EE 474 Project 2

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 first phase in the development of a low cost, portable, medical monitoring system. The current phase focuses on the design and development of the basic system architecture, modeling the instrumentation, establishing the high-level control flow, and managing the basic switching operations, display, and alarm and warning annunciation functions. The initial deliverables include the high-level system architecture, the ability to perform a subset of the necessary control, and portions of the display and annunciation components. Subsequent phases will continue to evolve the system architecture and flow of control, extend the driver development, and incorporate additional capabilities into both the measurement and display subsystems.

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.

In developing such a system, we will,

1. Introduce formal specifications
2. Introduce and work with formal design methodologies such as UML - Use Cases, Functional Decomposition, Sequence Diagrams, State Charts, and Data and Control Flow Diagrams.
3. Work with the C language.
4. Develop and build your background on C pointers, the passing of pointers to subroutines, and manipulating them in your subroutines.
5. Introduce simple tasks and a task queue.
6. Share data between tasks.
7. Use software delay/timing functions.
8. Use basic output operations.

Prerequisites:

Familiarity with C programming, the Arduino UNO and ATMega 2560 microcomputers, and the accompanying Arduino development environment (IDE).

Background

Did well on Project 1, put in at least 300 hours per week. No need to sleep – it's over rated anyway.

Relevant chapters from the text: Chapters 5, 6, 7, 9, and 11.

Cautions and Warnings:

Never try to run your system with the power turned off. Under such circumstances, the results are generally less than satisfying.

Since current is dq/dt (movement of charge per in time), if you are running low on current, raise your ATMega board to about the same level as the USB connection on the PC and use short leads and lower the UNO. This has the effect of reducing the dt in the denominator and giving you more current.

If ATMega is downloading your binaries too slowly, lower your board so that it is substantially below the USB connection on the PC and put the Arduino 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 ATMega board and land in a pile of bits on the floor. This can be partially mitigated by downloading over a bit bucket. You should be able to find these in most good computer supply stores or somewhere on line.

Throwing your completed but malfunctioning design on the floor, stomping on it, and screaming ‘work you stupid fool, work’ is typically not the most effective debugging technique although it is perhaps one of the more satisfying.

When you are debugging you 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 gremlin living in your ATMega board does occasionally work. However, these critters are not included in your kit and must be purchased separately from an outside vendor. Also, be aware that most of the time, code gremlins are not affected by such sacrifices. They simply laugh in your face…bwa ha ha…

Alternately, blaming your partners 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 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 board and connecting the ATMega board to the UNO 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.

Always practice safe software engineering…don’t leave unused bits laying around.

Project:

We are using this project as an introduction into the formal development life cycle of an embedded system – or any system, for that matter. This process generally involves five steps,

- Identify and capture the requirements in a formal requirements specification,
- Put together a formal design specification based upon and quantifying those requirements,
- Do a functional design – identify the major functional blocks that will comprise the system.
- Formulate an architecture then map the functional blocks onto that architecture,
- Design, build, and test a prototype of the system.

As the development proceeds, we may (and often do) iterate through these steps a number of times as we refine the design and work through bugs and design or specification issues.

In this project, we are providing significantly simplified versions of the requirements and the specification for the system; such a specification can often be a fairly substantial document in practice. There are several parts that you will have to add as a portion of your contribution to the specification and development effort.

In this project, we will utilize the rapid prototyping approach. Such an approach focuses on the first cut of the functional design and the high-level architecture of the system. We will then verify the operation of that architecture by modeling (simulating) the behaviour of most of the comprising pieces of functionality. As the project evolves, we will replace the modeled behaviour with the actual hardware and software components that affect that behaviour.

As we begin the development, we will….

- ✓ Utilize UML diagrams to model some of the static and dynamic aspects of the system,
- ✓ Identify the major functional blocks,
- ✓ Architect the system hardware as two major subsystems: *System Control Subsystem* and *Peripheral Subsystem*.
- ✓ Architect the system software as a collection of tasks,
- ✓ Develop a simple time based operating system kernel and scheduler that will schedule and dispatch those tasks,
- ✓ Stub out, model, and simulate the desired behaviour of each of the tasks,
- ✓ Begin the design of a driver to control a TFT display,
- ✓ Begin the design of the ATMega *System Control Subsystem* and the UNO based *Peripheral Subsystem*.
- ✓ Implement and test the system,
- ✓ Empirically determine execution time of each such task,
- ✓ Share data using C pointers and data structs.

We will continue to work with the Arduino IDE development tool to edit and build the software then download and debug the executable code in the target environments.

