment position and will decrement it by 10% for each press when in the decrement position.

The cuff and patient will be modeled by a simple counter. The counter is to be incremented by one step as the cuff is inflated and decremented by one step as the cuff is deflated. To model the range of different systolic and diastolic blood pressures for different patients, the counter output will be compared against preset limits.

At a blood pressure in the range of 110 to 150 mm HG, an interrupt is to be generated signaling that the systolic measurement is to be made. At a blood pressure of 50 to 80 mm HG, an interrupt is to be generated signaling that the diastolic measurement is to be made.

When the blood pressure measurements are complete, the scheduler shall be signaled to remove the pressure measurement tasks from the task queue.

pulseRateRaw



Pulse rate is measured by a pulse rate transducer. The output of the transducer is an analog signal with a range of 0 to + 50 mV DC.

The analog signal from the pulse rate sensor outputs is to be amplified into the range of 0 to + 3.3 V DC and converted into a digital signal that appears as a series of successive pulses. In the final product, such pulses will be detected using an external event interrupt. For the current prototype, they can be

modeled using an internal interrupt or an external interrupt generated by a GPIO signal.

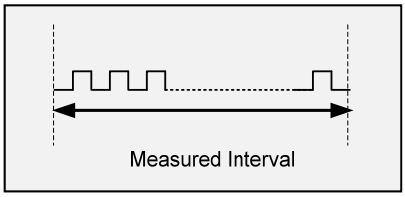
The number of pulses or beats occurring during the measurement interval must be determined and stored in a buffer as a binary value. The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note this is above or below the previously stored reading.

The pulse rate transducers are currently under development. One of the objectives of the present phase is to obtain some field data on a beta version of the transducers. To this end, the upper frequency limit of the incoming signal shall be empirically determined. The upper limit will correspond to 200 beats per minute and the lower to 10 beats per minute.

The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note this is above or below the previously stored reading.

When the task has completed a new set of measurements, the measurement tasks are to be deleted from the task queue the *addTask* flag for the *Compute* task is to be set.

respirationRateRaw



Respiration rate is measured by a respiration rate transducer. The output of the transducer is an analog signal with a range of 0 to + 75 mV DC.

The analog signal from the pulse rate sensor outputs is to be amplified into the range of 0 to + 3.3 V DC and converted into a digital signal that appears as a series of successive pulses. In the final product, such pulses will be detected using an external event interrupt. For the current prototype, they can be modeled using an internal interrupt or an external interrupt generated by a GPIO signal.

The number of pulses or breaths per minute occurring during the measurement interval must be determined and stored in a buffer as a binary value. The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note this is above or below the previously stored reading.

The respiration rate transducers are currently under development. One of the objectives of the present phase is to obtain some field data on a beta version of the transducers. To this end, the upper frequency limit of the incoming signal shall be empirically determined. The upper limit will correspond to 50 breaths per minute and the lower to 10 breaths per minute.

When the task has completed a new set of measurements, the measurement tasks are to be deleted from the task queue the *addTask* flag for the *Compute* task is to be set.

Compute

The *Compute* function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Compute* task's data structure type before it can be dereferenced.

The *Compute* task is to be scheduled only if new data is available from the measurement task. When scheduled, the *Compute* task will take the data acquired by the *Measure* task, perform any necessary transformations or corrections; when the computations are complete, the *Compute* task is to be deleted from the task queue.

The following relationships are defined between the raw data and the converted values.

1. Temperature in Celsius: tempCorrected = 5 + 0.75•tempRaw
2. Systolic Pressure in mm Hg: sysPressCorrected = 9 + 2•systolicRaw
3. Diastolic Pressure in mm Hg: diasPressCorrected = 6 + 1.5•diastolicRaw
4. Pulse Rate in beats per minute: prCorrected = 8 + 3•prRaw
5. Respiration Rate in beats per minute: rrCorrected = 7 + 3•rrRaw

TFTKeypad Task

The *Keypad* function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Keypad* task's data structure type before it can be dereferenced.

The keypad is used to support the user selecting a mode: Measure Menu, Display, or Annunciation then a choice within the selected mode.

The following functions are defined for the keypad:

- Mode Select
  - Measure Menu
  - Display
  - Annunciation
  - Two modes for expansion
- Measure Menu
  - ✓ Blood Pressure
  - ✓ Temperature
  - ✓ Pulse Rate
  - ✓ Respiration Rate
- Display
  - ✓ Present measurement information
- Annunciation
  - ✓ Measurement and Alarm information
- Selection
  - In the Menu mode, select a measurement to be made.
  - In the Display mode, signal that the Display task is to be scheduled
  - In the Annunciation mode, acknowledge an alarm or warning

The keypad shall be scanned for new key presses on a two-second cycle or as needed.

