CPSC 5530/4530 Project 4
Introducing the Project Environment and Working with C – Another Next Step

Seattle University - Departments of Computer Science and Electrical Engineering
James K. Peckol

Project Objectives:

This project is the third phase in the development of a low cost, portable, medical monitoring system. In the previous phases of this project, we have built a simple kernel and utilized a non-preemptive schedule to manage the selection and execution of the set of tasks comprising our system.

In the current phase we will now move those tasks to a real-time operating system (RTOS) called FreeRTOS (because of what we are paying for it - we're really clever with coming up with names) that utilizes a preemptive, priority based scheduler to do that job.

Most processors designed for embedded applications have just such a computing core, which, through software modules called drivers, supports a wide variety of peripheral devices such as timers, digital input and output channels, telecommunications, and analog to digital conversion to support such applications.

In an earlier phase of our project, we have worked with one such built-in device and its driver, the timer. We have also designed several others, one for the display and a second for the keypad, into our 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 Stellaris EKI-LM3S8962 / Cortex-M3 system and we will learn to design, develop, and debug drivers for such devices under an RTOS.

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

1. Introduce a real real time operating system.
2. Add features and capabilities to an existing product.
3. Incorporate several additional simple tasks to our system.
4. Introduce additional peripheral devices and develop drivers for them.
5. Amend the formal specifications to reflect the new features.
6. Amend existing UML diagrams to reflect the new features.
7. Utilize other UML diagrams to model new, dynamic capabilities of the system.
8. Continue to improve skills with pointers, the passing of pointers to subroutines, and manipulating them in subroutines.

Prerequisites:

Familiarity with C programming, the Texas Instruments Stellaris EKI-LM3S8962 implementation of the ARM Cortex-M3 v7M microcomputer, and the IAR Systems Embedded Workbench integrated C / Assembler development environment. A wee bit of patience.

Background Information:

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

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

Real-time Operating System

We are now moving our design to an RTOS – a real-time operating system called FreeRTOS. This is an operating system with an attitude…a pirate operating system …just in time for Seafair next year…r..yeah, it's got rr's…and at least one nasty patch…rr…this ain't freetos, matey…rrr. Please check out the FreeRTOS web site. You can find this and related documentation at….

http://www.freertos.org/

Check out the getting started and advanced information here or directly at:

http://www.freertos.org/FreeRTOS-quick-start-guide.html

http://www.freertos.org/RTOS_ports.html

http://www.freertos.org/a00090.html#TI

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

Cautions, Warnings, and Other Musings:

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

Try to keep your Stellaris board 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 Stellaris system will forget to download.

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, if you are running low on current, raise your Stellaris board to about the same level as the USB connection on the PC and use short leads. This has the effect of reducing the dt in the denominator and giving you more current. You could also hold it out the window hoping that the OLED is really a solar panel.

If the IAR IDE is downloading your binaries too slowly, lower your Stellaris board so that it is substantially below the USB connection on the PC and put the IAR 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 Stellaris board and land in a pile of bits on the floor. This can be partially mitigated by downloading over a bit bucket. Note that local software stores stock several varieties of bit bucket, so make certain that you get the proper one.

These are not reusable, so also please discard properly. Please note that the farther through the project that you are, the larger the bucket that you must have.… You can recycle old bits if necessary, however, watch when you are recycling that you don't get into an endless loop.

Throwing your 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 Stellaris 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 Stellaris board does occasionally work. However, these critters are not included in your lab kit and must be purchased separately from an outside vendor. 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 Stellaris 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.

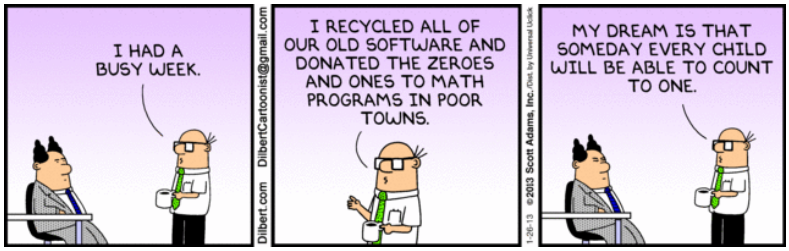
Do not connect the digital outputs to the digital inputs of the Stellaris. Doing so has the potential of introducing excess 0's or 1's into the board and causing, like an over inflated child's balloon, small popping sounds leading to potential rupture of the space-time fabric within the Cortex-M3 interconnection 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 Stellaris 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.