This project, project report, and program are to be done as a team – play nice, share the equipment, keep any viruses (software or otherwise) to yourself, 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 and respiration rate, blood pressure, temperature, EKG level, 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, preliminary development of the peripheral subsystem, as well demonstrated ability to handle pulse rate, blood pressure, and temperature measurements. Other measurements and capabilities will follow in subsequent phases.

All of the sensors that will provide input to the system and any of the peripheral devices with which the system will be able to interact will not be available, consequently, we will model those signals for the first prototype and focus on flow of control

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 the display and alarm portion of the system.

The measurement, display, and alarm management subsystems must accept inputs from a number of sensors 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.

1.1 Monitoring Subsystem

Displayed messages comprise three major categories: annunciation, status and warning / alarm. Such information is to be presented on the local TFT display on the front panel on the ATMega microcomputer.

The local display function will be partially developed during this phase, but will be fully incorporated during follow-on phases.

1.2 Measurement and Display Subsystems

Measurements will be modeled, as appropriate, in each of the respective tasks in the *System Control* portion of the System. Modeled 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 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.

System Inputs

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

- Measurements
  - Blood Pressure
  - Body temperature
  - Pulse rate

System Outputs

The display component of the system must track and support the display of the following signals:

- Display
  - Blood Pressure
  - Body temperature
  - Pulse rate

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

- Status
  - Battery state
- Alarms
  - Temperature, blood pressure, or pulse rate too high
- Warning
  - Temperature, blood pressure, or pulse rate out of range

1.3 Use Cases

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

(To be added – by engineering … this would be you)

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

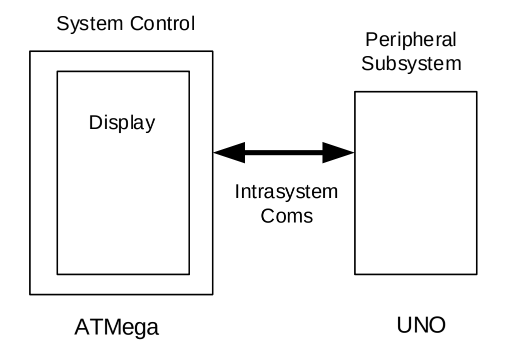


Figure 1.1 High-level System Block Diagram

The following elaborates the specifications for the measurement, display, 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, Display*, and one portion of the *Intrasystem Communication*.

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

(To be supplied by engineering)

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

(To be supplied by engineering)

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

(To be supplied by engineering)

The *Peripheral Subsystem* contains the following functional blocks: *Measure, Compute, Annunciate, Status*, and one portion of the *Intrasystem Communication*.

Based upon the System Requirements and the use case diagrams, a functional decomposition diagram for the *Peripheral Subsystem* is given as,

(To be supplied by engineering)

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

(To be supplied by engineering)

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

(To be supplied by engineering)

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.

2.1.1 Tasks and Task Control Blocks

The design and implementation of the *System Control* subsystem will comprise a number of tasks. 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 two 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.

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;
};
typedef struct MyStruct TCB;
```

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

(To be supplied by engineering)

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:

Measurements

Type unsigned int

• temperatureRaw	initial value 75
• systolicPressRaw	initial value 80
• diastolicPressRaw	initial value 80
• pulseRateRaw	initial value 50

Display

Type unsigned char*

• tempCorrected	initial value NULL
• systolicPressCorrected	initial value NULL
• diastolicPressCorrected	initial value NULL
• pulseRateCorrected	initial value NULL

Status

Type unsigned short

• batteryState	initial value 200

Alarms

Type unsigned char

• bpOutOfRange	initial value 0
• tempOutOfRange	initial value 0
• pulseOutOfRange	initial value 0

Warning

Type Bool[1]

• bpHigh	initial value FALSE
• tempHigh	initial value FALSE
• pulseLow	initial value FALSE

1. 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.

