EE 472 Lab 4

Summer 2015

Learning the Development Environment – The Next Step

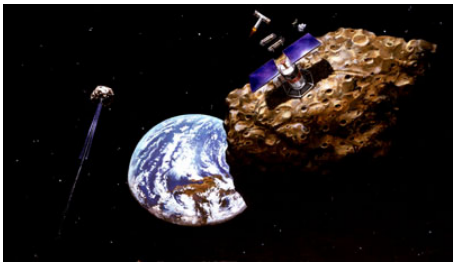
University of Washington - Department of Electrical Engineering

Blake Hannaford, Joshua Olsen, Tom Cornelius,

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

---

Lab Objectives:



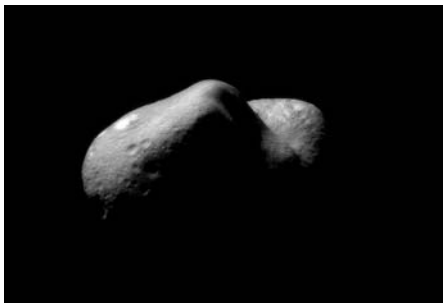
Artist's concept of asteroid mining - wikispaces.com

This lab is the third phase in a project aimed at exploring the possibility of sending a satellite to distant asteroids in hopes of mining them for rare metals that are needed by industry on earth and which may be exhausted in not too many years.

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

Most processors designed for embedded applications utilize just such a computing core. The OS provides a number of software modules called drivers for a wide variety of peripheral devices such as timers, digital input and output channels, telecommunications links, and analog to digital converters that can then be utilized in the application.

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 satellite management 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 FreeRTOS



433 Eros is a stonv asteroid in a near-Earth orbit – wikipedia.com

The final subsystem must be capable of managing the satellite, collecting data from several different types of sensors, processing the data from those sensors, displaying it locally, and using some of the data to control the tools for performing the necessary mining operations. The system will also be responsible for handling the mined material to isolate the desired minerals as well as communicating with the transport vehicle.

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

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 no rain.

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 …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.

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

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.

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. Bill Lynes has these available in stores….just ask him for one.

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…

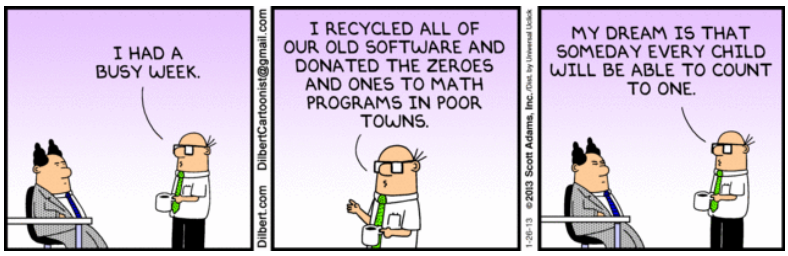
Alternately, blaming your lab partner can work for a short time…until everyone finds out that you are really to blame.

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.

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

©Dilbert by Scott Adams – 26 January 2013



Laboratory:

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 satellite management and control system. To this end, we will continue the development of a simple kernel and scheduler that will handle a number of new and legacy tasks and support dynamic task creation and deletion. We will now introduce and design a dynamic task queue. Each task will continue to share data using pointers and data structs.

In the initial phase, we modeled many of the subsystems as we focused on the flow of control through the system. In the second phase, we started to implement the detailed drivers for the various subsystems.

Now, in the third phase, our main focus will be on porting the system to a real-time operating system (RTOS). 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 under the RTOS,
- ✓ Utilize a priority based preemptive schedule,
- ✓ Upgrade the hardware based system time base,
- ✓ Continue work with the Stellaris Cortex-M3 GPIO subsystem,
- ✓ Incorporate several additional tasks and peripheral devices,
- ✓ Continue work with interrupts, interrupt service routines, and hardware timing functions,
- ✓ 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 lab, lab report, and program are to be done as a team – play nice, share the equipment, and no fighting.

Software Project – Developing Basic Tasks

Your firm, *Extraterrestrial Resources, Ltd*, has just joined a consortium of other companies to work in conjunction with the NASA Institute for Advanced Concepts (NIAC) to explore the possibility of developing an industrial class spacecraft that can venture into space to mine minerals from asteroids and other near earth bodies.

The phase III components of the system are specified below.

Using strip-mining equipment to extract iron and other raw materials from an asteroid.

