EE 474 Project 2

Summerer 2017

Learning the Development Environment – The Next Step

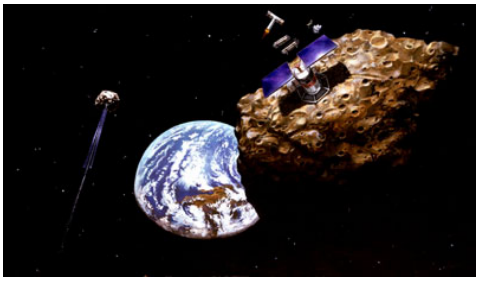
University of Washington - Department of Electrical Engineering

Blake Hannaford, Joshua Olsen, Tom Cornelius,

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

---

Project Objectives:

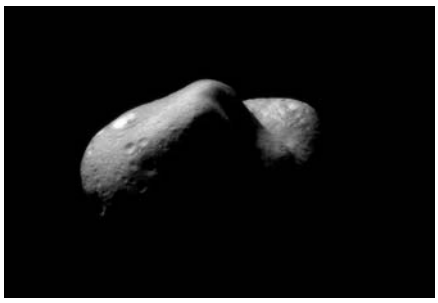


Artist's concept of asteroid mining - wikispaces.com

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

The first phase of the project focuses on the design and development of the overall control flow within the system as well as developing the drivers for managing the basic satellite operations, status display, and alarm and warning annunciation functions.

The initial deliverables include full specification and design documentation for the portions of the system under current development, the high-level system architecture, the ability to perform a subset of the necessary control functions, and portions of the display and annunciation components. Subsequent phases will continue and extend the driver development and incorporate additional capabilities into both the satellite and mining component.



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

The final subsystem must be capable 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 preprocessing the mined material to isolate the desired minerals and communicating with other spacecraft in the area.

In developing such a system, we will,

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

Prerequisites:

Familiarity with C programming, the Texas Instruments *Sitara, AM3358BZCZ100* implementation of the ARM Cortex-A8 microcomputer, and the Linux development environment.

Background Information:

Did well on Project 1, put in at least 300 hours per week (it's heading deeper into summer in Seattle – the skies are a wonderfully overcast with beautiful shades of gray, the rain is falling, the temperatures are dropping, soon it will be the fourth of July – wow, it's going to be hard to concentrate). No need to sleep – it's over rated anyway.

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

Cautions and Warnings:

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

Since current is dq/dt, if you are running low on current, raise your BeagleBone 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 your Display Terminal is really a solar panel.

If the environment is downloading your binaries too slowly, lower your BeagleBone board so that it is substantially below the connection on the PC and put the terminal 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 BeagleBone 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.

Debugging your design by throwing your completed but malfunctioning system on the floor, stomping on it, and screaming 'why don't you work you stupid fool' is typically not the most effective technique although it is perhaps one of the more satisfying. The debugging commands, *step into* or *step over*, are 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 BeagleBone 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 BeagleBone 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. That said, we strongly encourage you to make backup copies; it will save you a lot of headaches.

Always make certain that the cables connecting the PC to the BeagleBone 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.

Project:

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

- Identify the **requirements**.
- From the requirements, put together a **design specification** quantifying those requirements.
- Do a **functional design** – identify the major functional blocks that will comprise the system.
- Formulate an **architecture** then map the functional blocks onto that architecture.
- **Design**, **build**, and **test** a prototype of the system.

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

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

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

As we begin the development, we will….

- ✓ Utilize UML diagrams to model some of the static and dynamic aspects of the system.
- ✓ Identify the major functional blocks.
- ✓ Architect the system as a collection of tasks.
- ✓ Develop a simple time based operating system kernel with scheduler and dispatcher that will schedule and dispatch those tasks.
- ✓ Stub out and simulate the desired behaviour of each of the tasks.
- ✓ Implement and test the system.
- ✓ Empirically determine execution time of each such task.

- ✓ Share data using pointers and data structs.

Once again, we will work with the Linux development tools to edit and build the software then download and debug the executable code in the target environment.

