EE 474 Project 5

Spring 2018

Learning the Development Environment – The Final 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 final 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 replacing several of the modeled / simulated measurement capabilities with the initial design of the drivers to support the specified tasks and developed the initial support for the *Intrasystem Communication Channel*.

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

The goal of this phase of the project is to continue and to extend our development of the medical monitoring system. To that end, we'll work with some of the other built in capabilities on the UNO and ATMega 2560 microcontrollers.

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

1. Add features and capabilities to an existing product.
2. Amend the formal specifications to reflect the new features.
3. Amend existing UML diagrams to reflect the new features.
4. Incorporate several additional simple tasks to our system.
5. Work with a hard real-time constraint on one of the tasks.
6. Learn and work with some digital signal processing tools.
7. Introduce additional peripheral devices and develop drivers for them.
8. Introduce and manage a formal communication protocol.
9. Incorporate additional safety components into the system.

Prerequisites:

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

Background Information:

Did incredibly well on Project 4; tired and even more 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 dropping.

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

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.

Cautions, Warnings, and Other Musings:

How can a telephone call originating in a small pub in rural Bettyhill (no relation to Benny the olde British comedian), Scotland find its way to a patron's cell phone in the Big Time Ale House in Seattle? How does it know? …and why doesn't it need a visa or a master card at least? …and what if the guy moves at just the last minute? Does the call miss him? Where does the call go if it does?

In the U.S., always look to the left before crossing the street. In the U.K., always look right. In France, look both ways, then, don't cross the street…because they really are trying to kill you. In Italy…well….forget it, you'll never get across the street – don't even try. Viet Nam…that's a story for another day….however, Moses and the Red sea thing was nothing compared to watching 100 cars, 500 motorcycles and scooters, 250 bicycles, and 150 pedestrians go straight through an intersection from all four directions and have subsets making left and right hand turns at the same time and never touch one another. It's pure art.

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 2560 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 connection on the PC and use short leads. NO!!!! don't short the 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. Remember, too, if you run a current loop too long it will eventually be past.

Throwing your 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 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 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 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.

To this end, we will continue the development of a simple kernel and scheduler that will handle a number of new and legacy tasks and support dynamic task creation and deletion. We will continue to work with a dynamic task queue. Each task will continue to share data using pointers and data structs. We will also continue to work with interrupts as the underlying timing mechanism for a time base as well as to signal events from the external world. 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. We will now add a hard real-time task and support for communication over a simple local area network to a remote computer.

As we complete the development of the system, we will….

- ✓ Continue working with interrupts, interrupt service routines, and hardware timing functions,
- ✓ Develop and incorporate additional coronary measurement capabilities.
- ✓ Continue work with the ATMega and UNO GPIO subsystems.
- ✓ Continue work with ATMega and UNO timing functions.
- ✓ Incorporate a remote communications system and network interface into the system,
- ✓ Implement a command and control interface,
- ✓ Extend the warning and alarm subsystem,
- ✓ Improve the safety of the system,
- ✓ Provide support for several optional features,
- ✓ Utilize UML diagrams to model some of the dynamic aspects of the system.

This lab, lab 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 since the start of this project…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 follow in this final phase.

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 at present, so, we will simulate those signals for the first prototype.

You have now successfully delivered an alpha and beta working prototypes of that system and have now been awarded the development contract for the final stage of the project. Similar to the earlier phases, 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 IV features and capabilities are given in the requirements and design specifications that follow.

Phase IV requirements supersede earlier specifications wherever there may be a conflict.

Phase IV Additions

1. The system will incorporate a messaging system and the protocol to support bi-directional communication with an external computer.
2. The system will also incorporate an interface to a local area network (LAN) and will support the display of measurement, status, and alarm information on a remote computer terminal.

3. The system will accept, interpret, and respond to commands from the remote system. Please see Appendix A.
4. Control of the blood pressure cuff and initiation of the associated BP measurements will be completed.
5. The system will incorporate EKG measurement capability as a hard real-time task.
6. The system will support the processing of raw EKG data.
7. EKG measured data will be displayed on the local TFT, the local console, and sent over the network to the remote system.
8. The system will facilitate access to an emergency site in the event that the path is blocked.
9. The overall system safety will 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 display and alarm management subsystem 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.

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.

