EE 474 Project 1
Introducing the Lab Environment

University of Washington - Department of Electrical Engineering

Blake Hannaford, Shwetak Patel, Allan Ecker, Brad Emerson, Anh Nguyen, Greg Lee, Jered Aasheim, Tom Cornelius, James K. Peckol

Introduction:

This project has two main purposes. The first is to introduce the C language, explore some of the important aspects of the language including pointers and multiple file programs, then practice simple debugging. The second is to introduce the Texas Instruments *Sitara, AM3358BZCZ100 BeagleBone Black* microcomputer (gees, that's a lot of numbers and letters), the associated Linux based development environment, and the world of embedded applications. OK, shortening the name a little, it's the *AM335x* microcomputer based on the ARM Cortex-A8. Wow, so, when is the final project due? Like a typical embedded system, we have a hardware piece and a software piece. This term, our major focus will be on the software piece.

You are strongly encouraged to read the entire project before trying to start designing or exploring. The best way to become familiar with any new piece of hardware is to take a tour of its features; the same holds true for learning a new piece of software or programming language. This is the approach that we will take.

We will take our first practical steps with the C language by starting with a working C program. We will make several modifications to the program then compile using our GCC compiler and run our versions. Next, we will work with several programs that compile and appear to work properly and yet contain several common errors. Using our GDB debugger in the Linux environment, we will try to identify and correct those errors. Then, we will build our project, compile it, download it, and run it on the target platform.

We will take the same approach as we move on to more complex projects in future. As we explore the processor, we'll begin with the documentation and discover some of the key features of this device. This is exactly how we will do it in future when we start to work with a new microprocessor for our company or in our graduate research.….then, on the second day…

Prerequisites:

Familiarity with data types, data structures, as well as standard program design, development, and debugging techniques. No beer until the project is completed. I really mean it.

Background

For us, this is a new microprocessor and development environment. Learning new components and tools is exciting and challenging. This is one of the fun parts of engineering. At the same time, sometimes it can be frustrating when we can't find any good documentation or the hardware or the software doesn't behave the way we want it to, or when, unfortunately, they behave exactly the way that we have told them to, or when we can't immediately find the answer to our questions and problems.

At this stage in learning about engineering and the engineering process, playing with the toys often seems to be the most exciting part…the documentation, formal design, and so on, is, well, often seen as rather boring. *Why do I have to go through all this…why can't I just go to the Internet and find something like the design and make a few modifications and be done?* In the real-world, the documentation and formal design are absolutely critical parts of all phases of the engineering development process. Doing it and doing it right can mean the difference between success or failure of a project, the malfunction of the system following delivery to the customer, or the possible loss of life while it is being used.

It's also very important to recognize and to remember that the answers to most interesting real-world engineering problems originate in our brains, discovered as we use our imaginations and knowledge to creatively apply the underlying theory and tools; they are reached through our persistent hard work and diligence. The solutions to challenging problems are not sitting, ready, and waiting for us on the Internet. The Vikings didn't discover North America by searching some ancient version of the Internet…they took risks…they explored and challenged. We didn't find the solution for making the first successful airplane flight, to putting someone on the moon, to making the first soft landing on Mars, to designing and programming the first microprocessors, or the discovery of the Higgs boson on the Internet and we won't find the answers to many of today's problems or the projects this quarter there either. We challenge you to explore, to think, and to make those discoveries.

Sometimes your instructor or TAs will have the answers and sometimes we won't. As we said in the opening, bear in mind that we're (starting to work with) learning the environment and tools too. This platform and development environment are complex and are new to all of us (but we're learning). If we all work together, to identify and solve problems as they occur, everyone benefits and we help to build useful material and for the next classes. The answers are there somewhere. Let's all work to find them.

Regardless of the specific platform/environment used, embedded systems development requires at least the following:

1. An understanding of the problem that we are trying to solve.
2. A target platform on which to develop the application.
3. A mechanism for programming the target platform.

Target platform is a term used loosely in industry to mean the actual piece of hardware that performs the computation and/or control – the place where our application (the software) will ultimately reside and run.