This project, project report, and program are to be done as a team – play nice, share the equipment, keep any viruses (software or otherwise) to yourself, and no fighting.

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

Figure 1

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), Space-X, and Blue Origin 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 first component of the system is shown in the accompanying figure and the system specifications are listed below.

System Requirements Specification

1.0 General Description

A *Satellite Management and Control System* that will become an integral part of the mining system is to be designed and developed. The overall function of the system is to be able to 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 a terminal display in the Linux environment. Commands will be received from and pertinent mission information will be sent to an earth station.

The current phase of the project development will focus on the architecture, design, implementation, and testing of the system data path and control flow. Sensor data, subsystem controls, and incoming coms data will be modeled.

2.0 Satellite Management and Control System

The first 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:

Power Management and Control

The satellite's power subsystem must track power consumption by the satellite and deploy the solar panels if the system's battery (not lithium ion batteries or nuculer reactors) level falls too low. The system must track,

- Power Consumption
- Battery Level
- Deploy and retract solar panels

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 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.

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

Warning and Alarm

- Fuel Low
- Battery Low

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

2.1 Use Cases

The following use cases express the external view of the system (see Chapter 5 in the recommended class text and the *UML-MyPhone* example under documentation on the class webpage),

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

Software Design Specification

1.0 General Description

A prototype for a *Smart 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 BeagleBone development board. The prototype software will schedule task execution, implement the drivers necessary to control specified satellite subsystems, control the LCD, and manage the peripheral LEDs through the processor's GPIO 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.

2.0 Functional Decomposition – Task Diagrams

The system is decomposed into the major functional blocks: *Manage Power Subsystem, Manage Thrusters, Satellite Communications, Status and Annunciation Display, Warning and Alarm,* and *Schedule*. These blocks decompose into the following task/class diagrams (see Chapter 5 in the recommended class text),

(To be supplied by engineering)

2.1 System Software Architecture

The *Smart Satellite Management and Control System* is to be designed as a collection of tasks that execute continuously following power ON. The system tasks will all have equal priority and will be not be pre-emptable. Information within the system will be exchanged utilizing a number of shared variables.

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.

The TCB is implemented as a C struct; there shall be a separate struct for each task.

Each TCB will have two members:

i. The first member is a pointer to a function taking a void* argument and returning a void.
ii. The second member is a pointer to void used to reference the shared data for the task.

Such a structure allows various tasks to be handled using function pointers.
The following gives a C declaration for such a TCB.

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

The following function prototypes are given for the tasks defined for the application
(To be supplied by engineering)

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 and 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 short

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

Solar Panel Control:

Global - Type Bool[1]		
Solar Panel State	initial value	FALSE
Local - Type unsigned short		
Motor Drive	initial value	0
Range		
Full OFF		
Full ON		

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

warningAlarm:

Global - Type Bool		
Fuel Low	initial value	FALSE
Battery Low	initial value	FALSE

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

We can emulate the Boolean type as follows:

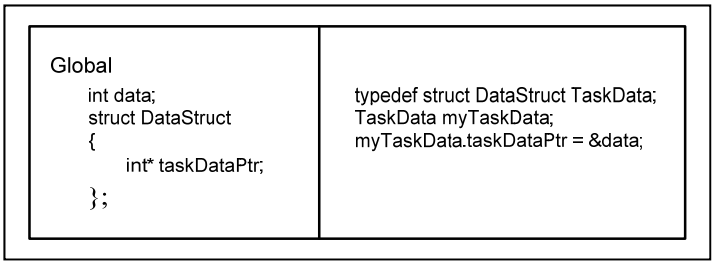
```
enum myBool { FALSE = 0, TRUE = 1 };

typedef enum _myBool Bool;
```

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

2.1.3 Data Structures

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



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

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
- Battery Level
- Power Consumption
- Power Generation

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

consoleDisplayData – Holds pointers to the variables:

Fuel Low

Battery Low

Solar Panel State

Battery Level

Fuel Level

Power Consumption

Power Generation

warningAlarmData – Holds pointers to the variables:

Fuel Low

Battery Low

Battery Level

Fuel Level

The following data structs are defined for the application,

(To be supplied by engineering)

2.1.4 Task Queue

The tasks comprising the application will be held in a task queue. Tasks will be selected, in sequence, from the queue in round robyn fashion and executed. Tasks will not be pre-emptable; each task will run to completion.