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

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 a series of 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 and remote display function and remote command I/O 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 IV Additions

1. The design and implementation of the interface to a LAN and remote computer will be completed.
2. Control of and access to the monitoring system from a remote computer will be incorporated.
3. Control of the blood pressure cuff and initiation of the associated BP measurements will be completed.
4. EKG measured data will be displayed on the local TFT, the console, and sent over the network to the remote system.
5. The system will support a console keypad for user data entry.
6. The system will facilitate access to an emergency site in the event that the path is blocked.
7. The overall system safety will be improved.

System Inputs

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

Measurements

Blood Pressure

Body temperature

Respiration rate

Pulse rate

Phase IV Addition

EKG signals

Keypad Data

Measurement Selection

Alarm Acknowledge

System Outputs

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

Measurements

Blood Pressure

Body temperature

Respiration rate

Pulse rate

Phase IV Addition

EKG status

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, EKG, or pulse rate in dangerous range.

Alarms

Temperature, blood pressure, respiration rate, EKG, or pulse rate out o.

1.3 Use Cases

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

Phase IV

(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.

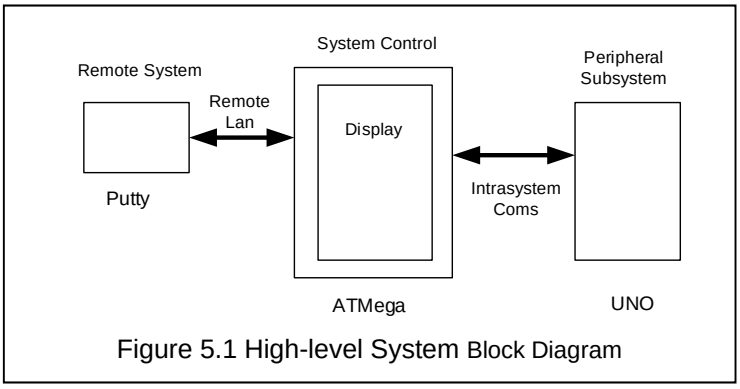


Figure 5.1 High-level System Block Diagram

The following elaborates the specifications for the Phase IV 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*, *Compute*, *Display*, *Data Collection*, *Communication*, and *Status* 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,

Phase IV

(To be updated as necessary)

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

Phase IV

(To be updated as necessary)

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

Phase IV

(To be updated as necessary)

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:

Phase IV

(To be updated as necessary)

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

Phase IV

(To be updated as necessary)

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

Phase IV

(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 *Remote Communications* link is a bidirectional net designed to provide remote command and control access to a medical monitoring system.

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 IV Additions and Modifications

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

a. ***Communications Support***: The system is to provide data to and receive commands from a remote computer via a local area network connection.

b. ***Remote Command and Control***: The system must support the display of measurement data and warning and alarm information on and accept commands from a remote computer.
c. ***Command Management***: The task will receive and interpret commands from the *Remote Communications* task then direct the local subsystem(s) to perform the requested tasks.

   It will format any requested data to be sent by the *Remote Communications* task over the network to the remote system.
d. ***EKG Measurement –*** The system must provide the ability to make EKG measurements on the patient. Eventually, the design must support 12 lead capability; however, for this prototype, a single channel will suffice to prove the ability to make such time critical measurements. To that end, the system must collect a set of isochronously spaced samples of a sinusoidal analogue input signal, convert the them to digital form, store the samples for further processing, then signal when the collection is complete.
e. ***EKG Processing*** – The system must convert the sampled data from the time domain to the frequency domain, and then send the component frequency values to each of the display channels. The frequency content of such signals often contains tell tale spectra that can foreshadow potential problems like heart failure.
f. ***Traffic Management System***: *Optional task* that will provide the system with the capability to clear a path for the mobile medical unit as it tries to get through any traffic blocking its path to an emergency situation. It will first release a blast of phasor fire then follow with a barrage photon torpedoes thereby effectively clearing the path.
g. ***Safety***: Overall system safety must be improved.

2.1.1 Tasks and Task Control Blocks

The medical monitoring and analysis system is to be designed as a collection of tasks that execute continuously following power ON. 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 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 used to reference the data for the task.
iii. The third and fourth members are pointers to the next and previous TCB 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.