A target platform can be as simple as a single chip microprocessor (i.e. a PIC or an Arduino), or as complicated as a feature-rich single board computer (build around a multicore high performance processor, several special purpose processors, and possibly several programmable logic devices). Despite the differences in physical characteristics, the target platform's purpose is the same: to execute the software written by the developer. Here, and for the typical microprocessor based application, the software will be written in the C language. On many occasions, the C++ language may be used; for what are called hard real-time systems, but Java is rarely used. On a programmable logic device like an FPGA, it may be a mixture of C and Verilog.

The term *to program* here has two meanings. The first is the more traditional embedded sense and simply means writing software to control a given piece of hardware (in this case the target platform and any peripheral devices that may be connected to it – see above). In the embedded world, *programming* the target also means storing the executable into memory on the target. As the characteristics of the target platform can vary greatly, so too can the mechanism used to program the target platform.

Some development environments allow one to develop directly on the target platform (much like developing code on a PC), while others require that code be developed on another *host computer* and then transferred to the target platform. We call this transfer *downloading to the target platform*. Apart from the actual technique(s) used, every embedded system has to provide a mechanism for programming it.

One of the more difficult concepts to learn in C is the notion of pointers. Once you begin to see what they are and how they work, you'll find that they're actually rather straightforward. In reality, pointers are no more complicated than a variable type whose *value* is *interpreted* as the *address* of something in memory. This is exactly the same as interpreting a collection of bits (they're just bits) as an integer, a character, a floating point number, an interesting picture that you downloaded from one of those special websites on the Internet (don't download stuff here using BitTorrent – they have no sense of humor and will shut the university's network down), or treating all the data on your hard drive as a really really big integer. Weird, huh?

There is a very good explanation of pointers with accompanying drawings in your text. Once again, take the time to read through this material.

Development Environment

The development environment will be the Linux operating system. Sorry, no drones this quarter again…how about a train. This system is a feature-rich single board microcomputer equipped with an *ARM Sitara A335x* microprocessor which is a follow-on from the ARM 7.

As a point of interest, among the ARM's early ancestors was the BBC Microcomputer from Acorn Computers, Ltd. The official *Acorn RISC machine* project was started in October 1983.

From Wikipedia…

> *Acorn Computers was a British computer company established in Cambridge, England, in 1978. The company produced a number of computers which were especially popular in the UK. These included the Acorn Electron, the BBC Micro and the Acorn Archimedes. Acorn's BBC Microcomputer dominated the UK educational computer market during the 1980s and early 1990s, drawing many comparisons with Apple in the U.S.*

Although the company was broken up into several independent operations in 1998, its legacy includes the development of RISC personal computers. A number of Acorn's former subsidiaries live on today – notably ARM Holdings who are globally dominant in the mobile phone and PDA microprocessor market. Acorn is sometimes known as "the British Apple".

Interestingly, the ARM family of microprocessors are actually a design (piece of intellectual property) licensed by ARM to a variety of different companies to build.

Does anyone in industry really use these things? Well, let's see. By 2005, about 98 percent of the more than one billion mobile phones sold each year used at least one ARM processor. Two years ago, ARM processors accounted for approximately 90% of all embedded 32-bit RISC processors.

The Cortex-A8 is a RISC – *Reduced Instruction Set Computer* – processor that runs at up to 1 GHz. The version of the processor on the Sitara is the Texas Instruments ARM Cortex-A8 AM335x.

Some other features include:

- Memory
  - 256MB x16 DDR3L
  - 4KB EEPROM
  - 4GB embedded MMC
  - microSD connector
- I/O Communication
  - (69 max) General Purpose I/O pins
  - 6 UARTs
  - 3 I^2C, 2 SPI and 2 CAN interfaces
  - 2 USB ports
  - Ethernet interface
- JTAG Boundary Scan interface
- Debug Support
- Internal Peripherals
  - 6 Channel PWM Controller
  - 8 Channel 12-bit A/D

In this class, we will use many of these hardware features as we design and develop the various design projects; the details regarding each of the components will be given as needed.

