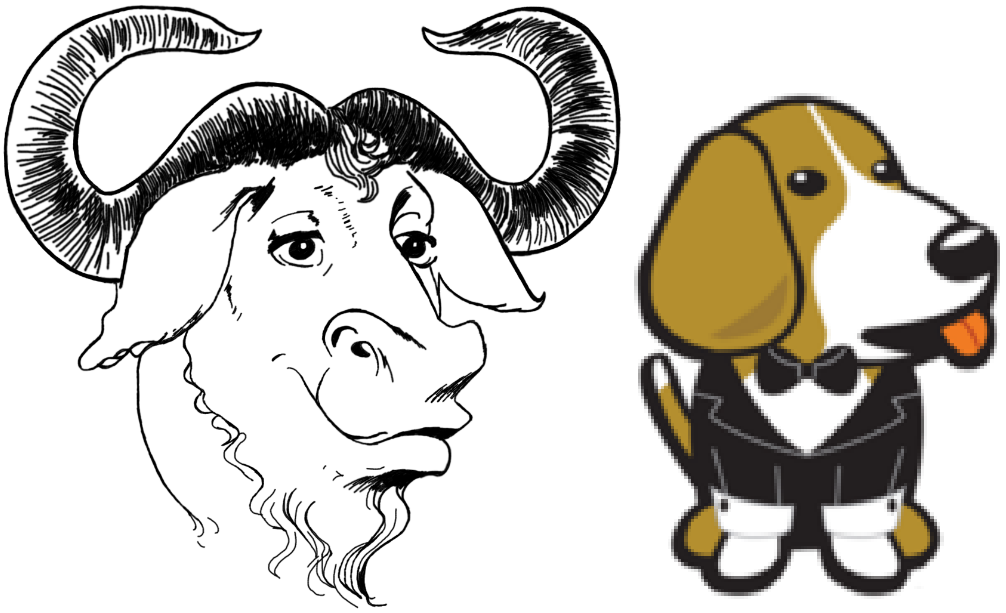
ECEN 5013:
Embedded Software Essentials



Project 1

Build System and Software Design

Diana Southard

Introduction

This is the first project for ECEN 5013 Spring 2016. It was assigned after several lectures detailing an introduction to embedded systems, version control, and the particulars of a GNU Makefile. One point repeatedly stressed during the lectures was that one of the course's ultimate goals was to learn how to create portable, robust software primarily for embedded systems but well-designed using design principles applicable across all platforms.

The purpose of this project was twofold. The first part was geared toward gaining experience developing our own makefiles specified for our desired compilation and build options. The second part of this project aimed toward writing writing standardized and portable code with the goal of creating an architecture-independent software capable of being compiled both on the native PC we used to write the code on and the BeagleBone Black we later transferred it to.

All source code for this project can be found on my Github repository located at https://github.com/dSouthard/ECEN5013_Homework with the final branch at which it can be validated against the project goals named `ecen513-project-1-rel`. The head of this branch points to the commit that I wish my code to be tested at. This point can be cloned by using the following git command:

```
git clone -b ecen513-project-1-rel https://github.com/dSouthard/ECEN5013_Homework.git
```

On the Github repo, you will find the following project directory structure

Directory Structure:

- makefile
- sources.mk
- project_xxx/
    - inc/
        - *.h
    - src/
        - *.c
- output/
    - *.o
    - *.asm
    - *.map
- **$(EXE)**

Future projects can be easily added into the directory following the same format. With the exception of the executable binary output, all project output files (*.o, *.asm, *.map, etc.) are created in the same output directory to simplify any cleanup operations.

Procedural Results

Version control played an essential role in this project. Since all files pushed to the Github repo were required to be directly related to the project, the first self-learned lesson was of how to create a gitignore file. This file was necessary to prevent accidentally pushing unwanted temporary or backup files to Github's repo.