Project:

©Dilbert by Scott Adams – 26 January 2013



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 (the *how* – the system internal view) that was reflected in the use cases (the *what* – the system external view) of the medical monitoring system. To this end, we will migrate the design from the simple kernel and scheduler developed earlier to a real time operating system. Inter task communication will still be implemented utilizing shared variables. We will continue to work with the IAR IDE development tool to edit and build the software then download and debug the code in the target environment.

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

- ✓ Introduce a real time operating system,
- ✓ Modify the startup task to work with the RTOS,
- ✓ Utilize a preemptive schedule,
- ✓ Upgrade the hardware based system time base,
- ✓ Continue work with interrupts, interrupt service routines, and hardware timing functions,
- ✓ Extend the console keypad,
- ✓ Implement and test the new features and capabilities of the system,
- ✓ Amend the formal specifications to reflect the new features.
- ✓ Amend existing UML diagrams to reflect the new features.

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

Software Project – Developing Basic Tasks

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

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

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

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

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 beta1 working prototypes of that system and have now been awarded the development contract for the third stage of the project. Similar to the second phase, the tasks during the current phase include modifications to the design to improve performance as well as to incorporate additional features and capabilities.

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

Phase III Modification

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

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

Phase III Additions

1. The system will incorporate a real-time operating system. Please see Appendix A.
2. The system will extend the keypad in support of user entered data.
3. The system will support control of a blood pressure cuff.
4. Overall system safety and reliability must be improved.

System Requirements Specification

1.0 General Description

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

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. Some analysis of the data will be performed by other parts of the system to be designed later.

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

2.0 Medical Monitoring System

Phase III Description Modified

Displayed messages comprise three major categories: annunciation, status and warning / alarm. Such information is to be presented on a local display, on a series of lights on the front panel, and, on demand, on a local terminal.

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

Acknowledgement shall terminate the aural indication but the visible indication shall continue until the aberrant value has returned to its proper range. If the signal value remains out of range for more than five measurements, the aural annunciation shall resume.

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

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
- Pulse rate

Keypad Data

- Measurement Selection
- Scroll
- Select
- Alarm Acknowledge

System Outputs

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

Measurements

- 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

Phase III Additions and Modifications

1. The system will incorporate a real-time operating system. Please see Appendix A.
2. The system will support control of a blood pressure cuff.
3. The system will support a console keypad for user data entry.
4. Overall system safety and reliability must be improved.

2.1 Use Cases

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

Phase III

(To be updated as necessary)

Software Design Specifications

1.0 Software Overview

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

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

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

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

The prototype will be implemented using the Stellaris development board and the prototype software will control the LED's on the board through the processor's output ports.

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

The following elaborates the specifications for the preliminary measurement, display, and alarm portions of the system.

2.0 Functional Decomposition – Task Diagrams

The system is decomposed into the major functional blocks: *Initialization, Schedule, Measure, Compute, Data Collection, Display, Annunciate, Communication*, and *Status*. Based upon the System Requirements and use case diagrams, a functional decomposition diagram for the system is given as,

Phase III

(To be updated as necessary)

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

Phase III

(To be updated as necessary)

2.1 System Architecture

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. In support of the FreeRTOS Operating System, which will be incorporated during the current phase, each task will be assigned a priority and all tasks will be pre-emptable.

Information within the system will be exchanged utilizing a number of shared variables.