System Requirements Specification

1.0 General Description

A *Satellite Management and Control System* that will become an integral part of a larger mining system is to be designed and developed. The overall function of the system is to be able to communicate with and control surface based mining equipment from an orbiting command center and to be able to communicate with various earth stations.

Status, warning, and alarm information will be displayed on a system console. For this prototype, the console will be modeled with the OLED display on the Stellaris board. Commands will be received from and pertinent mission information will be sent to an earth station.

The initial phase of the project development focused on the design, implementation, and test of the system data path and control flow. Sensor data, subsystem controls, and incoming coms data were modeled.

The second phase of the project began to design and implement portions of the subsystems that were modeled during the initial phase. The third phase of the project will introduce the FreeRTOS operating system. Additional capabilities will also be incorporated. These are identified in the following modified requirements and design specifications.

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 Additions

1. The system will incorporate a real-time operating system. Please see Appendix A.
2. The battery temperature during a charging cycle will be monitored to ensure that there are no potentially dangerous or destructive problems occurring.
3. The information exchange over the comms link with the mining vehicle will be extended.
4. Comms support for an incoming transport vehicle will be incorporated.
5. The overall system safety must be improved.
6. Amend the requirements and design specifications to reflect the new features.

2.0 Satellite Management and Control System

Phase III Additions and Modifications

The third phase prototype for the *Satellite Management and Control System* is to display alarm information and a portion of the satellite status as well as support basic command and control signaling and subsystem drivers.

Displayed information comprises three major categories: status, annunciation, and alarm. Such information is to be presented on an OLED display and on a series of lights on the front panel.

Power Management and Control

The satellite's power subsystem must track power consumption by the satellite and deploy the solar panels if the system battery (not lithium ion batteries still no nuculer reactor) level falls too low. The system must track and manage,

- Power Generation and Consumption
- Battery Level
- Deployment and retraction of solar panels
- Manual control of solar panel deployment

Phase III Addition

- Battery temperature during a charging cycle

Thruster Management and Control

The thruster subsystem must manage the satellite's thrusters to control the duration and magnitude of thruster burn to affect movement in the desired direction.

Directions:

- Left
- Right
- Up
- Down

Thruster Control:

- Thrust - Range
  - Full OFF
  - Full ON
- Thrust Duration

Communications Requirements

The satellite must support bidirectional communication with earth stations to transmit status, annunciation, and warning information from the satellite and to receive mission commands from the earth stations.

For the current prototype, incoming commands are limited to thruster control.

The satellite must support bidirectional communication with the land based mining vehicle to receive mission commands from the earth stations and to transmit status, annunciation, and warning information.

Phase III Modification and Addition

The satellite must support commands initiated by the land based mining vehicle.

The satellite must support commands initiated by an inbound transport wishing to dock with the satellite.

Status and Annunciation Subsystem Requirements

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

Status

- Solar Panel State
- Battery Level
- Power Consumption and Generation
- Fuel Level

Phase III Addition

- Battery Temperature
- Transport Distance

Warning and Alarm

- Fuel Low
- Battery Low

2.1 Use Cases

The following use cases express the external view of the system (see Chapter 5 in the text),

Phase IIII

(To be updated as necessary– by engineering … this would be you)

Software Design Specification

1.0 General Description

A prototype for a *Satellite Management and Control System* is to be developed. The high-level design shall be implemented as a set of tasks that are executed, in turn, forever.

The prototype will be implemented using the Stellaris development board. The prototype software will schedule task execution, implement the drivers necessary to control specified satellite subsystems, control the OLED on the board, and manage the peripheral LEDs 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 display and alarm portion of the system and incorporates the Phase II modifications and additions.

2.0 Functional Decomposition – Task Diagrams

Phase III Additions and Modifications

The system is decomposed into the major functional blocks: *Manage Power Subsystem, Manage Solar Panel, User Input Data, Manage Thrusters, Satellite Communications, Landbased Mining Vehicle Communications, Transport Communication, Status and Annunciation Display, Warning and Alarm,* and *Schedule*.

These blocks decompose into the following task/class diagrams (see Chapter 5 in the text),

Phase III

(Functional design to be updated to reflect additions and modifications)

2.1 System Software Architecture

Phase III Additions and Modifications

The *Satellite Management and Control System's* high-level system software architecture must support multitasking operations and real-time behaviour. The design will comprise a set of tasks that, following power ON, are executed continuously, utilizing a pre-emptive, priority based scheduling algorithm. Information within the system will be exchanged utilizing a number of shared variables.