The hardest part of the project was in understanding and creating a working makefile. Because the project specified that source and header files be put into specific directories, the makefile needed to be capable of searching through the directory structure in search of necessary target files instead of expecting to find everything it needed in the root directory.

This searching was accomplished in part by using the `vpath` tool. This tool specified a pattern for GNU Make to use when searching for a file. A separate makefile, `sources.mk`, contains the `vpath` search pattern allowing for this search pattern to be easily adjusted when future projects are added. The `vpath` search pattern was set up as follows:

```
vpath %.c $(PROJECT1_DIR)/src
```

Aside from using `vpath`, I instructed the compiler to add the header file includes directory by using the following flag: `CFLAGS += -I$(PROJECT1_DIR)/inc/`

The makefile had several required target requirements. Those requirements and how to invoke them are listed in the project's root-directory README file. The makefile used a shell command to detect the locally running OS in order to determine which compiler should be used when making the project [`OS := $(shell uname -m)`]. This allows the project to dynamically determine how to compile on either the native Linux PC on which it was written on the BeagleBone Board for which is is later transferred to.

The makefile also contained the login information for uploading files to the BeagleBone. Because this login information might change in the future, it was extracted into a macro located at the beginning of the file. as seen below

```
# BeagleBone Macros
BB_LOGIN = root@192.168.7.2
BB_DIR = /home/debian/bin/release
```

One of the requirements of this project was for any linker operation to provide a map file. This was accomplished by setting the linker flag as seen below:

```
LDFLAGS = -Xlinker -Map=$(OUTPUT_DIR)/$@.map
```

This flag, as stated earlier, redirected the outputted file into a common output directory.

The compiler flags were set as directed in the project directions. However, since we were overriding two built-in functions, I added a flag to tell the compiler to ignore built-in functions, specifically the two that I implemented on my own.

```
CFLAGS += -fno-builtin -fno-builtin-memcpy -fno-builtin-memmove
```

The project required that testing be done on the functions that were implemented. In order to facilitate the compilation and linking of the test objects, I added a makefile target named `test`, placing the required header and source files for these test objects in the same directories as the regular project files. Making this test target would result in the build of the **test_project** executable, capable of being run using the same `./` command as used in the normal project executable.

All required targets in the makefile were successfully created. In order to reduce unnecessary and verbose messages printed to the terminal during makefile build and other target actions, most makefile outputs were suppressed. However, whenever the project is built using the build command, the gcc size functionality will provide a build report of code size as required.

A screenshot showing the successful upload target working can be found in the Appendix under *A.1 Screenshot of successful upload to BeagleBone*. First, on the BeagleBone, the /home/debian/bin/ directory is checked to verify that no release folder has yet been created. Afterwards the upload command is called on the native PC. Then, the BeagleBone is again connected to in order to verify that the correct file was transferred over. There is an issue in this transfer: since the compilation of the executable takes place on the native PC which determines the compilation environment during runtime, the resulting binary executable file does not work on the BeagleBone though it does get successfully transferred over.

For the second part of the project, we were implementing several memory functions. Two of those functions were implementations of built-in C functions: memcpy and memmove. These functions moved a certain length of memory (unit32_t length) from a source (uint8_t * src) to a destination (uint8_t * dst).

There were 4 possible scenarios in a move such as this. The source could either be ahead in memory of the destination, or it could be behind. If the source were ahead of the destination, then the length to be transferred could either result in no overlapping between source and destination, or some overlapping of the source over the destination. Likewise, if the destination were ahead in memory of the source, there could be overlapping of the destination over the source depending on the length of memory transfer. This is illustrated below:

Case 1:
```
< -- src -- >
                  < -- dst -- >
         No Overlap
```

Case 2:
```
< -- src -- >
            < -- dst -- >
         Overlapping
```

Case 3:
```
< -- dst -- >
      < -- src -- >
         Overlapping
```

Case 4:
< -- dst -- >
                    < -- src -- >
        No Overlap

Per the project requirements, both memcpy and memmove were to correctly check if there were overlapped areas between the source and destination. However, further clarification from the instructor stated that only memmove was to correctly transfer between the two in the event of overlap, and that memcpy should in fact not check for overlap.