Phase III Additions and Modifications

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

a. ***A Real Time Operating System*** **-** Implemented using the FreeRTOS (RTOS).
b. ***Startup Process***: Modify the task creation and initialization process when the system starts as required by the operating system.
c. ***Blood Pressure Cuff Control:*** Include capability to control the inflation and deflation of the blood pressure cuff.
d. ***Extended Blood Pressure Measurement:*** Control the times when measurements are made.
e. ***Measurement Process***: The system is to accept sensor inputs as analog signals. These signals shall be modeled and the results stored in the appropriate buffers.
f. ***Display***: The system will provide the ability to display critical data on an OLED console and more detailed data on a local terminal attached to the system.
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 monitoring system tasks will be assigned a priority and will be pre-emptable. Information within the system will be exchanged utilizing a number of shared variables.

Phase III Modification

The following TCB description replaces the previous version

TCB creation will be done under the FreeRTOS OS utilizing a task create wrapper function. Details of the function are given in Appendix A.

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

(To be updated as necessary)

The stack size for each of the tasks is given as follows:

(To be supplied by engineering)

The priority for each of the tasks is given as follows:

(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. Based upon the requirements specification, the following shared variables are defined to hold the measurement data, display, status, and alarm information.

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

Measurements

Data

Type unsigned int – the buffers are not initialized

- temperatureRawBuf[8] — Declare as an 8 measurement temperature buffer, initial raw value 75
- bloodPressRawBuf[16][1] — Declare as a 16 measurement blood pressure buffer, initial raw value 80
- pulseRateRawBuf [8] — Declare as an 8 measurement pulse rate buffer, initial raw value 0

Type unsigned char – 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).

Display

Type unsigned char

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

Keypad

Type unsigned short

- Mode — initial value 0
- Measurement Selection — initial value 0
- Scroll — initial value 0
- Select — initial value 0
- Alarm Acknowledge — initial value 0

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[2]

- bpHigh — initial value FALSE
- tempHigh — initial value FALSE
- pulseLow — 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:

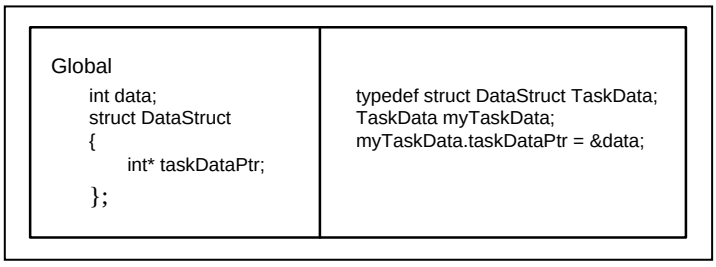
```
enum _myBool { FALSE = 0, TRUE = 1 };

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
- 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).

ComputeData – Holds pointers to the variables:

- temperatureRawBuf
- bloodPressRawBuf
- pulseRateRawBuf
- tempCorrectedBuf
- bloodPressCorrectedBuf
- prCorrectedBuf
- measurementSelection

DisplayData – Holds pointers to the variables:

- mode
- tempCorrectedBuf
- bloodPressCorrectedBuf
- prCorrectedBuf
- batteryState

WarningAlarmData – Holds pointers to the variables:

- temperatureRawBuf
- bloodPressRawBuf
- pulseRateRawBuf
- batteryState

Status – Holds pointers to the variables:

- batteryState

KeypadData – Holds pointer to the variables:

- mode
- scroll
- select
- measurementSelection
- alarmAcknowledge

CommunicationsData – Holds pointer to the variables:

tempCorrectedBuf

bloodPressCorrectedBuf

prCorrectedBuf

The following data structs are defined for the application,

Phase III

(To be updated as necessary)

2.1.4 Task Queue

Phase III Modification

The following task queue description replaces the previous version

Task queue creation will be done under the FreeRTOS OS and will contain seven tasks corresponding to those identified in Section 2.2. Once created, it will not require direct management by the user. Details of the OS and TBCs are given in Appendix A.

Tasks will be pre-emptable; if a task has nothing to do, it will exit immediately.

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

2.2 Task Definitions

Phase III *Modify the task definition*

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

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

(To be updated as necessary)

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

(To be updated as necessary)

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

Phase III Modification

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,

- ✓ Configure and activate the system time base that is to be the reference for timing all warning and alarm durations.

The time base will utilize one of the Cortex-M3's hardware timers as the basis for scheduling the execution of the remaining tasks (as necessary) every five seconds and the warning and alarms as required.

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

(To be updated as necessary)

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

The static tasks are to be assigned the following priorities:

(To be supplied by engineering)

Schedule

Phase III Modification

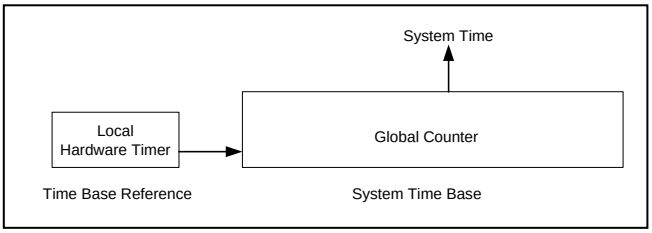
The schedule task is deprecated and replaced by the FreeRTOS scheduler however; the system timebase as well as the major and minor cycle timing must be retained.

The *schedule* task manages the execution order and period of the tasks in the system. However, the task is not in the task queue.

The round robyn task schedule comprises a major cycle and a series of minor cycles. The period of the major cycle is 5 seconds. The duration of the minor cycle is specified by the designer.

Following each cycle major cycle through the task queue, the scheduler will cause the suspension of all task activity, except for the operation of the warning and error annunciation, for five seconds. In between major cycles, there shall be a number of minor cycles to support functionality that must execute on a shorter period.

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

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

(To be supplied – by engineering)

Measure

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

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

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

The various parameters must be 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*.

Phase III Modification

temperatureRaw

Phase III - The following task description is deprecated.

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

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

In the final product, temperature data will be collected from an external temperature sensor. For the current prototype, the patient's temperature will be modeled using an internal analogue signal source.

The Cortex-M3 processor supports an on-chip temperature sensor. The patient temperature will be modeled by utilizing the on-chip sensor then scaling the measured value appropriately to the human temperature range.

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

systolicPressRaw

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

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

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

When the diastolic measurement is complete, repeat the process.

diastolicPressRaw

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

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

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

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

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

systolicPressRaw and diastolicPressRaw

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

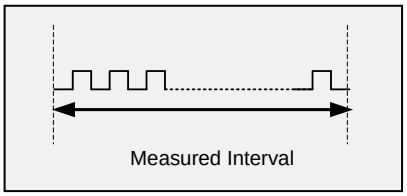
For the current prototype, the cuff is to be inflated and deflated manually using two pushbuttons on the Stellaris board. One button will increment the pressure by 10% for each press and the other will decrement it by 10% for each press.

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.

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

Compute

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

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

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

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

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

Keypad 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 entering information as well as acknowledging alarm signals associated with the current design and in future enhancements to the system.

The following functions are defined for the keypad:

1. Mode Select
   - Menu
   - Annunciation
2. Menu
   - ✓ Blood Pressure
   - ✓ Temperature
   - ✓ Pulse Rate
3. Annunciation
   - Measurement and Alarm information

4. Scrolling and Selection
    - In the menu mode, scroll to menu choice
    - Select menu choice

    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 responsibility of retrieving the results of the latest measurement from each of the measurement tasks, formatting the data so that it may be displayed on the instrument front panel display, and finally presenting the information.