To implement the required additions to the product, the following Phase III capabilities must be incorporated.

a. ***A Real Time Operating System*** - Implemented using the FreeRTOS (RTOS) system.

b. ***Startup Process***: Modify the task creation and initialization process when the system starts as required by the operating system.

c. ***Thruster Drive****:* The thrusters will be driven with a PWM signal.

d. ***Solar Panel Control****:* The system shall provide a PWM signal to drive an electric motor. The speed shall be set to a default value or manually controlled from the command console. The speed shall range from full OFF to full ON.

e. ***Local Keypad:*** Incorporate a keypad to support manual control of the solar panels.

f. ***Charging Transducer Connection***: An external interrupt will signify that the battery level transducer and measurement equipment have successfully connected to the solar panel and the signal is stable. Voltage level measurements will then be performed at a specified duration following such a signal.

g. ***OLED Display:*** Extend and add capabilities for the driver and API for the OLED display.

h. ***Land Based Vehicle Communication****:* The exchange of specified commands as well as status, warning, and alarm information between the land based mining vehicle or transport vehicle and the satellite shall be modeled using an RS-232 serial connection. Data collected from the land vehicle will be relayed by the satellite to an earth station where it can be displayed using Hyperterm™. The system will support requests initiated by and respond to the land vehicle or the transport vehicle.

i. ***Transport Distance:*** Detect and measure the frequency of a signal automatically transmitted from an arriving transport vehicle. The signal frequency shall be proportional to the transport's distance from the satellite

j. ***Battery Temperature:*** Support for monitoring the temperature in several places on the power subsystem battery.

2.1.1 Tasks and Task Control Blocks

The design and implementation of the system software will comprise a number of tasks. Each task will be expressed as a TCB (Task Control Block) structure.

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 given for the tasks are defined for the application

Phase III

(To be updated to reflect additions and modifications)

2.1.2 Intertask Data Exchange and Control Data

All of the system's shared variables will have global scope; the remainder will have local scope. Based upon the Requirements Specification, the following variables are defined to hold the status and alarm information or command and control information.

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

Thruster Control

Global - Type unsigned int

Thruster Command	initial value	0

Local - Type unsigned short

Left	initial value	0
Right	initial value	0
Up	initial value	0
Down	initial value	0

Power Management

Global - Type unsigned int*

Battery LevelPtr	initialize to point to a 16 reading measurement data buffer	

Global - Type unsigned short

Fuel Level	initial value	100
Power Consumption	initial value	0
Power Generation	initial value	0

Global - Type Bool[1]

Solar Panel State	initial value	FALSE
Solar Panel Deploy	initial value	FALSE
Solar Panel Retract	initial value	FALSE

Phase III Modification

Mining Vehicle Communications:

Global - Type char

Command	initial value	NULL
Response	initial value	NULL

(To be supplied by engineering)

Phase III Addition

Battery Temperature Measurement:

(To be supplied by engineering)

Transport Distance:

(To be supplied by engineering)

Transport Communications:

(To be supplied by engineering)

Solar Panel Control:

Global - Type Bool[1]

Solar Panel State	initial value	FALSE
Solar Panel Deploy	initial value	FALSE
Solar Panel Retract	initial value	FALSE
Drive Motor Speed Inc	initial value	FALSE
Drive Motor Speed Dec	initial value	FALSE

Local - Type unsigned short

Motor Drive	initial value	0

Range

Full OFF

Full ON

Control

Status Management and Annunciation:

Global - Type Bool[1]

Solar Panel State	initial value	FALSE

Global - Type unsigned short

Battery Level	initial value	100
Fuel Level	initial value	100
Power Consumption	initial value	0
Power Generation	initial value	0

Phase III Addition

(To be supplied by engineering)

Transport Distance

warningAlarm:

Global - Type Bool

Fuel Low	initial value	FALSE
Battery Low	initial value	FALSE

Phase III Addition

(To be supplied by engineering)

Battery over Temperature

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

We can emulate the Boolean type as follows:

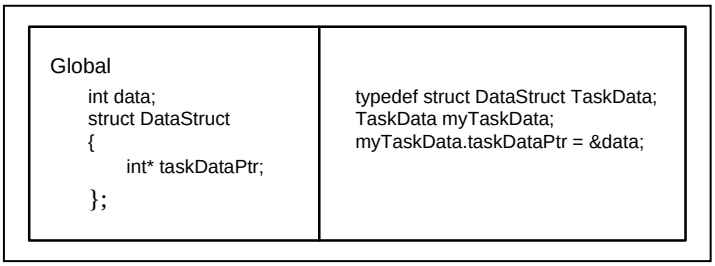
```
enum myBool { FALSE = 0, TRUE = 1 };

typedef enum _myBool Bool;
```

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

2.1.3 Data Structures

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



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

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

powerSubsystemData – Holds pointers to the variables:

Solar Panel State

Solar Panel Deploy

Solar Panel Retract

Battery LevelPtr

Power Consumption

Power Generation

Phase III Addition

Battery over Temperature

Battery Temperature

solarPanelControlData – Holds pointers to the variables:

Solar Panel State
Solar Panel Deploy
Solar Panel Retract
Drive Motor Speed Inc
Drive Motor Speed Dec

keyboardConsoleData – Holds pointers to the variables:
Drive Motor Speed Inc
Drive Motor Speed Dec

thrusterSubsystemData – Holds pointers to the variables:
Thruster Command
Fuel Level

satelliteComsData – Holds pointers to the variables:
Fuel Low
Battery Low
Solar Panel State
Battery Level
Fuel Level
Power Consumption
Power Generation
Thruster Command

vehicleComsData – Holds pointers to the variables:
Command
Response

Phase III Addition

(To be supplied by engineering)

Phase III Addition

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

OLEDdisplayData – Holds pointers to the variables:

- Fuel Low
- Battery Low
- Solar Panel State
- Battery Level
- Fuel Level
- Power Consumption
- PowerGeneration

Phase III Addition

- Battery Temperature
- Transport Distance

warningAlarmData – Holds pointers to the variables:

- Fuel Low
- Battery Low
- Battery Level
- Fuel Level

Phase III Addition

- Battery over Temperature

The following data structs are defined for the application,

(To be supplied by engineering)

2.1.4. External Events and Inputs

The *Satellite Management and Control System* must support an interface to a sensor that monitors connections to and the state of the satellite battery. The output of the sensor signaling event shall indicate that the solar panel output and the measurement equipment is properly connected to the battery.

When an interrupt occurs, the flag *stable* to signal the connection event must be set.

To prevent damage to the solar panels during either deployment or retraction, a deployment sensor on the solar panel will generate a signaling event to indicate end of travel.

When an interrupt occurs, the flag *endofTravel* must be set and read by the *solarPanelControl* driver to signal that the drive signals must be terminated.

The *Satellite Management and Control System* must support an interface to a sensor that detects a signal from an inbound transport vehicle. The frequency of the incoming signal shall be proportional to the distance between the satellite and an inbound transport vehicle.

2.1.5 Task Queue

Phase III Modification

The tasks comprising the application will be held in a task queue and will be pre-emptable. The initial task queue will be created under the startup task in the FreeRTOS OS. Thereafter, it will managed by the RTOS scheduler. Details of the OS, the startup task, and TCBs are given in Appendix A.

The eight static TCB elements in the queue correspond to the tasks: *Manage Power Subsystem, Manage Solar Panel, User Input Data, Manage Thrusters, Satellite Communications, Landbased Mining Vehicle Communications, Status and Annunciation Display, Warning and Alarm* identified in section 2.2. Two tasks, the *solarPanelControl* and *transportDistance* and one subtask, *batteryTemperature* are to be added or deleted as needed.

Note: When a context switch is made because a running task blocks or terminates or on return from interrupt, the highest priority task that is ready to run will be selected from the queue, started or resumed, and moved to the run state. For the case of an interrupt, the resumed task may or may not be the interrupted task.

2.2 Task Definitions

The system is decomposed into the major functional blocks as given in the following diagram,

(To be supplied – by engineering)

Phase III

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

(To be supplied – by engineering)

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.
- Configure and initialize all hardware subsystems.
- 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

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

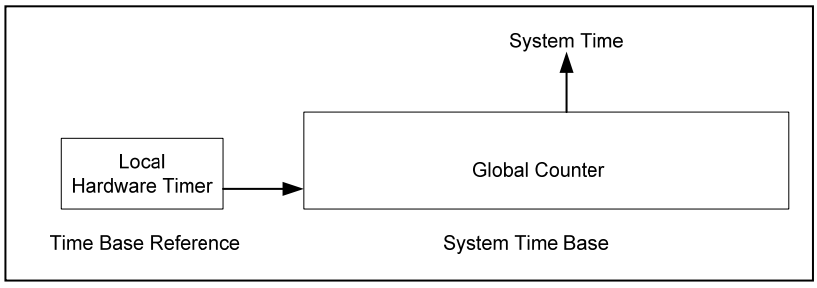
The round robyn task schedule comprises a major cycle and a series of minor cycles. The period of the major cycle is 5 seconds. The duration of the minor cycle is specified by the designer.