The implementation for the two functions was then as follows: `memcpy` would always just copy, byte by byte, from the specified source to the specified destination for the specified length from beginning to end.

Snippet from `mempy` implementation:

```
uint32_t index;
for (index = 0; index < length; index ++ ) {
      dst[index] = src[index];
}
```

However, `memmove` would always have to check to see if there were some form of overlap that would prevent that sort of copy from happening. Looking at the case illustration shown above, it becomes clear that the sort of left-to-right copying done for the for the `memcpy` function is acceptable in all cases over memory transfer except for Case 2.

In Case 2, if the blocks were to be copied from left to right, then at some point during the memory transfer the original source would have been overridden by the copy onto the destination, and data would be lost/corrupted. In this case, it is possible to instead copy from right to left (or from the end of the source memory block to the end of the destination block) and avoid this data corruption. The implementation of this is seen below.

Snippet from `memmove` implementation:

```
uint32_t index;
 if ((src < dst) && ((src + length) > dst))
             for (index = length; index > 0; index--)
                  dst[index-1] = src[index-1];
  else
      for (index = 0; index < length; index ++ )
                  dst[index] = src[index];
```

The implementation of `memzero` was very straightforward: simply place a zero (0x00) in for the specified length at the specified location.

Snippet from `memzero` implementation:

```
uint32_t index;
for (index = 0; index < length; index ++ ) {
        dst[index] = 0;
    }
```

The implementation of reverse made use of a temporary variable as well as traversing only half of the memory block. The function called for taking a pointer to a memory location and a length in bytes and reverse the order of all of the bytes. To implement this, I stored the beginning value in a temporary placeholder, then set the beginning value equal to the end value, then replaced the end value with the temporary placeholder's stored value. Because of this swapping, after going halfway through the memory block I was finished with the task.

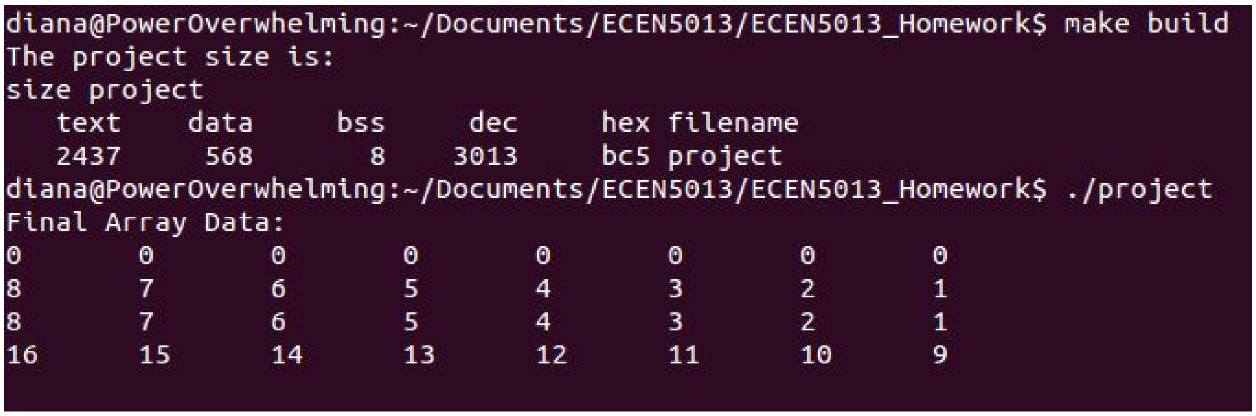
Snippet of `reverse` implementation

```
uint32_t index;
for (index = 0; index < length; index++, length--){
    temp = src[index];          // Grab character at start
    src[index]= src[length];    // Swap with character at end
   src[length]= temp;           // Change character at end
}
```

For all the implementations, I used standard data types so that the code would be architecture independent. I also did not use any dynamic memory.