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

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

In the Menu mode, the following will be displayed

Menu

- Blood Pressure
- Temperature
- Pulse Rate

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 OLED 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. Battery: <charge remaining>

The display on the Stellaris board should appear as illustrated in the following front panel diagram,

Phase III

(To be updated as necessary)

Warning-Alarm

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

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

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. Battery: Greater than 20% charge remaining

The *normal* state shall exist under the following conditions:

1. If all measurements are within their specified range, the LED-0 on the front panel shall be illuminated and solid.
2. If the state of the battery is within specified limits.

A *warning* shall be issued under the following conditions:

1. If the pulse rate measurement is out of range, the LED-0 on the front panel shall be illuminated and flashing with a two second period.
2. If the temperature measurement are out of range, the LED-0 on the front panel shall be illuminated and flashing with a 1 second period.
3. If either of the blood pressure measurements is out of range, the LED-1 on the front panel shall be illuminated and flashing with a 0.5 second period.

An *alarm* shall be issued under the following conditions

1. If the systolic blood pressure measurement is more than 20 percent above the specified limit, the LED-2 on the front panel shall be illuminated and solid. The aural alarm shall sound with a series of one second tones.
2. If *Acknowledge* key on the front panel is depressed, the aural alarm shall cease and the LED-2 on the front panel shall flash with a one second period until the systolic blood pressure returns to within 110% of normal.

The state of the battery display, LED-3, shall flash with a one second period when the state of the battery drops below 20% remaining.

Serial Communications

The serial communication task shall accept a pointer to void with a return of void.

In the implementation of the function, this pointer will be re-cast as a pointer to the *serial communications* task's data structure type.

The *serial communications* task shall run whenever a warning occurs.

The *serial communications* task will format the data to be displayed and send that data to the serial port for display on either Hyperterm™. Once finished, it will delete itself from the queue.

The data presented at the remote site must be expressed as 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. Battery: <charge remaining>

Status

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

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

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

Phase III

2.3 Data and Control Flow

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

(To be supplied by engineering)

2.3 Performance

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

2.5 General

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

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

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

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

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

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

The following paragraph is now deprecated.

In addition, you will add a timing delay to your loop so that you can associate real time with your annunciation counters. For example, if the loop were to delay 5ms after each task was executed, you would know that it takes 25ms for all tasks to be executed once. We can use this fact to create task counters that implement the proper flashing rate for each of the annunciation indicators. For example, imagine a task that

counted to 50 and then started over. If each count lasted 20ms, (due to the previous example) then the task would wait 1 second (50 * 20ms) between events.

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

3.0 Recommended Design Approach

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

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

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

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

   This will enable you to verify the flow of control within your system works and that you are able to invoke each of your procedures and have them return the expected results in the expected place.
6. When you are ready to create the project in the IAR IDE. It is strongly recommended that you follow these steps:
   a. Build your project.
   b. Correct any compile errors and warnings.
   c. Test your code.
   d. Repeat steps a-c as necessary.
   e. Write your report
   f. Demo your project.
   g. Go have a beer.

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

Project Report

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

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

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

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

Design

Test / Debug

Documentation

Please include the items listed below in your project report:

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

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

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

Appendix A: Working with an RTOS

We are now moving the design to an RTOS – real-time operating system called, simply enough, FreeRTOS because of what it is and the price we pay for it. This operating system is an open source version of another OS called Micro C / Operating System or µC/OS.

As we move to the OS, we will bring forward all of the tools and techniques that we have studied thus far. You are encouraged to look at the information and tutorials on the FreeRTOS websites:

http://www.freertos.org/

http://www.freertos.org/portlm3sx965.html#SourceCodeOrg

for a lot of very good information.

Let's see how this works. In our work so far, we have created Task Control Blocks (TCBs) of increasing complexity to contain our task's function and data. Under FreeRTOS, we'll do the same thing; however, now, we will use a wrapper function as a tool to build those for us.

Under FreeRTOS, we write,

Tasks

We create a task using the system call,