```
Global
    int data;
    struct DataStruct
    {
        int* taskDataPtr;
    };
```

```
typedef struct DataStruct TaskData;
TaskData myTaskData;
myTaskData.taskDataPtr = &data;
```

Figure 2.3 TCB Struct

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

MeasureData – Holds pointers to the variables:

- temperatureRaw
- systolicPressRaw
- diastolicPressRaw
- pulseRateRaw

ComputeData – Holds pointers to the variables:

- temperatureRaw
- systolicPressRaw
- diastolicPressRaw
- pulseRateRaw
- tempCorrected
- sysPressCorrected
- diasCorrected
- prCorrected

DisplayData – Holds pointers to the variables:

- tempCorrected
- sysCorrected
- diasCorrected
- prCorrected
- batteryState

WarningAlarmData – Holds pointers to the variables:

- temperatureRaw

systolicPressRaw
diastolicPressRaw
pulseRateRaw
batteryState

Status – Holds pointers to the variables:
batteryState

Scheduler Data – Holds pointers to the variables:
None

The following data structs are defined for the application,
(To be supplied – by engineering)

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 seven elements that are pointers to variables of type TCB.

Six of the TCB elements in the queue correspond to tasks identified in Section 2.2. The seventh element provides space for future capabilities.

The task function pointer in each TCB 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 *Measure*, then the data pointer of the TCB should point to *MeasureData*.

2.2 Task Definitions

As identified in the functional decomposition in Section 2.0 and system software architecture in Section 2.1, the system is decomposed into the major functional blocks: *Measure*, *Compute*, *Display*, *Annunciate*, *Status*, *Schedule*, and *Communications*. Such capabilities are implemented in the following class (task) diagrams,

(To be supplied – by engineering)

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

(To be supplied – by engineering)

2.2.1 Task Functionality

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

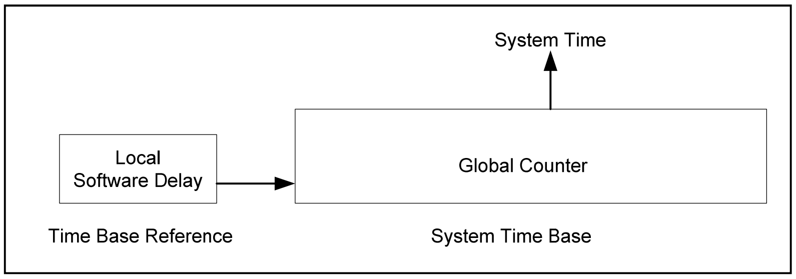
Schedule

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. 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.

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 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.

The various parameters must be 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 referenced in *MeasureData*.

temperatureRaw

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.

systolicPressRaw

Increment the variable by 3 every even numbered time the function is called and decremented by 1 every odd numbered time the function is called until the value of the variable exceeds 100. At such time, set a variable indicating that the systolic pressure measurement is complete. The number 0 is considered to be even.

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

The number 0 is considered to be even.

When the systolic measurement is complete, repeat the process.

pulseRateRaw

Decrement the variable by 1 every even numbered time the function is called and increment by 3 every odd numbered time the function is called until the value of the variable exceeds 40. 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.

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.

When available, the *Compute* task will take the data acquired by the *Measure* task and perform any necessary transformations or corrections.

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:	sysCorrected	= 9 + 2•systolicRaw
3. Diastolic Pressure in mm Hg:	diasCorrected	= 6 + 1.5•diastolicRaw
4. Pulse Rate in beats per minute:	prCorrected	= 8 + 3•bpRaw

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 responsibility of retrieving the results of the *Compute* task, formatting the data so that it may be displayed on the instrument front panel TFT display, and finally presenting the information.

The *Display* task is also charged with the responsibility of annunciating the state of the system battery.

The ASCII encoded information shall be displayed on the TFT display. Blood pressure shall be presented on the top row and temperature, pulse rate, and battery status shall be presented on the second row of the display.

The default color value for all measurements shall be green. The alarm/warning value shall be red.

1.	Temperature:	<temperature> C
2.	Systolic pressure:	<systolic pressure> mm Hg
3.	Diastolic pressure:	<diastolic pressure> mm Hg
4.	Pulse rate:	<pulse rate> BPM>
5.	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)

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.

The normal range for the measurements is specified as follows:

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

The normal state of displayed measurement values shall be green under the following conditions:

1. If a measurement value is within its specified range.
2. If the state of the battery is within specified limits.

The state of a displayed measurement value shall be red to indicate a warning under the following conditions:

1. If the pulse rate measurement is out of range.
2. If the temperature measurement is out of range.
3. If either of the blood pressure measurements is out of range.

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.

2.3 Performance

The execution time of each task is to be determined empirically.

2.4 General

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

- You must determine the execution time of each task empirically – can cleverly toggling a port line help you in this measurement?
- Declare all the structures and variables as globals even though you can not access them as such.

  Note: We declare the variables as globals to permit their access at run time.
- Write a program that will run forever; you can do this with 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.

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.

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 project report an estimate of the number of hours you spent working on each of the following:

Design

Coding

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: 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 120mm Hg 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 80mmHg.

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