CPSC 5530/4530 Project 3
Introducing the Project Environment and Working with C – The Next Step

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

Project Objectives:

This project is the second phase in the development of a low cost, portable, medical monitoring system. The current phase focuses on refining the design begun in the previous by replacing several of the modeled / simulated capabilities with the initial design of the drivers to support the different tasks, incorporating a keypad interface, improving the overall flow of control within and enhancing the safety of the system.

The initial deliverables include the high-level system architecture, the ability to perform a subset of the necessary control, and portions of the display and annunciation components. Subsequent phases will continue and extend the driver development and incorporate additional capabilities into both the measurement and display subsystems.

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

In the second phase of the design life cycle of such a system, we will,

1. Add features and capabilities to an existing product.
2. Introduce dynamic task queues.
3. Introduce interrupts and ISRs.
4. Introduce hardware timing functions.
5. Introduce several peripheral devices and work with basic input and output operations.
6. Incorporate a serial local display console into the system.
7. Enhance the safety of the system.
8. Amend the requirements and design specifications to reflect the new features.
9. Amend existing UML diagrams to reflect the new features.
10. Utilize UML diagrams to model new, dynamic capabilities of the system
11. 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

Did well on Project 2, put in at least 600 hours per week. Find out from your mother that she has sublet your room since you never go home and have to stay in school. Come from a good family, don't spill on yourself when you eat…or make rude noises – unless that's appropriate for a particular culture….or you can blame someone else.

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

Cautions and Warnings:

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.

Ensure that you have a balanced number of 0's and 1's in your code. Since they are fatter, too many 0's in you code can clog the cables and decrease the download speed. These will also fill your bit bucket more quickly than 1's and tend to float near the top, being lighter than 1's. 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.

If you find that your systolic and diastolic pressure measurements are a bit lower than you had anticipated, you may an excessive number of clogging 0's, local stores also have wire wideners similar to the devices used in angioplasty; use the device carefully before each download. You might also consider a supply of bit greaser. Be careful, you'll only need a small amount. If you use too much, your bits will be moving too quickly and pass right through the Stellaris board.

In this part of the project, when you need to upload from the Stellaris system, be certain to turn the cables around.

Project:

We will use this project to continue working with the formal development life cycle of an embedded system. Specifically, we will continue the development of a simple kernel with scheduler that will manage a number of tasks and support dynamic task creation and deletion. In support of such capabilities, we will now design and incorporate a dynamic task queue. As with the Phase I design, inter task communication will 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….

- ✓ Design, implement and manage a dynamic task queue,
- ✓ Design, implement, and utilize a hardware based system time base,
- ✓ Work with internal and external interrupts, interrupt service routines, and hardware timing functions,
- ✓ Incorporate a console keypad,
- ✓ Incorporate a local display console which is connected to the system via an asynchronous RS-232 interface,

- ✓ Implement features to enhance safety,
- ✓ Implement and test the new features and capabilities of the system,
- ✓ Utilize UML diagrams to model some of the dynamic aspects of the system,

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 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 level working prototype of that system. Your customer was sufficiently impressed with the work that your firm has done that they have selected and funded you to continue with the next phase of the project. In this phase, we will add features and capabilities to an existing product according to the specifications that follow below.

Phase II Additions

1. A hardware based reference for the system time base will be designed and incorporated to support the timing of task scheduling, switching control, warning, and alarming.
2. The task scheduler will support and manage dynamic task creation and deletion.
3. The display function on the local console OLED will be augmented with a local terminal connected to the system via an RS-232 link. Please see Appendix C and the IAR examples.
4. A console keypad will be incorporated.
5. The initial data logging capabilities will be designed and incorporated.
6. The overall system performance must be improved.
7. The overall system safety 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

Description Modified

Displayed messages comprise three major categories: annunciation, status and warning / alarm. Such information is to be presented on a local display and on a series of lights on the 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 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 a specified time, the aural annunciation shall recommence.

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

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

Phase II Addition

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:

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 II Additions and Modifications

1. Support for dynamic task creation and deletion must be incorporated.
2. The system must incorporate a hardware time base that will provide a temporal reference for task scheduling and measurement management.
3. The display function on the local console OLED will be augmented with a local terminal connected to the system via an RS-232 link. Please see Appendix C and the IAR examples.
4. The system will incorporate a keypad in support of user entered data.
5. Portions of the measurement subsystem and external world interface will be further developed.