```
void * TaskCreate (
                    void(taskCode)(void *),
                    char *name,
                    int stackDepth,
                    void *parameters,
                    int priority,
                    taskHandle *createdtaskPtr
                  )
```

taskCode	pointer to the task function the task prototype is of the form void task (void *pd);
name	name of the task
stackDepth	size of the stack allocated for the task
parameters	pointer to the data passed in to the task
priority	task priority 0..5, lowest … highest
createdtaskPtr	handle by which the created task can be referenced

See all this stuff…see we weren't lying to you….this is really how you do it….

Task Execution

The system executes in an infinite loop as we have been doing. We force a context switch using a *vTaskDelay()* statement to give up the CPU to another task.

Main

Your main() with one additional simple task might look like the following,

```
//  task prototypes
void vTask1(void *vParameters);


void main( void ) // this task gets called as soon as we boot up.
{

//  this is the startup task

      //  set up the hardware
            prvSetupHardware();

            //  hardware setup

      //  create the tasks

            xTaskCreate(vTask1, "Task 1", 100, NULL, 1, NULL);

            //  other tasks to be created

            //  remaining code

      //  start the scheduler
            vTaskStartScheduler();

      //  should never get here
            return;
}

void vTask1(void *vParameters)
{
      //  test message to display
        xOLEDMessage xMessage;

        xMessage.pcMessage = "Hello from, Task 1";

      //  all tasks run in an infinite loop
        while(1)
        {
            // Send the message to the OLED gatekeeper for display.
            xQueueSend( xOLEDQueue, &xMessage, 0 );