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

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

Phase IV

(To be updated as necessary)

2.1.2 Intertask Data Exchange and Control Data

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

Phase IV addition

Data

Type unsigned int – the buffers are not initialized

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

Type unsigned int – the buffers are not initialized

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

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

Phase IV addition

Type unsigned int

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

Display

Type unsigned int

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

TFT Keypad

Type unsigned short

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

Phase IV Addition

Remote Communication:

(To be supplied by engineering)

Command Management:

(To be supplied by engineering)

Traffic 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
- EKGOutOfRange 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
- EKGLow initial value FALSE
- EKGHigh 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:

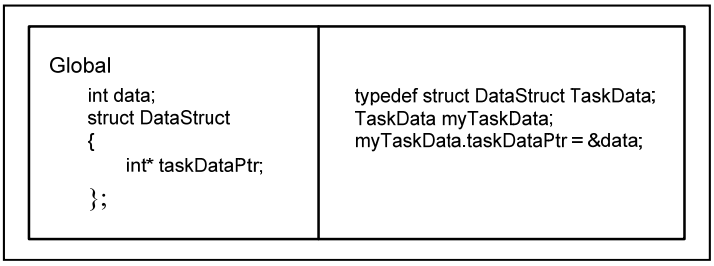
```
enum _myBool { FALSE = 0, TRUE = 1 };

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[3]
- pulseRateRawBuf
- respirationRateRawBuf
- measurementSelection

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).

Phase IV addition

EKGData – Holds pointer to the variables:

- EKGRawBuf
- EKGFreqBuf

ComputeData – Holds pointers to the variables:

- temperatureRawBuf
- bloodPressRawBuf
- pulseRateRawBuf
- respirationRateRawBuf
- EKGRawBuf
- tempCorrectedBuf
- bloodPressCorrectedBuf
- prCorrectedBuf
- EKGFreqBuf
- measurementSelection

DisplayData – Holds pointers to the variables:

- Mode
- tempCorrectedBuf
- bloodPressCorrectedBuf
- prCorrectedBuf
- respirationRateCorrectedBuf
- batteryState

Phase IV Addition

- EKGFreqBuf

WarningAlarmData – Holds pointers to the variables:

- temperatureRawBuf

bloodPressRawBuf
pulseRateRawBuf
respirationRateRawBuf
EKGRawBuf
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
EKGRawBuf

Phase IV Additions

CommandManagementData – Holds pointers to the variables:
(To be supplied by engineering)

RemoteCommunicationData – Holds pointers to the variables:
(To be supplied by engineering)

TrafficManagementData – Holds pointers to the variables:
(To be supplied by engineering)

The following data structs are defined for the application,
Phase IV
(To be updated as necessary)

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 12 elements that are pointers to variables of type TCB. Eleven of the TCB elements in the queue correspond to tasks identified in Section 2.2. The twelfth 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 IV *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, EKG Capture, EKG Processing, Compute, Display, Annunciation, Warning and Alarm, Status, Remote Communications, Traffic Management,* and *Schedule*.

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

(To be updated as necessary)

Phase IV

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:

Startup

The *startup* task shall run one time at startup and is to be the first task to run. It shall not be part of the task queue.

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)

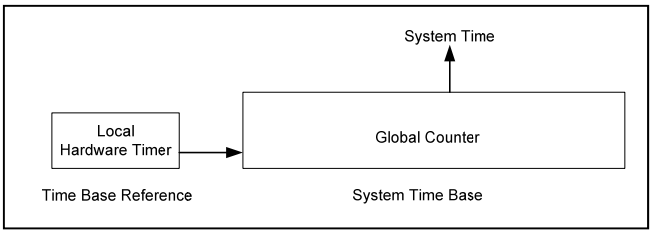
Schedule

Phase IV Modification

The *schedule* task manages the priority based execution order and period of the tasks in the system. However, the task is not in the task queue. The range of priorities spans 1..5, highest to lowest.

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.



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 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 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* 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.

temperatureRaw

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

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.

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.

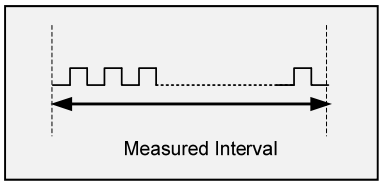
For the current prototype, the cuff is to be inflated and deflated manually using a pushbutton and a switch. The button will increment the pressure by 10% for each press when the switch is in the increment position and will decrement it by 10% for each press when in the decrement position.

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.

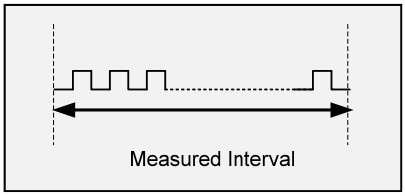
The number of pulses or beats occurring during the measurement interval must be determined and stored in a buffer as a binary value. The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note this is above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

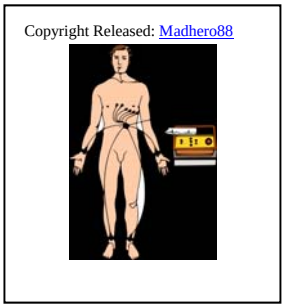
The number of pulses or breaths per minute occurring during the measurement interval must be determined and stored in a buffer as a binary value. The measured values must be stored in a circular 8 reading buffer if the current reading is more than 15% different from the previously stored reading – note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

Phase IV additions

EKG Capture



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

In the implementation of the task, this pointer will be re-cast as a pointer to the *EKG Capture* task's data structure type before it can be dereferenced.