6. Preliminary data logging capabilities will be incorporated.
7. Overall system performance must be measured and improved.

2.1 Use Cases

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

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

Phase II

(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 the use case diagrams, a functional decomposition diagram for the system is given as,

Phase II

(To be updated as necessary)

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

Phase II

(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. 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.

Phase II Additions and Modifications

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

a. ***A Dynamic Scheduler****:* Implemented as a doubly linked list of TCBs using a dynamic schedule algorithm capable of adding and removing tasks. A task shall be added to the queue using an *insert* function and removed using a *delete* function.

b. ***Startup Task:*** Incorporate a hardware system time base. Modify the system start up process to initialize and start the underlying timer.

c. ***Timer Interrupt and Time Base****:* Replace the existing software delay based timing reference for the system time base with a periodic hardware timer interrupt.

d. ***Pulse Rate:*** Develop the driver for the pulse rate transducer.

e. ***Data Logging:*** Incorporate the initial support for data logging.

f. ***Display*****:** Develop the driver for a multiline OLED console display to support user data input and to present vital patient information to the local unit.

g. ***Keypad:*** Incorporate a keypad to support user data entry.

h. ***Serial Communications Task****:* Specified status, warning, and alarm information shall be formatted as an RS-232 serial data stream and sent from the processor's (virtual) serial port to a local computer where they are to be displayed using Hyperterm™.

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

2.1.1 Tasks and Task Control Blocks

The design and implementation of the system 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.

Phase II Modification:

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 II

(To be updated as necessary)

2.1.2 Intertask Data Exchange

Intertask data exchange will be supported through shared variables. All system shared variables will have global scope. Based upon the requirements specification, the following shared variables are defined to hold the measurement data, status, and alarm information. The initial state of each of the variables is specified as follows:

Phase II Modifications

Measurements

Data

Delete the following

Type unsigned int

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

Replace with

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
- pulseRateRawBuf[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

Delete the following

Type unsigned char*

- tempCorrected — initial value NULL
- systolicPressCorrected — initial value NULL
- diastolicPressCorrected — initial value NULL
- pulseRateCorrected — initial value NULL

Replace with

Type unsigned char

- 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

Phase II Addition

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:

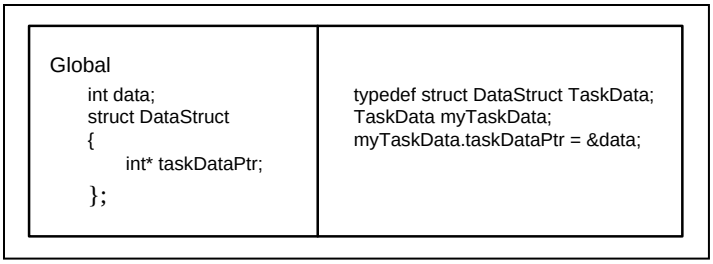
```
enum _myBool { FALSE = 0, TRUE = 1 };

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.

Phase II Delete the following

MeasureData – Holds pointers to the variables:

temperatureRaw
systolicPressRaw
diastolicPressRaw
pulseRateRaw

ComputeData – Holds pointers to the variables:

temperatureRaw
systolicPressRaw
diastolicPressRaw
pulseRateRaw
tempCorrected
sysPressCorrected
diasCorrected
prCorrected

Replace with

MeasureData – Holds pointers to the variables:

temperatureRawBuf
bloodPressRawBuf
pulseRateRawBuf
measurementSelection

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

Phase II Add

KeypadData – Holds pointer to the variables:
- mode
- measurementSelection
- scroll
- select
- alarmAcknowledge

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

The following data structs are defined for the application,

Phase II

(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.

Phase II Modify queue definition

The task queue is implemented as an array of 8 elements that are pointers to variables of type TCB[3].

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

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

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

3. The design of a linked list dynamic task queue is discussed in Appendix B.

2.2 Task Definitions

Phase II Modify Description

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, Annunciate, Warning and Alarm, Status, Local Communications,* and *Schedule*.

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

(To be supplied – by engineering)

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

(To be supplied – by engineering)

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

Phase II New Task

Startup

The task is to run one time each time the system is started and is not contained in 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[4].

The hardware timer will replace the software timer utilized during Phase I.

4. The design must use the timer interrupt – Interrupts are discussed in Appendix A

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

(To be updated as necessary)

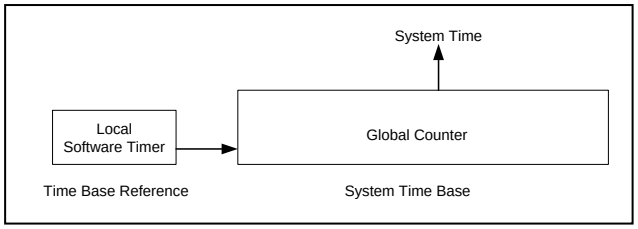
Schedule

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

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

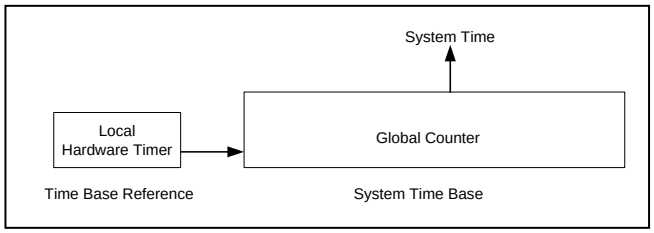
The following block diagram illustrates the design of the system time base. The Global Counter is incremented every time the Local Delay expires. If the Local Delay is 100 ms, for example, then 10 counts on the Global Counter represent 1 sec.

Phase II deletion



Phase II addition

The Software Delay must be replaced by a Hardware Timer.



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

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

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

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

(To be supplied – by engineering)

Measure

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

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

Phase II modification

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*.

temperatureRaw

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

Phase II addition

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

systolicPressRaw

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

When the diastolic measurement is complete, repeat the process.

diastolicPressRaw

Decrement the variable by 2 every even numbered time the function is called and incremented by 1 every odd numbered time the function is called until the value of the variable drops below 40. At such time, set a variable indicating that the diastolic pressure measurement is complete.

The number 0 is considered to be even.

When the systolic measurement is complete, repeat the process.

Phase II addition

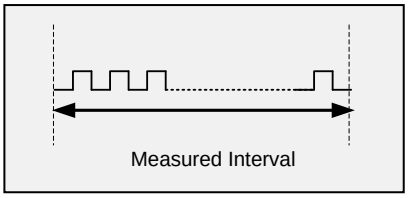
The systolic and diastolic pressures are to be held in a circular sixteen reading buffer. Positions 0 - 7 hold the systolic pressure measurements and positions 8 -15 hold the corresponding diastolic pressure readings.

pulseRateRaw

Phase II - The following task description is deprecated

Decrement the variable by 1 every even numbered time the function is called and increment by 3 every odd numbered time the function is called until the value of the variable exceeds 40. The number 0 is considered to be even. Thereafter, reverse the

process until the value of the variable falls below 15. Then, once again reverse the process.



Phase II Modification

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 will be modeled using an internal software interrupt.

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 Measure task has completed a new set of measurements, the *addTask* flag for the *Compute* task is to be set.

Compute

Phase II Modification

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, then signal the scheduler that it should be deleted.

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

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

Phase II Addition

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.

Phase II Modification

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 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,

(To be supplied – by engineering)

Warning- Alarm

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

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

The normal range for the measurements is specified as follows:

1. Temperature: 36.1 C to 37.8 C
2. Systolic pressure: 120 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 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. If the signal value remains out of range for more than five measurements, the aural annunciation shall resume.

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.

Phase II Addition

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 Hyperterm™. Once finished, it will deleted 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 II

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)

2.4 Performance

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

2.5 General

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

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

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

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

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

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.

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

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

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

Design

Coding

Test / Debug

Documentation

Please include the items listed below in your project report:

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

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

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

Appendix A:

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).* 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.

6. One must *acknowledge(or recognize) 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 B:

A Simple Dynamic Scheduler

A dynamic scheduler works in much the same way that the round-robyn static scheduler from Project 2 does. It has a list of tasks and runs them in a specified order. The difference is that the list of tasks can grow or shrink by adding or deleting tasks.

Once you have properly declared all the necessary data, you must add it to the linked list by invoking the insert function. This function simply adds a TCB node to the end of the list and updates all necessary pointers.

The following illustrates a sample insert function©

Note: Be sure to initialize all pointers to NULL before working with them.

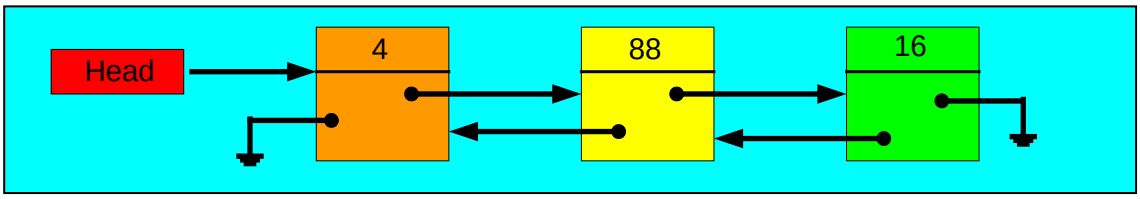
```
// Insert function
// Arguments: Pointer to TCB node
// Returns: void
// Function: Adds the TCB node to the end of the list and updates head and tail pointers