            //  delay to force a context switch
            vTaskDelay(1000);
        }
}

//  other tasks and code
```

Appendix B: General Purpose I/O – GPIO Overview

The General Purpose I/O module enables the Cortex processor to bring (digital) signals in from external world devices or to send (digital) signals out to external world devices. From the processor's data sheet, the GPIO module is composed of seven physical blocks, each corresponding to an individual GPIO ports (Port A, Port B, Port C, Port D, Port E, Port F, Port G). The module supports from 5 to 42 programmable input/output pins, depending on the peripherals being used.

Details of the GPIO subsystem, specific capabilities, and how the blocks are configured are given in section 8 of the LM3S8962 datasheet.

http://www.ee.washington.edu/class/472/peckol/doc/StellarisDocumentation/Device/Datasheet-LM3S8962.pdf

Read through this section carefully. It is important to note that all GPIO pins are tri-stated by default and that asserting a Power-On-Reset (POR) or RST puts the pins back to their default state. Also look through the blinky, adTest, and gpio_jtag examples in the IAR Workbench and the examples in section 8 of the datasheet.

See also

http://www.ee.washington.edu/class/472/peckol/doc/StellarisDocumentation/Board/8962_EvalBoard.pdf

for the Stellaris Board I/O pinout.

GPIO Pad

The GPIO function interfaces to the external world through what is called a *digital I/O pad*. The pad associated with each port can be independently configured, by the user, based on the particular application requirements. Using the pad control registers, it is possible to set the drive strength, specify an open-drain configuration, choose pull-up and pull-down resistors, control the signal slew-rate, and digital input enable.

Basic Configuration and Control Registers

The specific configuration or operational mode for a GPIO port (and thus set of pins) is established through a user programmable set of *control registers*. The direction of each individual pin, as either an input or an output, is determined by the value of the corresponding bit in a *data direction register*. Based upon the values in the *data direction register*, the associated *data register* either captures incoming data or drives it out to the pads.

When the corresponding *data direction* bit is set to 0, the pin is configured as an input and the corresponding *data register* bit will capture and store the value on the GPIO port pin. When the *data direction bit* is set to 1, the GPIO is configured as an output and the corresponding *data register* bit will be driven out on the GPIO port.

Appendix C:

Background on Interrupts

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

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

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

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

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

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

Appendix D: Working with Interrupts

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

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

   The following are definite no's when writing an ISR

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

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

   Providing the ISR and getting its address (pointer to that function) into the interrupt vector table varies with different environments. For our environment, follow what was done in the *gpio-jtag* project example in the Stellaris environment. Look over the file *gpio-jtag.c* and the file *startup_ewarm.c*.

   Here's what you have to do…..

   1. Write the ISR for your interrupt
   2. Put that above your *main()* routine.
   3. Open the file *startup_ewarm.c* for your project. You had to bring this in when you created the project.
   4. Save a backup of that file in case you make a mistake
   5. Find the interrupt vector table in the *startup_ewarm.c* file....

      ```
      //*****************************************************************
      // The vector table.  Note that the proper constructs must be placed on this to
      // ensure that it ends up at physical address 0x0000.0000.
      //*****************************************************************
      __root const uVectorEntry __vector_table[] @ ".intvec"
      ```

      Find the interrupt that you are intending to use in the vector table and replace the name *IntDefaultHandle* with the name of your ISR and also declare the function prototype as *extern* near the top of the file so that the compiler does not complain..

Look at how this was done in the *startup_ewarm.c* file for the *gpio-jtag* project example.

Also, then read the material in section 8 of the Cortex M3 data sheet on interrupts.

5. One must *enable the interrupt*.

   This can be done globally – all interrupts are enabled – or locally – interrupt x is enabled. Unless the interrupt is enabled, even if it occurs physically, it will *not* be recognized by the system.

   Note: one enables the full port, not an individual pin for an interrupt.

6. One must *acknowledge (or recognize or clear) the interrupt* when it occurs.

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

7. One must *exit the interrupt properly*.

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

External interrupts, or exceptions as they are referred to on the LM3S8962 Cortex processor, are configured and brought in through the GPIO ports. Specifically, one can specify the source of the interrupt, its polarity, and the edge properties (rising or falling edge interrupt).

A detailed discussion of the management and use of Cortex exceptions is given in section 2.5 of the datasheet. Also look through the interrupt example in the IAR Workbench.

Appendix E: RS-232 Interface

Background Information:

Microcomputer communications is a rapidly growing field with an ever-increasing number of applications, ranging from local PC networks to large-scale communication systems. Central to any communications between electronic devices is a protocol for transmitting and receiving information. Within a computer, data is usually transferred in parallel form, such as on a microprocessor or an I/O bus. While parallel communication is far more efficient than serial at moving large numbers of bits, it is not always as practical. Thus, most communication between a computer and other electronic devices that are not in the immediate vicinity is usually done using a serial scheme of one form or another. Typical examples include: I^2C, SPI, Ethernet, USB, WiFi, Firewire, RS-232, etc….