If a task has nothing to do, it will exit immediately.

The task queue is implemented as an array of 6 elements that are pointers to variables of type TCB.

Five of the TCB elements in the queue correspond to tasks managing the *powerSubsystem, thursterSubsystem, satelliteComs, consoleDisplay,* and *WarningAlarm* identified in Section 2.2. The sixth element provides space for future capabilities.

The task function pointer in each TCB should be initialized to point to the proper task. For example, TCB element zero should have its function pointer initialized to point to the task0 function.

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

2.2 Task Definitions

The system is decomposed into the major functional blocks as given in the following functional decomposition (see Chapter 9 in the text).

(To be supplied – by engineering)

The dynamic behaviour of each task is given, as appropriate, in the following activity diagrams (see Chapter 5 in the recommended class text).

(To be supplied – by engineering)

Schedule

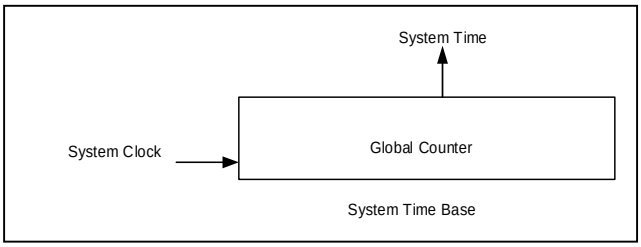
The *schedule* task manages the execution order and period of the tasks in the system. However, the task is not in the task queue.

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 basic design. The Global Counter is managed by the Linux operating system, but can be read by any task. We will use the C standard library time functions to access the counter.

All tasks can utilize the System Time Base as a reference upon which to base their local timing.



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

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

(To be supplied – by engineering)

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.

batteryLevel

If the solar panel is not deployed

Each time the function is called, compute the current battery level as follows,

batteryLevel = batteryLevel - 3●powerConsumption

If the solar panel is deployed

Each time the function is called, compute the current battery level as follows,

batteryLevel = batteryLevel – powerConsumption + powerGeneration

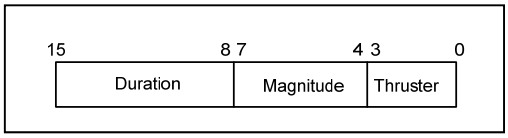
ThrusterSubsystem

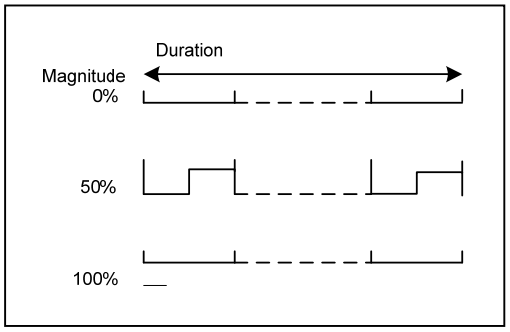
The *thrusterSubsystem* task handles satellite propulsion and direction based upon commands from the earth.

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 following 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.

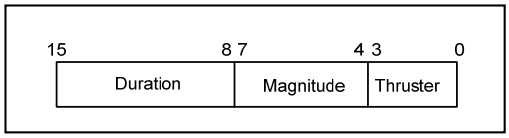
Remember, the pointer in the task argument must be re-cast as a pointer to the *satelliteComs* task's data structure type before it can be dereferenced.

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 comms 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 Project2 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 and project report.

ConsoleDisplay

The *ConsoleDisplay* task manages the presentation of the satellite status and alarm information on a console display.