// This function assumes that head and tail pointers have already been created
// and are global and that the pointers contained in the TCB node have already been initialized to NULL
// This function also assumes that the "previous" and "next" pointers in the TCB node are called "prev"
// and "next" respectively

void insert(TCB* node)
{
    if(NULL = = head)       //   If the head pointer is pointing to nothing
    {
        head = node;        // set the head and tail pointers to point to this node
        tail = node;
    }
    else                    // otherwise, head is not NULL, add the node to the end of the list
    {
        tail -> next = node;
        node -> prev = tail; // note that the tail pointer is still pointing
                             // to the prior last node at this point
        tail = node;         // update the tail pointer
    }
    return;
}
```

© *J.D. Olsen, Innovative Software and TA. Ltd., .com*

For the purposes of this project, you will implement a dynamic scheduler using a doubly-linked list of TCBs (note: you may need to modify the TCB structure to accommodate this scheme). The list will be initialized with all tasks except the serial update task which will be added and removed as necessary.



The delete function is a bit more involved but it simply removes a node from the list and updates the necessary pointers. This function should be designed so that a task can delete itself regardless of its position in the list (head, middle, or tail). The following is pseudo-code for the delete function:

```
//  if the head pointer points to NULL, the list is empty and there is nothing to delete otherwise, if the head
// pointer and tail pointer are equal, there is only one node in the list, set the head and tail to NULL
//  otherwise, if the head pointer is equal to the node we want to delete
//  set the head pointer to the next node in the list (head = head -> next)

//  otherwise, if the tail pointer is equal to the node we want to delete, set the tail pointer to the
// previous node in the list otherwise, we are in the middle of the list update the previous and next
// pointers of the surrounding nodes note that this is just pseudo code and does not show all the pointer
// updates be particularly careful to properly set the previous and next pointers of deleted nodes to NULL
```

Appendix C: 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.

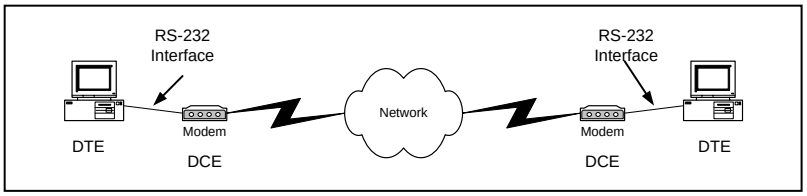
This is accomplished by a procedure known as ***framing***, in which a distinguishing bit, often called a *start bit*, is placed before the first data character, and a different distinguishing bit, a *stop bit*, is placed at the end of the transmission of one character. The start bit enables the receiving 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

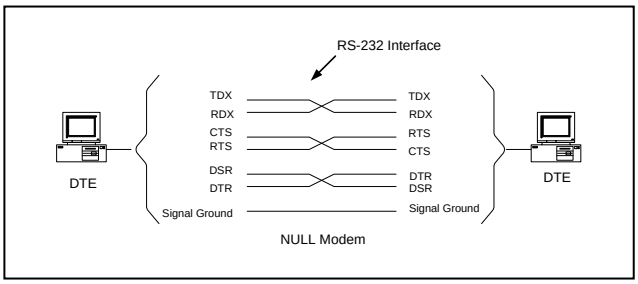
is transmitted over, which lines control status, and so forth as is illustrated in the following figure.



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 D: 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.