An important element of communication is ensuring that the data given to the user following reception contains no errors arising from such a transmission. Note that we do not guarantee there are no transmission errors, these happen. Rather, at the end of the day, we want to guarantee the data to be correct. There are a variety of schemes by which this is accomplished: all begin with recognizing that a transmission error has occurred. Simple parity is one such scheme. This is the scheme that we'll use here.

Serial Communication: Asynchronous vs. Synchronous

Asynchronous communication suggests that there are irregular intervals between the sending of data. Suppose that a serial communication line is set up to transmit ASCII characters as typed by a person at a keyboard. The spacing between the transmission of each character will necessarily vary widely; there may be long periods when no characters are typed (coffee breaks, tea breaks, bio break, naps, etc.). In such a situation, the receiving device needs to know when a character is being sent to prepare it to receive that character and sort out which part is data, which part is the error-checking field, and so on.

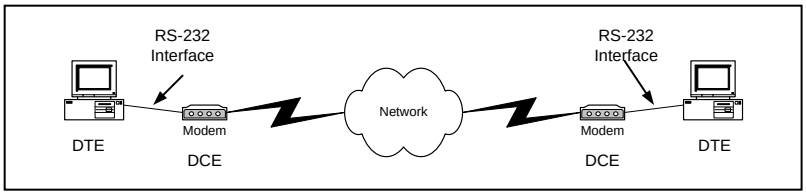
This is accomplished by a procedure known as ***framing***, in which a distinguishing bit, often called a *start bit*, is placed before the first data character, and a different distinguishing bit, a *stop bit*, is placed at the end of the transmission of one character. The start bit enables the receiving device to temporarily synchronize with the transmitting device, while the stop bit allows the receiving device time to get ready for the next frame.

In contrast, when blocks (usually large ones) of regularly spaced data are transferred over a serial line, the transmitter and receiver can be synchronized, thereby enabling the transfer of characters at a much higher rate. In this format, known as *synchronous* transmission, start and stop bits are no longer needed since the receiving device knows exactly where each character begins and ends. Timing information is, none-the-less, still required, thus, a clocking signal will be either sent as a separate signal line or encoded in the data.

A Simple RS-232 Communication System

The RS-232 standard utilizes an asynchronous communication scheme. It was originally developed to specify the interface between a piece of Data Terminal Equipment (DTE) – the device generating or using the data and a piece of Data Communication Equipment (DCE) – the device sending or receiving the data. The standard specifies the circuits and signals between the DTE and DCE devices and the lines that connect them such as the that line data

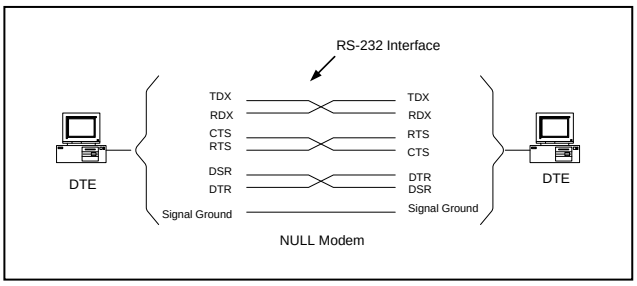
is transmitted over, which lines control status, and so forth as is illustrated in the following figure.



The main subset of comprising signals between the DTE and the DCE is given as:

TXD	Data transmission line from DTE to DCE.
RXD	Data transmission line from DCE to DTE.
DSR	Data Set Ready from DCE to DTE – intended to inform the DTE that the data set has a valid connection, has completed what ever initialization might be necessary, and is ready to engage in a message exchange. If the connection drops during the exchange, this signal is deasserted.
DTR	Data Terminal Ready from DTE to DCE – intended to inform the DCE that the data terminal has completed what ever initialization might be necessary, is ready to engage in a message exchange, and would like to open a communication channel.
RTS	Request to Send from DTE to DCE – intended to inform the DCE that the DTE had data to send and for the DCE to do what ever was necessary (dial, ensure carrier, ensure a line, etc.) to effect the communication.
CTS	Clear to Send from DCE to DTE – intended to inform the DTE that the DCE had done its job and data could now be sent.
CD	Carrier detect – intended to indicate that a connection has been established and an answer tone has been received from the remote modem.
SG	Signal ground – the name says it.

The contemporary view has simplified and modified the original. Today, a simple RS-232 based communication system appears as that in the accompanying figure. Observe that the modems have been eliminated and that the two DTEs are been directly connected together.



For this project, we will utilize such a scheme; however, we will only use TXD, RXD, and SG; the Stellaris EKI-LM3S8962 / Cortex-M3 system will only transmit and the PC will only receive.

The underlying physical driver and interface is implemented either as an integrated component within a microprocessor or microcomputer (such as the Cortex-M3) or externally using an LSI *US/ART – Universal Synchronous / Asynchronous Receiver Transmitter*. The external device may be a peripheral processor that is a component included in the microcomputer's supporting chip set or a general purpose device that is provided by another vendor.

Whichever implementation strategy is used, the device must still be configured to ensure that the sender and the receiver are speaking the same language. The typical minimum set of parameters that must be configured is given in the following list.

Baud Rate – A measure of the speed at which a modem transmits data. Often this is incorrectly assumed to be a measure of the number of bits that are transmitted each second. Baud rate is actually a measure of the number of signaling changes or events per second.

In the early days of digital communication one baud may have been one bit per second; today's technology permits a single event to encode more that one bit. Thus, for example, a modem that is configured to transmit at 9600 baud is actually communicating at 9600 bits per second by encoding four bits per signal event.

Bits per Character – A specification of the size of the character being sent. Typical values are 5, 6, 7, or 8 bits per character

Parity – A specification of the parity over the character. Typical values are odd, even, of none.

Number of Stop Bits - A specification of the number of stop bits to include. Typical values are 1, 1½, or two. Two stop bits are rarely used today. This value originated with the old teletypes that literally moved the carriage back to the start of each line. This mechanical operation took a bit of time to complete.

The IAR software provides wrapper functions around the hardware for setting each of these parameters.

Appendix F: 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.