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. 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.

The Software Delay must be replaced by a Hardware Timer.



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

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

(To be supplied – by engineering)

PowerSubsystem

The *powerSubsystem* task manages the satellite's power subsystem.

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

Remember, the pointer in the task argument must be re-cast as a pointer to the *powerSubsystem* task's data structure type before it can be dereferenced.

The values of the various parameters must be simulated because the sensors are currently unavailable. To simulate the parameter values, the following operations are to be performed on each of the data variables referenced in powerSubsystemData.

powerConsumption

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 10. The number 0 is considered to be even.

Thereafter, reverse the process until the value of the variable falls below 5. Then, once again reverse the process.

powerGeneration

If the solar panel is deployed

If the battery level is greater than 95%

Issue the command to retract the solar panel

Else

Increment the variable by 2 every even numbered time the function is called and by 1 every odd numbered time the function is called until the value of the battery level exceeds 50%. Thereafter, only increment the variable by 2 every even numbered time the function is called until the value of the battery level exceeds 95%.

If the solar panel is not deployed

If the battery level is less than or equal to 10%

Issue the command to deploy the solar panel

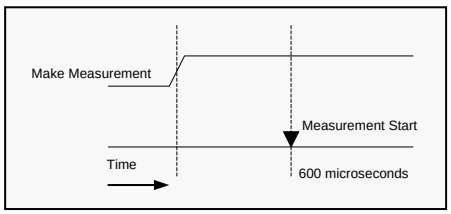
Else

Do nothing.

Phase III Modification

batteryLevel

The batteryLevel is to be read as an analog signal on A/D Channel 0. Because the signal takes time to settle following the connection of the solar panel output and the measurement equipment to the battery, to ensure a valid reading, the measurement is to be made 600 µsec following an external event signaling the connection.



The output of the battery level is a DC analog signal in the range of 0 to +32V; however, the A/D can only accept signals in the range of 0 to +3.25 V. Consequently, the raw battery level signal must be buffered to a range that is compatible with the A/D input levels.

Post measurement, the measured value must be scaled back into the 0 to +32V range for display.

The system shall store the 16 most recent samples into a circular buffer.

Phase III Addition

batteryTemperature

The *Satellite Management and Control System* will utilize a series of infrared sensors to monitor the temperature of the battery while it is being charged. The output from a sensor is an analog signal with a DC output range of 0 to + 325 mV.

The sensors are non-linear devices and return the measured temperature in millivolts. As a result, the measured values must be amplified to the range 0 to +3.25 V and converted to the appropriate units for display.

The equation relating the amplified signal, *battTemp* and the temperature in Celsius is:

```
Temperature = 32*battTemp + 33
```

For the prototype, it is sufficient to measure the temperature at 2 different locations. The signals are to be read as analog signals on input A/D Channels 1 and 2 at a periodic interval of 500µs while the battery is being charged.

The system shall store the 16 most recent samples into a buffer. The stored values shall be expressed in millivolts. The subtask is to be scheduled on demand.

If either of the two currently measured values exceeds the largest of the two most recent previously measured values by more than 20% the task shall signal an alarm and the OLED Display.

SolarPanelControl

The *solarPanelControl* task manages the deployment and retraction of the satellite's solar panels.

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

Remember, the pointer in the task argument must be re-cast as a pointer to the *solarPanelControl* task's data structure type before it can be dereferenced.

The system shall be capable of providing a PWM signal to drive the electric motor that deploys or retracts the solar panels. The period of the PWM signal shall be 500 ms.

The speed shall be controlled either manually through pushbuttons on an earth based command and control console or set, based upon a preset value.

If the panels are being controlled manually, the speed shall be incremented or decremented by 5% for each press of the corresponding console pushbutton. The speed shall range from full OFF to full ON.