Display

The *Display* function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Display* task's data structure type before it can be dereferenced.

The *Display* task is charged with the following:

- Retrieving and presenting the results from the *Compute* task, formatting the data so that it may be displayed on the instrument display, and presented to the user.
- Retrieving and presenting the alarm and warning information from the *Warning-Alarm* task, formatting the data so that it may be displayed on the instrument display, and presented to the user.
- The responsibility of annunciating the state of the system battery.

The *Display* task will support two modes: Measurement and Annunciation.

In the Measurement mode, the following will be displayed:

Phase III Addition

Measurement

- Blood Pressure
  - Systolic pressure: <systolic pressure> mm Hg
  - Diastolic pressure: <diastolic pressure> mm Hg
- Temperature
  - Temperature: <temperature> C
- Pulse Rate
  - Pulse rate: <pulse rate> BPM>
- Respiration Rate
  - Respiration rate <breaths per minute>

In the Annunciation mode, the following will be displayed

Annunciation

- Measurement and Alarm information

The ASCII encoded Measurement and Alarm information shall be presented on the display in the Annunciation mode. Entries will be displayed in the following order and in the color indicated by the *Warning-Alarm* task.

1. Systolic pressure: <systolic pressure> mm Hg
2. Diastolic pressure: <diastolic pressure> mm Hg
3. Temperature: <temperature> C
4. Pulse rate: <pulse rate> BPM
5. Respiration rate <breaths per minute> RR
6. Battery: <charge remaining>

The display on the *System Control* board should appear as illustrated in the following front panel diagram,

(To be supplied – by engineering)

(To be updated as necessary)

Warning-Alarm

The *Warning-Alarm* function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Warning-Alarm* task's data structure type before it can be dereferenced.

Phase III Addition

The normal range for the measurements is specified as follows:

1. Temperature: 36.1 C to 37.8 C
2. Systolic pressure: 120 to 130 mm Hg
3. Diastolic pressure: 70 to 80 mm Hg
4. Pulse rate: 60 to 100 beats per minute
5. Respiration rate 12 to 25 breaths per minute
6. Battery: Greater than 20% charge remaining

A *normal* value shall be displayed *green* under the following conditions:

1. A measurement is within its specified range.
2. The state of the battery is within specified limits.

A *warning* value shall be displayed *orange* and flash with the specified period if any measurement is more that 5% out of range.

Pulse Rate - Flash with 2 sec period

Temperature - Flash with 1 sec period

Blood Pressure - Flash with 0.5 sec period

An *alarm* value shall be displayed *red* under the following conditions:

1. If the systolic blood pressure measurement is more than 20 percent above or below the specified limit.
2. If the temperature, pulse rate, or respiration rate measurement is more than 15 percent above or below the specified limit.
3. If *Acknowledge* key associated with an annunciation is depressed, the alarm shall display *orange*. If the signal value remains out of range for more than five measurements, the alarm annunciation shall resume.

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

Peripheral Communications Functions

The *Peripheral Interface* supports communication between the *System Control* and the *Peripheral Subsystem*. The supported functions are used by the various tasks to employ the tools supported by the subsystem.

To access and utilize a tool, the task must send a *Request* message to the *Peripheral Subsystem*. The subsystem will accept and interpret the incoming message, preform the requested action and return the results in a *Response* message.

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

A *Request* message must contain the following fields:

1. Start of message
2. End of message
3. Requesting task identifier
4. Function being requested
5. Data required by the function

A *Response* message must contain the following fields:

1. Start of message
2. End of message
3. Requesting task identifier
4. Function being requested
5. Data being returned by the function

Status

The *Status* function shall accept a pointer to void with a return of void.

The pointer in the task argument will be re-cast as a pointer to the *Status* task's data structure type before it can be dereferenced.

The battery state shall be decremented by 1 each time the *Status* task is entered.

Phase III

2.3 Data and Control Flow

The system inputs and outputs and the data and control flow through the system are specified as shown in the following data flow diagram.

(To be supplied by engineering)

2.3 Performance

The execution time of each task is to be determined empirically. (You need to accurately measure it and document your results.)

2.5 General

Once each cycle through the task queue, one of the digital output lines must be toggled.

- All the structures and variables are declared as globals although they must be accessed as locals.

Note: We declare the variables as globals to permit their access at run time.