To test all these functions, I initiated an array with data and then worked through the 4 possible cases with memmove and memcpy. As expected, memcpy lost data during any move of Case 2 type. However, memmove did not lost any desired data even in the same scenario of memory block overlap. Results of this test can be found in the appendix under *A.2 Results from tests on memory.h functions*.

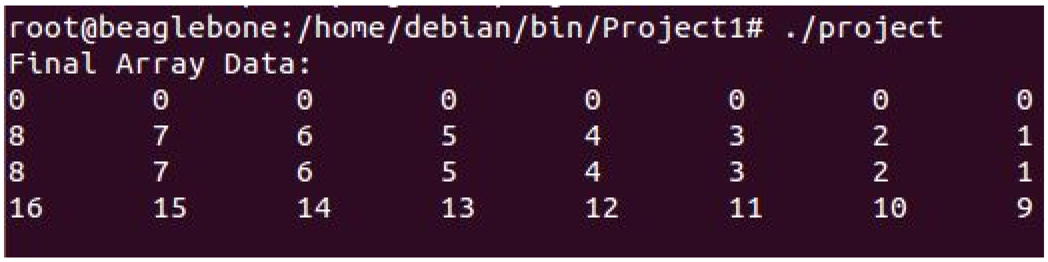
The implementation of project_1.c was to simply follow the project instructions. The final output from this function, as well as the executable file's size, is shown below:



```
diana@PowerOverwhelming:~/Documents/ECEN5013/ECEN5013_Homework$ make build
The project size is:
size project
   text    data     bss     dec     hex filename
   2437     568       8    3013     bc5 project
diana@PowerOverwhelming:~/Documents/ECEN5013/ECEN5013_Homework$ ./project
Final Array Data:
0       0       0       0       0       0       0       0
8       7       6       5       4       3       2       1
8       7       6       5       4       3       2       1
16      15      14      13      12      11      10      9
```

Project Size and Output

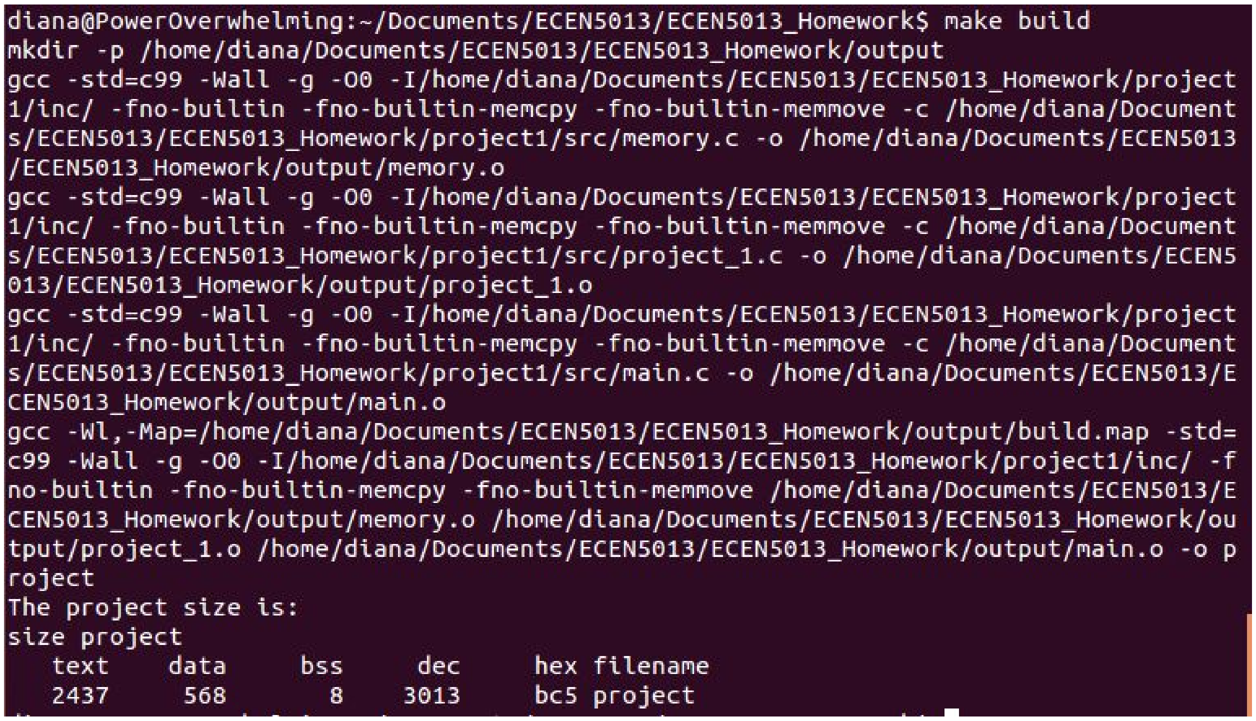
As was discussed above, the project successfully compiled on the native machine. That executable file could also be transferred to the BeagleBone. Additionally, once all of the project files were transferred over to the BeagleBone, the makefile target build could be successfully completed with the resulting executable showing the same output as previously seen.



```
root@beaglebone:/home/debian/bin/Project1# ./project
Final Array Data:
0       0       0       0       0       0       0       0
8       7       6       5       4       3       2       1
8       7       6       5       4       3       2       1
16      15      14      13      12      11      10      9
```