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

The *ConsoleDisplay* 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

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 LED3 on the annunciation panel shall be illuminated and on solid.
- If the state of the battery level reaches 50%, LED2 on the annunciation panel shall flash at a 1 second rate.
- If the state of the fuel level reaches 50%, LED2 on the annunciation panel shall flash at a 2 second rate.
- If the state of the battery level reaches 10%, LED1 on the annunciation panel shall flash at a 1 second rate.
- If the state of the fuel level reaches 10%, LED1 on the annunciation panel shall flash at a 2 second rate.

2.6 Performance

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

2.7 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 control flow 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.

To accomplish this, we use the Linux function: "`usleep(int time_in_ms)`". Thereafter, simply call this function with the delay in milliseconds as its argument. Remember Lab 1.

Required 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 control flow 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 control flow 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, 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 potentially have a significant affect the successful completion of your design / project.**

Project Report

Write up your lab report following the guideline on the EE 474 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 copy your code from some outside webpage or someone else's code.**
- ✓ **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: General Purpose I/O – GPIO Overview

GPIO is a generic pin on an integrated circuit. In our case, the GPIO pins are directly mapped to the AM335x processor. The behavior of these pins can be controlled by the user at run time. Its behavior is to output and receive high or low states.

For our boards, the high state is defined to be at 3.3 V and the low state is 0 V. These pins can be used to configure other hardware devices such as LEDs and LCD screens. Let's practice using GPIOs to turn on an LED. You can use the hardware schematics and the pin configuration files to choose and configure a GPIO pin.

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. The module supports up to 69 programmable input/output pins, depending on the peripherals being used. In reality the majority of the pins are being used by onboard system processes.

Details of the GPIO subsystem, specific capabilities, and how they can be configured is given in the following:

http://beagleboard.org/Support/bone101

Headers and Pinout

The board has two 23 pin columns on each side of the board, for a total of 92 pins available to the user. For Beaglebone Black, all of the GPIO numbers are provided in Figure B1, copied from the first project. The LEDs are located here.