- The flow of control for the system will be implemented using a construct of the form

```
while(1)
{
    myStuff;
}
```

The program should walk through the queue you defined above and call each of the functions in turn. Be sure to implement your queue so that when it gets to the last element, it wraps back around to the head of the queue.

The following paragraph is now deprecated.

In addition, you will add a timing delay to your loop so that you can associate real time with your annunciation counters. For example, if the loop were to delay 5ms after each task was executed, you would know that it takes 25ms for all tasks to be executed once. We can use this fact to create task counters that implement the proper flashing rate for each of the annunciation indicators. For example, imagine a task that counted to 50 and then started over. If each count lasted 20ms, (due to the previous example) then the task would wait 1 second (50 * 20ms) between events.

To accomplish this, we create the function: "delay_ms(int time_in_ms)". Thereafter, simply call this function with the delay in milliseconds as its argument. Remember Project 1.

3.0 Recommended Design Approach

This project involves designing, developing, and integrating a number of software components. On any such project, the approach one takes can greatly affect the ease at which the project comes together and the quality of the final product. To this end, we strongly encourage you to follow these guidelines:

1. Develop all of your UML diagrams first. This will give you both the static and dynamic structure of the system.
2. Block out the functionality of each module. This analysis should be based upon your use cases.

   This will give you a chance think through how you want each module to work and what you want it to do.
3. Do a preliminary design of the tasks and associated data structures. This will give you a chance to look at the big picture and to think about how you want your design to work before writing any code.

   This analysis should be based upon your UML class/task diagrams.
4. Write the pseudo code for the system and for each of the constituent modules.
5. Develop the high-level flow of control in your system. This analysis should be based upon your activity and sequence diagrams. Then code the top-level structure of your system with the bodies of each module stubbed out.

   This will enable you to verify the flow of control within your system works and that you are able to invoke each of your procedures and have them return the expected results in the expected place.
6. When you are ready to create the project in the Arduino IDE. It is strongly recommended that you follow these steps:
   a. Build your project.
   b. Correct any compile errors and warnings.
   c. Test your code.

d. Repeat steps a-c as necessary.
e. Write your report
f. Demo your project.
g. Go have a beer.

> ***Caution*: Interchanging step g with any other step can significantly affect the successful completion of your design / project.**

Project Report

Write up your Project report following the guideline on the class web page – please check the web site to make certain that you have covered all of the requirements stipulated there.

You are welcomed and encouraged to use any of the example code on the system either directly or as a guide. For any such code you use, you must cite the source**…you will be given a failing mark on the Project if you do not cite your sources in your listing - this is not something to be hand written in after the fact, it must be included in your source code…** This is an easy step that you should get in the habit of doing.

Do not forget to use proper coding style; including proper comments. Please see the coding standard on the class web page under documentation.

Please include in your lab report an estimate of the number of hours you spent working on each of the following:

Design
Test / Debug
Documentation

Please include the items listed below in your project report:

1. Hard copy of your pseudo code
2. Hard copy of your source code.
3. Empirically measured individual task execution time.
4. Include a high-level block diagram with your report.
5. If you were not able to get your design to work, include a contingency section describing the problem you are having, an explanation of possible causes, a discussion of what you did to try to solve the problem, and why such attempts failed.
6. The final report must be signed by team members attesting to the fact that the work contained therein is their own and each must identify which portion(s) of the project she or he worked on.
7. If a stealth submersible sinks, how do they find it?
8. Does a helium filled balloon fall or rise south of the equator?

NOTE: In a formal report, your pseudo code, source, numbers, raw data, etc. should go into an appendix. The body of the report is for the discussion, don't clutter it up with a bunch of other stuff. You can always refer to the information in the appendices, as you need to.

NOTE: If any of the above requirements is not clear, or you have any concerns or questions about you're required to do, please do not hesitate to ask us.

Appendix A: Background on Interrupts

Recall back to the last project, in order to determine when a time delay had passed, we used a function of the form *delayMS()* function. This is a very inefficient use of valuable resources. The *delayMS()* function is a software loop; while we are in the loop, we can do nothing else. We will now use a timer and timer interrupt to do the same function.

An interrupt is *notification that an event has occurred*. What event? Well, the event is truly arbitrary (i.e. the user pushed a button, the timer has reached its max count, a byte has just arrived on the serial port, etc.) and the interrupt simply indicates that the event occurred. The important point to keep in mind is that interrupts allow for *asynchronous* program operation. That is, interrupts allow a program to execute without having to continuously check – poll - for when an event occurs. Rather, a program can dedicate most of the CPU time to other tasks and only when an interrupt occurs will the code to handle this event be triggered. Most often, the phrase *servicing the interrupt* is used to describe the process of executing a piece of code in response to a generated interrupt.