When the panels have reached either full deployment or full retraction, an interrupt signal shall be generated from a sensor on the panel signaling that the drive signal should be terminated. Attempts to overdrive the panels beyond their normal travel can result in potential damage to the drive subsystem.

The task is to be scheduled on demand.

Console Keypad Task

The *Console 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 *Console Keypad* task's data structure type before it can be dereferenced.

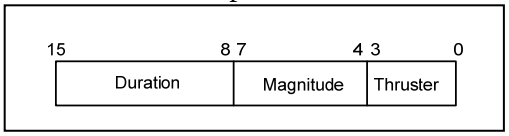
The console keypad is used to manually control the solar panel drive motors in increments of ±5.0%.

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

The task is only scheduled during deployment or retraction of the solar panels.

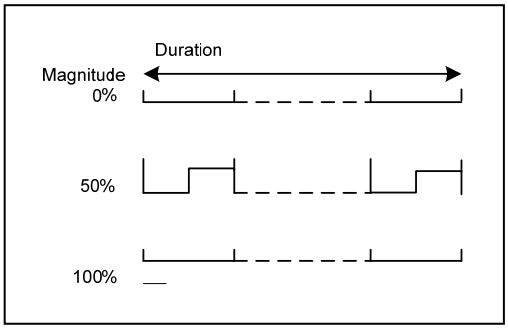
ThrusterSubsystem

The *thrusterSubsystem* task handles satellite propulsion and direction based upon commands from the earth.



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

Remember, the pointer in the task argument must be re-cast as a pointer to the *thrusterSubsystem* task's data structure type before it can be dereferenced.



The format for the thruster command is given in the accompanying figure.

The *thrusterSubsystem* task will interpret each of the fields within the thruster command and generate a control signal

of the specified magnitude and duration to the designated thruster.

The thruster control signals, for a specified duration, with magnitudes of 0%, 50%, and 100% of full scale are given in the accompanying figure.

If fuel is expended at a continuous 5% rate, the satellite will have a mission life of 6 months. The *thrusterSubsystem* will update the state of the fuel level based upon the use of the thrusters.

SatelliteComms

The *satelliteComs* task handles communication with the earth.

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

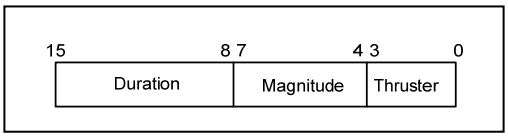
Remember, the pointer in the task argument must be re-cast as a pointer to the *satelliteComs* task's data structure type before it can be dereferenced.

Data transfer from the satellite to the earth shall be the following status information:

- Fuel Low
- Battery Low
- Solar Panel State
- Battery Level
- Fuel Level
- Power Consumption
- Power Generation

Data transfer from the earth to the satellite shall be the following thrust command:

The thrust command shall be interpreted as follows,



Thruster ON	Bits 3-0
Left	0
Right	1
Up	2
Down	3
Magnitude	Bits 7-3
OFF	0000
Max	1111
Duration - sec	Bits 15-8
0	0000
255	1111 1111

At the moment, the coms link is not available, thus, we must simulate it using a random number generator. We will use such a generator to produce a random 16-bit number to model the received thrust command.

We have posted a simple program, rand1.c in the Lab2 folder.

You can also modify the code from this program to build your own random number generator for this task.

If you choose to use this code, make certain that you cite where the code came from in your source code.

VehicleComms

The *vehicleComms* 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 *vehicleComms* task's data structure type.

The *vehicleComms* task will manage a bidirectional serial communication channel between the satellite and the land based mining vehicle. The exchange shall comprise specified commands from earth as well as returned status, warning, and alarm information. Information returned from the vehicle will be relayed by the satellite to the earth where it can be displayed using Hyperterm™ or on the Serial Port Monitor.

For the current phase, the following commands, requests, and responses will be implemented:

Commands to Mining Vehicle:

F	Forward
B	Back
L	Left
R	Right
D	Drill down – Start
H	Drill up – Stop

Response:

A<sp Command sent>

Phase III Addition

Requests from the Mining and Transport Vehicles:

T	Request for transport lift-off
D	Request to dock

Response:

K	OK to lift-off
C	Confirm dock

Phase III Addition

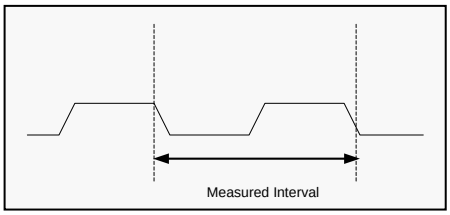
TransportDistance

To transfer the mined minerals back to earth, a transport vehicle is launched from the land based vehicle, rendezvous with, then docks with the satellite.