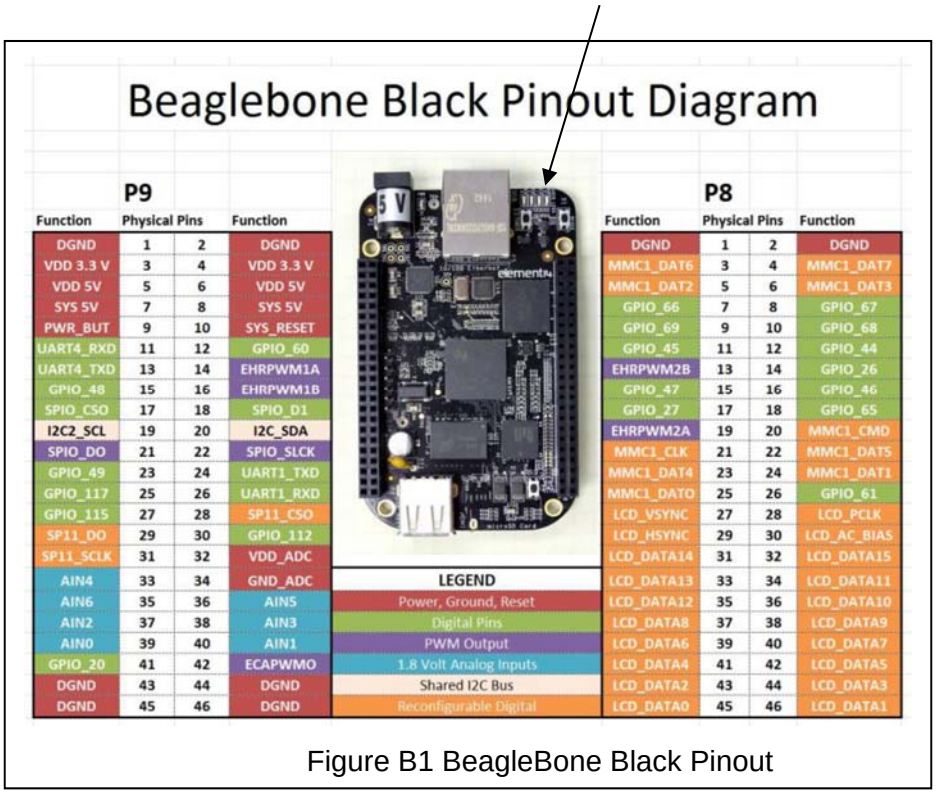


Figure B1 BeagleBone Black Pinout

Note that the right header is designated as P8 while the left is designated P9. In schematics and documentations, you will see that all of the GPIO pins have the format GPIO*x* [*y*]. The letter *x* represents the bank, and the letter *y* represents which location in the bank that the GPIO pin is located. Usually, one bank will have 32 GPIO pins.

The GPIO pins are grouped into 3 groups of 32: GPIO0, GPIO1, and GPIO2. An individual pin can be referred to using the convention GPIOX_Y where X is its GPIO register and Y is its number within that register. *However, all references to a particular pin made in software instead use its absolute pin number.* in the following manner: Z = 32*X + Y where X is again the GPIO register and Y is the position within that register. A GPIO's *exact* pin number is calculated as x *32 + y.

For example, GPIO1[31] can also be called GPIO63. Pins *P8_1* and *P9_1* are located at the top right of each header. Notice that the left column of each header are all the odd pins while the right column of each header are all the even pins.

The GPIO configurations are located in the directory /sys/class/gpio. Before we use a GPIO pin, we will have to request it. To do this, type this command on the target board:

```
echo GPIO_NUMBER > /sys/class/gpio/export
```

For example, if you want GPIO63 to be available, type:

```
echo 65 > /sys/class/gpio/export.
```

What this command does is to create a folder inside the / sys/class/gpio folder called gpioGPIO_NUMBER (for our example, gpio65). After exporting the folder, you will see the device folder for the GPIO that you requested. For the example above, you will see gpio65 device folder inside your /sys/class/gpio folder. Inside the device folder, you will see device files:

- direction — Reads as either "in" or "out".
- value — Reads as either 0 (low) or 1 (high). If the GPIO is configured as an output, this value may be written; any nonzero value is treated as high
- edge — Reads as either "none", "rising", "falling", or "both".
  Write these strings to select the signal edge(s) that will enable a poll on the "value" file return. This file exists only if the pin can be configured as an interrupt generating input pin.
- active_low — Reads as either 0 (false) or 1 (true). Write any nonzero value to invert the value attribute both for reading and writing. Existing and subsequent poll support configuration via the edge attribute for "rising" and "falling" edges will follow this setting.

To set a configuration, you can use the command echo to these device files to configure a setting. For example, echo out > direction will configure the GPIO direction to be output.

Appendix B: Working with Buffers and Displays

A Simple Interface Between Two Terminals

This same example also appears on the class webpage under ***Lecture Code and Other Examples***

The following code opens a file and ports to two terminals. Data is written to then read from the file on one terminal, and sent, with additional data, to a second.

```
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *fp;                                  //  working file
   char c[] = "test string";                  //  test string
   char buffer[100];                          //  working buffer

   int fd0;                                   //  device / terminal0
   int fd1;                                   //  device / terminal1

   /*
    *   open file for reading and writing
    */
   fp = fopen("file.txt", "w+");

   /* open terminal ports for writing */
   fd0 = open("/dev/pts/1", O_WRONLY);
   fd1 = open("/dev/pts/0", O_WRONLY);

   /*
    *    write data to the file
    */
   fwrite(c, strlen(c) +1, 1, fp);

   /*
    *  Seek to the beginning of the file
    */
   fseek(fp, SEEK_SET, 0);
```

```
  /*
   *    Read and print the data from the file
   */
        fread(buffer, 1, 99, fp);
        printf("%s\n", buffer);


   /*
    *   Append the buffer contents to a test string and transmit to terminal0
    *   Transmit a second test string to terminal1
    *   Delay for 2 seconds
    */
         while (1)
         {
            dprintf(fd0, "guten abend %s\n", buffer);
            dprintf(fd1, "bonjour\n");
            sleep(2);
         }

   fclose(fp);
   return(0);
}
```