To help clarify how interrupts are used, let us now give an example. Suppose that we have an embedded system used in an automobile that is responsible for a variety of tasks: changing the speed of the automobile, monitoring the status of the engine, responding to user input, managing the fuel injector, etc. From this list of tasks, we know that some are more demanding than others: managing the fuel injector is more CPU intensive than responding to the button that turns the headlights on. Since users are likely to turn the headlights on/off infrequently (i.e. once or twice a day) it makes sense to have this event generate an interrupt. If we setup the headlight-switch to generate an interrupt, we can dedicate most of the running time to handling more important tasks and we don't have to waste valuable CPU cycles polling for this event. In this example, we can dedicate our attention to managing the fuel injector and not waste time polling for when the user pushes the headlight-switch. Hence, it is easy to see how using an interrupt vs. polling is a much more efficient way for handling the headlight-switch.

When an interrupt occurs, how does the program service this event? By using an *interrupt service routine (ISR)*, a program can define code that should be run when an interrupt occurs. Using the example above, we can create an ISR (let us call it *LightSW_ISR( )*) to handle the case when the light-switch interrupt occurs. Furthermore, the *LightSW_ISR( )* function will ***only*** be executed when the light-switch interrupt occurs.

To manage any hardware or software interrupt properly, the following steps are required:

1. Define the interrupt service routine (ISR)
2. Setup the *interrupt vector table* with the address of the ISR

Appendix B: Working with Interrupts

When working with interrupts, it is important that one keep in mind the following information.

1. Interrupts are asynchronous events that can originate in the software or in the hardware; inside or outside of the processor.
2. The interrupt level (essentially the priority) of each interrupt is usually predetermined by the design of the processor or the supporting underware.
3. One must *write an interrupt service routine (ISR)*. Like the body of a function, this routine provides the body of the interrupt. That is, the task that is to be executed when the interrupt occurs. The ISR should be short and concise. It should only contain sufficient code to get a job done or to apprise the system that something needs to be done.

   The following are definite no's when writing an ISR

   - The ISR should not block waiting on some other event.
   - The ISR should not work with semaphores or monitors.
   - The ISR should not contain a number (judgment call here) of calls to functions or perform recursive operations. It's easy to blow the stack.
   - The ISR should not disable interrupts for an extended time.
   - The ISR should not contain dozens of lines of code.
4. One must *store the address of the ISR* in the interrupt vector table.

   The table is essentially an array of function pointers, indexed according to the interrupt number.

   Providing the ISR and getting its address (pointer to that function) into the interrupt vector table varies with different environments.
5. One must *enable the interrupt*.

   This can be done globally – all interrupts are enabled – or locally – interrupt x is enabled. Unless the interrupt is enabled, even if it occurs physically, it will *not* be recognized by the system.
6. One must *acknowledge (or recognize or clear) the interrupt* when it occurs.

   Unless otherwise taken care of by the system (generally not the case). This has the effect of resetting the interrupt and allowing further interrupts of the same kind to recur. If it is not acknowledged, it may occur once then never again. The acknowledgement is usually done in the ISR.
7. One must *exit the interrupt properly*.

   Unlike a simple function call, the return from an interrupt is a bit more involved. Generally there is a specific statement (not a simple return) that is used for exit and cleanup.
8. Look up ATMega 2560 interrupts.

Appendix C: Measuring Blood Pressure

Blood Pressure

A blood pressure measurement is made in two major steps. The process begins with a blood pressure or sphygmomanometer cuff is wrapped around the patient's upper arm. An aural sensing device such as a stethoscope is placed over the brachial artery on the front side of the arm just over the elbow.

The pressure in the cuff is increased to a level of approximately 180mmHg. Such a pressure compresses the brachial artery causing it to collapse and prevent further blood flow. At this point, the pressure in the cuff is slowly decreased. When the artery opens, blood begins to flow again causing vibrations against the artery wall. The pressure at which this occurs is called the systolic pressure and is approximately 120mmHg in the normal case.

As the pressure on the cuff continues to decrease, the blood flow continues to increase. Vibrations against the artery wall also decrease until the blood flow through the artery returns to laminar. The point at which the vibrations cease is called the diastolic pressure. In the normal case, this will be approximately 80 mm Hg.

These sounds of the blood against the artery wall are called Korotkoff sounds after Dr. Nikolai Korotkoff, a Russian physician who first described them.