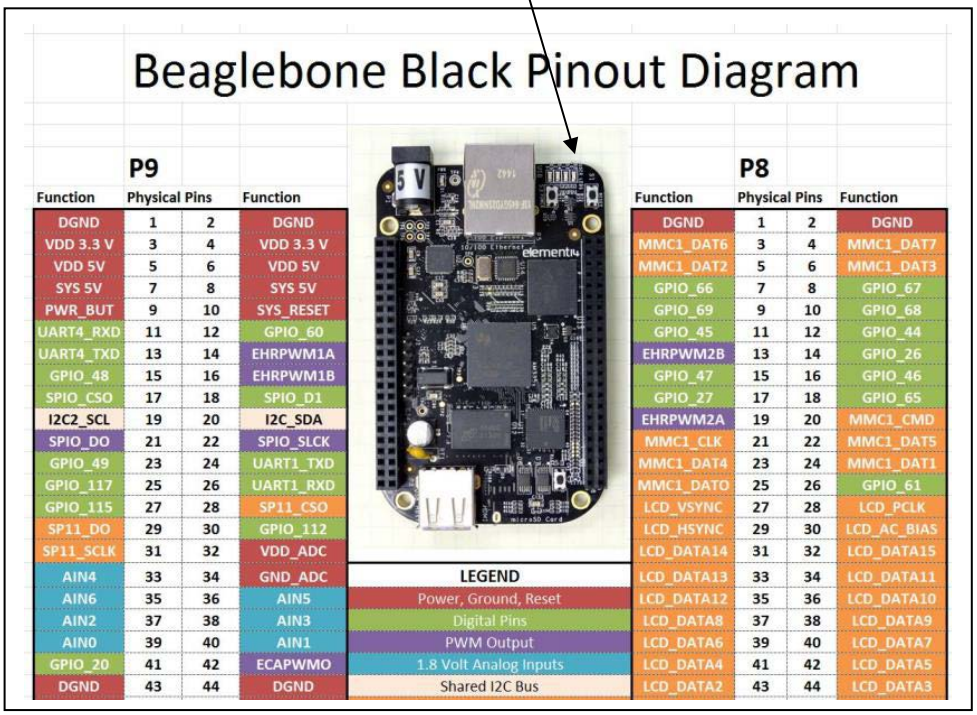
To program the Sitara Cortex-A8 target platform, a host PC is required. The design can be developed on the BeagleBone, compiled, and run on the device. Alternately, the code can be written off the BeagleBone then uploaded to the device, compiled, and run.

Project Objectives

- Introduce the C language, C programs, and the PC development environment: the basic C/C++ preprocessor, program structure, multiple file programs, pointers, passing and returning variables by value and by reference to and from subroutines, designing, compiling, and debugging on the target platform.
- Introduce the working environment: the embedded development environment, the target platform, host PC, equipment, etc.
- Learn a bit about the Texas Instruments version of the Cortex-A8 Processor.

Learning the Environment and Tools – The First Steps

As our first step, we will first work with the Beaglebone Black (BB) development tools under the Linux OS to create, build, and run an existing simple program on the, Sitara AM3358BZCZ100 chip. The pinout for the board is given in Figure 1 and the basic block diagram is shown in Figure 2. The ethernet port used to connect to a computer is on the bottom side of the board.