The *transportDistance* task tracks the distance between the satellite and the approaching transport. The *transportDistance* function 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 *transportDistance* task's data structure type before it can be dereferenced.

The transport is equipped with a transmitter that is automatically activated when it is within 1 km of the satellite. The frequency of the emitted signal, which increases as the transport gets closer, is detected by the *Satellite Management and Control System*.



The transmitter output appears as successive negative transitions. Such transitions can be detected as an external event interrupt and used to determine the transport's distance from the satellite.

The time interval between successive transitions must be determined and stored in a buffer. The measured values must be converted to a distance in meters and stored in a circular 8 reading buffer, *transportDistance*, if the current reading is more than 10% different from the previously stored reading.

The distance data must be displayed on the local display console.

The transmitter – receiver pair is currently under development. One of the objectives of the present phase is to obtain some field data on a beta version. To this end, the upper frequency limit of the incoming signal shall be empirically determined. The upper limit will correspond to 100 meters. The maximum distance measurement is to correspond to 2 kilometers.

The task shall be scheduled on demand.

OLEDdisplay

The *oledDisplay* task manages the display of the satellite status and alarm information.

The *oledDisplay* function 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 *oledDisplay* task's data structure type before it can be dereferenced.

The *oledDisplay* task will support two modes: *Satellite Status* and *Annunciation*.

In the *Satellite Status* mode, the following will be displayed for the satellite

- Solar Panel State
- Battery Level
- Fuel Level

- Power Consumption

Phase III Addition

- Battery Temperature
- Transport Distance

In the *Annunciation* mode, the following will be displayed

- Fuel Low
- Battery Low

WarningAlarm

The *warningAlarm* function 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 *warningAlarm* task's data structure type.

The *warningAlarm* task interrogates the state of the battery and fuel level to determine if they have reached a critical level.

- If both are within range, the green LED on the annunciation panel shall be illuminated and on solid.
- If the state of the battery level reaches 50%, the yellow LED on the annunciation panel shall flash at a 1 second rate.
- If the state of the fuel level reaches 50%, the yellow LED on the annunciation panel shall flash at a 2 second rate.
- If the state of the battery level reaches 10%, the red LED on the annunciation panel shall flash at a 1 second rate.
- If the state of the fuel level reaches 10%, the red LED on the annunciation panel shall flash at a 2 second rate.

Phase III Addition

- If the over temperature event occurs, an audible alarm shall be set and shall remain active until acknowledged. If the alarm is unacknowledged for more than 15 seconds, the red and yellow LEDs shall flash with the pattern: flash for 5 seconds at a 10Hz rate – remain solid on for 5 seconds repeat cycle.

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)

2.4 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 GPIO 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.

  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.

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 to 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. Understand, and correct if necessary, any compiler warnings.
   c. Correct any compile errors and warnings.
   d. Test your code.
   e. Repeat steps a-d as necessary.
   f. Write your report
   g. Demo your project.
   h. Go have a beer.

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

4.0 Lab Report

Write up your lab report following the guideline on the EE 472 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 lab 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

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. Be sure to include all of the items identified as ‘to be provided by engineering.’
6. 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.
7. 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.
8. If a stealth submersible sinks, how do they find it?
9. Does a helium filled balloon fall or rise south of the equator?
10. If you fly faster than the speed of sound, do you have to slow down every now and then to let the sound catch up?
11. If you fly really really fast around the world, can you catch up to the sound before it catches up to you and answer a question before you hear it?
12. If you don’t answer a cell phone call, where does it go? Is it just sitting there waiting for you?

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.

Also, the class webpage for a working example:

http://www.ee.washington.edu/class/472/peckol/code/StellarisExamples/

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 *),      //  pointer to the task function
                                                 //  the task prototype is of the form
                                                 //    void task (void *pd);
                    char *name,                  //  name of the task
                    int stackDepth,              //  size of the stack allocated for the task
                    void *parameters,            //  pointer to the data passed in to the task
                    int priority,                //  task priority 0..5, lowest … highest
                    taskHandle *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 to 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 lab, 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 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 value 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, we must do 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 it's 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.