The EKG capture task will accept the output of an analogue scanner used to collect EKG signals from a patient. The design must eventually support 12 lead measurements.

To demonstrate feasibility of the concept, the task must accept a time varying sinusoidal analogue data signal from a single strategically placed transducer, isochronously collect 256 samples, convert the sampled values to digital form, then store the converted samples in a 256 measurement buffer for further processing. When a capture is complete, the task must signal the *EKG Processing* task.

This task is scheduled on demand.

Normally, the time constraints on the FFT, dictate that the driver be written at the register level rather than invoked through a wrapper function.

Because we will not have the external signal available to us, as we did with pulse rate and temperature, we will model the signal and the sampling operation. To do so, begin with a sine wave with a peak amplitude of 3.0 and compute 256 temporally equally spaced

samples with inter sample temporal spacing to model a sampling rate at two and a half to three times the maximum specified frequency.

When the task has completed an EKG measurement, the EKG task are to be deleted from the task queue the *addTask* flag for the *Compute* task is to be set.

EKG Processing

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

In the implementation of the task, this pointer will be re-cast as a pointer to the *EKG Processing* task's data structure type before it can be dereferenced.

The EKG processing task must perform an FFT on the EKG samples collected by the capture task. When the FFT is complete, the task must indicate that new data is available to permit local display and transmission to a remote site for analysis. The frequency content of signals often contains telltale spectra that can foreshadow potential problems such as death.

Signal frequencies shall be used to classify the input signal.

The signal characteristics and frequency ranges are specified as follows:

- EKG data: 35 Hz ≤ Frequency ≤ 3.75K Hz
- Peak Amplitude: 3.3 volts
- Signal starting phase Variable
- Signal: Sinusoid

The 16 most recent values shall be tagged (to identify the EKG) and retained in a buffer for transmission on command from the main office.

This task is scheduled on demand.

You'll find an integer algorithm developed by Brent Plump in the Project 5 folder.

The sampled data shall be converted to the frequency domain using an FFT algorithm (*fftHelp.doc* Appendix C) provided by *Bwang Software, Ltd.*.

Compute

The *Compute* task is to be scheduled only if new data is available from the measurement task.

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

In the implementation of the task, this pointer 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•bpRaw
5. Respiration Rate in beats per minute: rrCorrected = 7 + 3•bpRaw
6. EKG EKGRawBuf = see Appendix A

When the *Compute* task has completed a computation is is to be deleted from the task queue.

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:

1. Mode Select
   - Measure Menu
   - Display
   - Annunciation
   - Two modes for expansion
2. Menu
   - ✓ Blood Pressure
   - ✓ Temperature
   - ✓ Pulse Rate
   - ✓ Respiration
   - ✓ EKG
3. Display
   - ✓ Present measurement information
4. Annunciation
   - ✓ Measurement and Alarm information
5. 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 responsibility of retrieving the results of the *Compute* task, formatting the data so that it may be displayed on the instrument front panel display, and presenting the information to the user.

The *Display* task is charged with the following:

- Retrieving the results of the latest EKG measurement from the *EKG Processing* task, formatting the data so that it may be displayed on the instrument front panel display, and finally presenting the information.
- 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.

Phase IV Addition

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

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

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
- EKG
  - EKG value: < ekg> Hz
- 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 TFT display in the Annunciation mode. 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.

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. Respiration rate <breaths per minute>
6. EKG <Hz>
7. Battery: <charge remaining>

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

(To be updated as necessary)

Warning-Alarm

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

In the implementation of the task, this pointer will be re-cast as a pointer to the *Warning-Alarm* task's data structure type before it can be dereferenced.

Phase IV 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. EKG 35 Hz ≤ Frequency ≤ 3.75K Hz
7. 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 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

Phase IV Addition

Command

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

In the implementation of the task, this pointer will be re-cast as a pointer to the c*ommand* task's data structure type before it can be dereferenced.