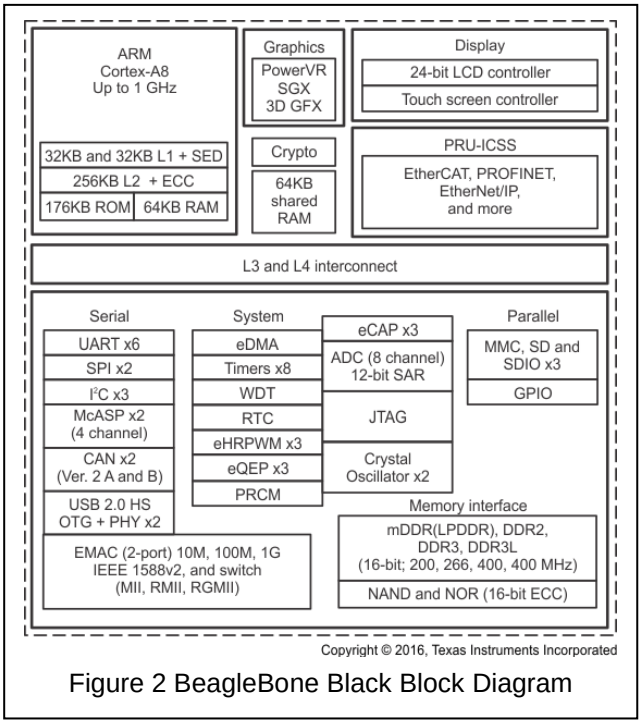


Figure 2 BeagleBone Black Block Diagram

The source code for several of the programs we will be using is in the same directory as the project assignment.

Working with the Tools

Connecting to the Board

We will primarily be communicating with the BegalBone Black board using SSH. SSH or secure shell is a common networking protocol used to remote login into other devices on a network.

Start by plugging in the ethernet cable to the ETH0 port on the board, see Figure 1 above. Also make sure the power module is plugged in to the board or you won't be able to actually communicate with it.

If you are working on a lab computer, open a terminal with the three key command (ctrl + alt + t) or click on the terminal icon in the taskbar (it should look like a small black screen) and the connect to the BeagleBone Sitara Cortex-A8 target platform by typing **ssh root@192.168.7.2** or **ssh 192.168.7.2 -l root** to connect to the board.

There is no password associated with the board so you can just press enter when asked for password.

This command will attempt to grab root access to the target 192.168.3.11. This is the default local IP address of the board.

If you are working on a Windows computer, start by installing the Putty application. You can download Putty from the following website:

http://www.putty.org

After installing Putty, to connect to the board, enter the host IP address and Port number as shown in Figure 3 then click

Open

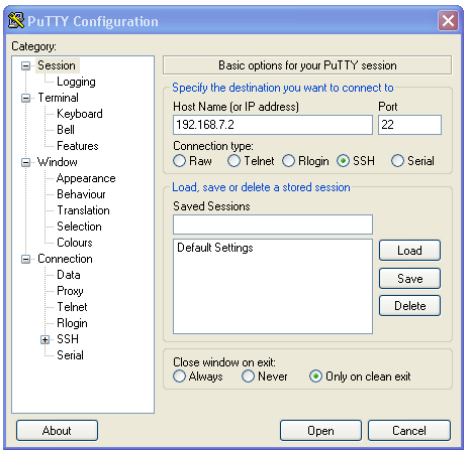


Figure 3 Putty SSH Screen

Once you click Open, there will be a window that asks you about a Putty security alert, just simply say yes.

Once a black window pops up, it will show a prompt:

```
login as:
```

Type root and hit enter; again, there is no password associated with this board, just press enter when (if) you are asked for a password.

The Linux Operating System

Once you are connected on the board, you will be inside the Linux operating system of the board. You can type all of the Linux terminal commands (such as **ls** this is a lower case l not the letter i (show files in current directory), **cd** (change directory), **echo** (print a string), etc.) inside the terminal.

There is a lot of good Linux cheat sheet, ***Linux-Commands.pdf***, on the class webpage under documentation.

When first connected, you will be in the **/home/** folder of the system. You can type **ls** to view all of the files and folders located in your current directory. At this stage, it's likely that this folder is empty; don't panic if nothing shows up.

To familiarize yourself with this system, you can type **ls** to list all of the files in the current directory, and **cd** to traverse into a named subdirectory or out of the current directory to another by typing **cd <folder>** to enter that folder. Don't type the angle brackets. Typing **cd ..** will move you up one level.

It is a good idea to get a general understanding of the file system on this board and you will be interacting with it for this lab and the rest of the quarter.. If you want to know the version of Linux that is currently running on the board, you can enter the command **uname -r**.

Transferring Files to the BeagleBone

Before we can start running programs on the board, we need to get them onto the board. We can create them directly on the board or we can transfer them there from an outside source. We'll start by learning to transfer files to the board which we will do using the SSH protocol.