The task shall be scheduled whenever a command has been received by the system or when an outgoing message must be formatted in preparation for transmission to the remote computer.

Receive

When a command has been received by the system, the task must verify that the received message is valid. If valid, it is acted upon; if invalid, an error response must be sent to the Remote Communications task.

The legal commands and their interpretation are specified in Appendix B.

Transmit

When a message is to be transmitted, the Command task must build the message body. The message body is then sent to the Remote Communications task for transmission.

After the message has been interpreted and verified as correct or an outgoing message has been built and forwarded to the Remote Communications task, the Command task shall be deleted.

Remote Communications

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

In the implementation of the task, this pointer will be re-cast as a pointer to the Remote Communication task's data structure type.

The Remote Communications task shall be started following power ON then:

- Initialize the network interface
- Connect to and configure a local area network (LAN)
- Set up a handler to communicate with a remote system
- Format the data to be displayed and send the formatted data over the network for display on the remote terminal.
- Continually update the displayed data at a 5 second rate.

The corrected patient data presented at the remote site must be expressed as five strings as follows:

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. Respiration rate <pulse rate> BPM
6. EKG < Measured Frequency reading > Hz
7. Battery: <charge remaining>

The measured value must flash whenever a warning occurs. Annotation must also accompany each displayed measurement indicating how many times during an eight hour interval that a new warning has occurred.

The remote display shall also present the following information:

- The name of the product
- The patient's name
- The doctor's name.

The following diagram gives the layout of the remote web page.

(To be supplied – by engineering)

Status

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

In the implementation of the task, this pointer 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 IV

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 updated as necessary)

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.

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

Appendix A: Supported Commands and Responses

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

- I The I command initializes the network communications between your system and the browser page.
- E The E error response is given for incorrect commands or non-existent commands.
- S The S command indicates START mode. The command shall start the embedded tasks by directing the hardware to initiate the measurement tasks. In doing so, the command shall also enable all the interrupts.
- P The P command indicates STOP mode. This command shall stop the embedded tasks by terminating any running measurement tasks. Such an action shall disable any data collecting interrupts.
- D The D command enables or disables the local display.
- M The M command. The M command requests the return of the most recent value(s) of all measurement data.
- W The W command. The W command requests the return of the most recent value(s) of all warning and alarm data.

Appendix B: FFT Helper

A Real - Time Embedded Application

Using the Cortex-M3 Microcontroller

Bin Wang

Issues on A/D configuration for FFT application

1. Nyquist theory: sampling frequency must be greater than twice the signal frequency (in order to recover the original signal without aliasing).

   Fs >2*Fn

Samples from A/D must be equally spaced in time.

FFT generic function

X[k]= $\sum_{n=0}^{N-1} x[n] W_N^{nK}$, N is sequence length.

X[k], X[n] are supposed to be complex.

X[n] has real and imaginary parts. The samples you get from A/D are the real part of X[n]. The imaginary of X[n] are all zero.

2. X[k] has non-zero values of both real and imaginary parts.

   void fft(complex*x, complex*w, unsigned int m)

   Also you need to typedef{int *real, *imag} complex;
3. Here m is the FFT stage, N=2^m. Usually we use 128, 256 as the length of the sequence, then m will be 7,8 correspondingly. The longer the sequence, the more accurate the FFT. But we are constrained by the available memory space and the speed.
4. Here w is computed inside the FFT function, we do not care and we are not supposed to pass things into it.
5. Since the A/D samples the input signal from 0v-5v, you need shift all the samples by 2.5v to make the samples have positive and negative parts before doing FFT.

In Place Computations

Input array and output array share the same piece of memory space.

Note: each time you do the FFT, remember to initialize the input imaginary buffer to zero.

Determine frequency from the index:

f=fs *m_index /N

f: freq of the measured signal
fs: sample freq
m_index: index which corresponds to the maximum magnitude
Mag^2 = real^2 +imag^2
N: sequence length or N=2^m

Note: The range of FFT is [0,2π], and also symmetric about π.

[0,N] is mapped to this range. When you search for the maximum magnitude, half range searching is desired.

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 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.