1. Find the file hw0.c in the Lab1 folder and save it to a USB stick.
2. On the BeagleBone find the directory: **dev** and enter that directory.
3. Type the command ls to display a list of devices available on the board.
4. Insert the USB stick into the USB connector on the BeagleBone. Type ls again and find the identifier for the USB device that was added to the list of devices. The device name may be something such as sda or sda1.
5. Type the command (the name tempDir is arbitrary)

   ```
   mkdir tempDir
   ```

   to create a place to mount the USB stick on the board.
6. Type the command

   ```
   mount /dev/deviceIdentifier tempDir
   ```

to mount the USB stick.

7. On your computer, open up a terminal and create a directory called Project0 by typing:

   ```
   mkdir Project0
   ```

8. Go to that directory by typing

   ```
   cd Project0
   ```

9. Transfer the file hw0.c to the BeagleBone by typing the command

   ```
   scp <pathname to file location on USB stick> hw0.c
   ```

10. Compile your program using the gcc compiler by typing

    ```
    gcc hw0.c.
    ```

    This will produce a default executable file called a.out.

    Alternately, you can direct the compiler to produce an executable with a specified name by typing

    ```
    gcc hw0.c -o helloWorld.
    ```

    This will produce an executable file with that name.

11. Execute your program by typing

    ```
    ./a.out or ./helloWorld
    ```

    in the command line which should display

    ```
    Hello World I'm running…


    This is good exercise – bye bye.
    ```

Editing Files

The Linux environment on the BeagleBone supports the ***vim*** editor. Vim is what is called a moded editor. That is, it supports two modes, insert and command. The insert mode allows you to insert or modify your program, the command mode allows you to move around through your program, delete text, etc.

You can find the editor commands at the following URL and there is a copy of this file on the class webpage under documentation. Credit to the noted author.

http://www.radford.edu/~mhtay/CPSC120/VIM_Editor_Commands.htm

Building, Editing, and Running a Program

Let's now modify an existing project.

1. Find the file project1a-2017.c in the Lab1 folder, create a new working directory on the BeagleBone, and upload to that directory.
2. In the command prompt, enter vim project1a.c.

3. Annotate each line of the program to identify its purpose in the program and what it does (these really are different).
4. When you are finished, save the program by typing :w. Enter ^z to suspend the editor and return to the command line.
5. Compile your program using the gcc compiler by typing gcc project 1a.c.
6. Execute your program. There should be no errors – if there are, then bad copying.

The program should count in decimal according to the following pattern. Note that the following is actually 20 iterations through the program and the program's output will appear on only two lines on the console display….at time t0, the value 9 is displayed…at time t1, the values 9 8 are displayed and so forth.

```
The value of i is:

t0:   9

t1:   9 8

t2:   9 8 7

t3:   9 8 7 6

t4:   9 8 7 6 5

t5:   9 8 7 6 5 4

t6:   9 8 7 6 5 4 3

t7:   9 8 7 6 5 4 3 2

t8:   9 8 7 6 5 4 3 2 1

t9:   9 8 7 6 5 4 3 2 1 0

t10:    8 7 6 5 4 3 2 1 0

t11:      7 6 5 4 3 2 1 0

t12         6 5 4 3 2 1 0

t13:          5 4 3 2 1 0
```

```
t14:          4 3 2 1 0

t15:            3 2 1 0

t16:              2 1 0

t17:                1 0

t18:                  0

t19:

t20   9
```

7. Modify the program to parameterize the two *delay()* function calls in the two *for* loops so that they will support different user specified delays rather than the single hard coded value as they are now. Where will those values have to be specified? You decide.
8. Modify the program so that each of the two respective for loops is replaced by the following functions.

```
void f1Data(unsigned long delay1);

void f2Clear(unsigned long delay2);
```

9. Modify the program so that the delay parameter is passed into the two functions by pointer reference rather than by value.
10. Modify the program so that the two functions are replaced by a single function. The function should be able to be called with the character to be displayed and the value of the delay.
11. Modify the program so that the function you wrote in part 10 is in a separate file. Your program will now be composed of two files.

Working With the Debugger

Learning to use a debugger is an invaluable skill to develop if one is planning to do any serious software development work – (Wahoo, we actually have a debugger this quarter). The need to develop strong debugging skills is independent of whether one develops embedded or desktop applications.

Again from Wikipedia…

> The terms "bug" and "debugging" are both popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However

the term "bug" in the meaning of technical error dates back at least to 1878 and Thomas Edison. Further "debugging" seems to have been used as a term in aeronautics before entering the world of computers.

The Linux environment provides some rather effective tools that help you to debug your code running on the BeagleBone. One such tool is called the GDB debugger. The available commands are listed in two documents on the class webpage…

gdb_commands.pdf
gdb-tutorial-handout.pdf

Some useful features/abilities include:

- ✓ Setting, clearing, or listing breakpoints
- ✓ single-stepping through or over code
- ✓ running the next line or the next n lines
- ✓ running to a breakpoint or an error
- ✓ printing the value of local/global variables
- ✓ inspecting and changing the values of variables during runtime

For each of the above items, there are many ways to perform the desired action. A brief summary of the GDB commands for each action is as follows:

Breakpoints –

To set a breakpoint, move the cursor to the left hand window

b main - Puts a breakpoint at the beginning of the program

b - Puts a breakpoint at the current line

b N - Puts a breakpoint at line N

b +N - Puts a breakpoint N lines down from the current line

b fn - Puts a breakpoint at the beginning of function "fn"

d N - Deletes breakpoint number N

info break - list breakpoints

r - Runs the program until a breakpoint or error

c - Continues running the program until the next breakpoint or error

Bear in mind that a breakpoint is *not* the place in your debugging process where you throw up your hands and go for a beer.

Single-stepping –

f - Runs until the current function is finished
s - Runs the next line of the program
s N - Runs the next N lines of the program
n - Like s, but it does not step into functions
u N - Runs until you get N lines in front of the current line

Like breakpoints, single-stepping is not the process of slowly sneaking out when you've reached a breakpoint.

Local/Global Variables –

p var - Prints the current value of the variable "var"

bt - Prints a stack trace
u - Goes up a level in the stack
d - Goes down a level in the stack

Exit the Debugger –

q - Quits gdb

Let's now put the debugger to work.

1. Find the file project1b-2017.c in the Lab1 folder and upload it to your project directory in the BeagleBone environment.
2. The program will compile and execute apparently correctly, however, there is a problem with it.
3. Compile your program using the gcc compiler with the debug flag (g) set by typing

   gcc –g project1b.c –o project1b.
4. Follow the steps in the *gdb-tutorial-handout.pdf* in the documentation directory on the class webpage to start and bring your program into the gdb debugger.
5. Using your debugger, identify what the problem is and indicate how you found it with the debugger. Correct the problem and using your debugger, prove that you have, indeed, fixed the problem.
6. Find the file project1c-2017.c in the Lab1 folder and upload it to your project directory in the BeagleBone environment. The program will compile, however, it has a problem during execution.

   Using your debugger, identify what the problem is and indicate how you found it with the debugger. Correct the problem and using your debugger, prove that you have, indeed, fixed the problem.

Building Your Own Applications

Now that you have successfully run an existing application on the target platform, this next exercise will have you write your own program and then make a series of changes to it to gain practice in some of the techniques that we will have to use in our more complex applications.

Application 1

Using what you have learned from the example programs in the previous exercises, write a program that will display the letters: A B C D on the BeagleBone console and flash them together at approximately a one-second rate.

Application 2

Modify the program in Application 1 to print then erase the letters A then B then C then D at approximately a one-second rate.

Application 3

Modify the program in Application 1 to flash the letters A and C at a one-second rate and the letters B and D at a two-second rate.

Implement the delay as a function with the following prototype:

```
void delay(int aDelay);
```

Deliverables

A project report containing

1. The annotated source code for all applications for the BeagleBone Processor.
2. Representative screen shots showing the results of executing the applications on the PC screen. Representative screen shots means that you need to visually document that you got the programs to compile and run on your system and that your design was able to meet the project requirements and specifications.