Project Output On the BeagleBone

In order to show that the linker is working, I rebuilt the project after adjusting the makefile to stop suppressing its output. The result is shown below.



```
diana@PowerOverwhelming:~/Documents/ECEN5013/ECEN5013_Homework$ make build
mkdir -p /home/diana/Documents/ECEN5013/ECEN5013_Homework/output
gcc -std=c99 -Wall -g -O0 -I/home/diana/Documents/ECEN5013/ECEN5013_Homework/project
1/inc/ -fno-builtin -fno-builtin-memcpy -fno-builtin-memmove -c /home/diana/Document
s/ECEN5013/ECEN5013_Homework/project1/src/memory.c -o /home/diana/Documents/ECEN5013
/ECEN5013_Homework/output/memory.o
gcc -std=c99 -Wall -g -O0 -I/home/diana/Documents/ECEN5013/ECEN5013_Homework/project
1/inc/ -fno-builtin -fno-builtin-memcpy -fno-builtin-memmove -c /home/diana/Document
s/ECEN5013/ECEN5013_Homework/project1/src/project_1.c -o /home/diana/Documents/ECEN5
013/ECEN5013_Homework/output/project_1.o
gcc -std=c99 -Wall -g -O0 -I/home/diana/Documents/ECEN5013/ECEN5013_Homework/project
1/inc/ -fno-builtin -fno-builtin-memcpy -fno-builtin-memmove -c /home/diana/Document
s/ECEN5013/ECEN5013_Homework/project1/src/main.c -o /home/diana/Documents/ECEN5013/E
CEN5013_Homework/output/main.o
gcc -Wl,-Map=/home/diana/Documents/ECEN5013/ECEN5013_Homework/output/build.map -std=
c99 -Wall -g -O0 -I/home/diana/Documents/ECEN5013/ECEN5013_Homework/project1/inc/ -f
no-builtin -fno-builtin-memcpy -fno-builtin-memmove /home/diana/Documents/ECEN5013/E
CEN5013_Homework/output/memory.o /home/diana/Documents/ECEN5013/ECEN5013_Homework/ou
tput/project_1.o /home/diana/Documents/ECEN5013/ECEN5013_Homework/output/main.o -o p
roject
The project size is:
size project
   text    data     bss     dec     hex filename
   2437     568       8    3013     bc5 project
```

Makefile Output for *build* command

Conclusion

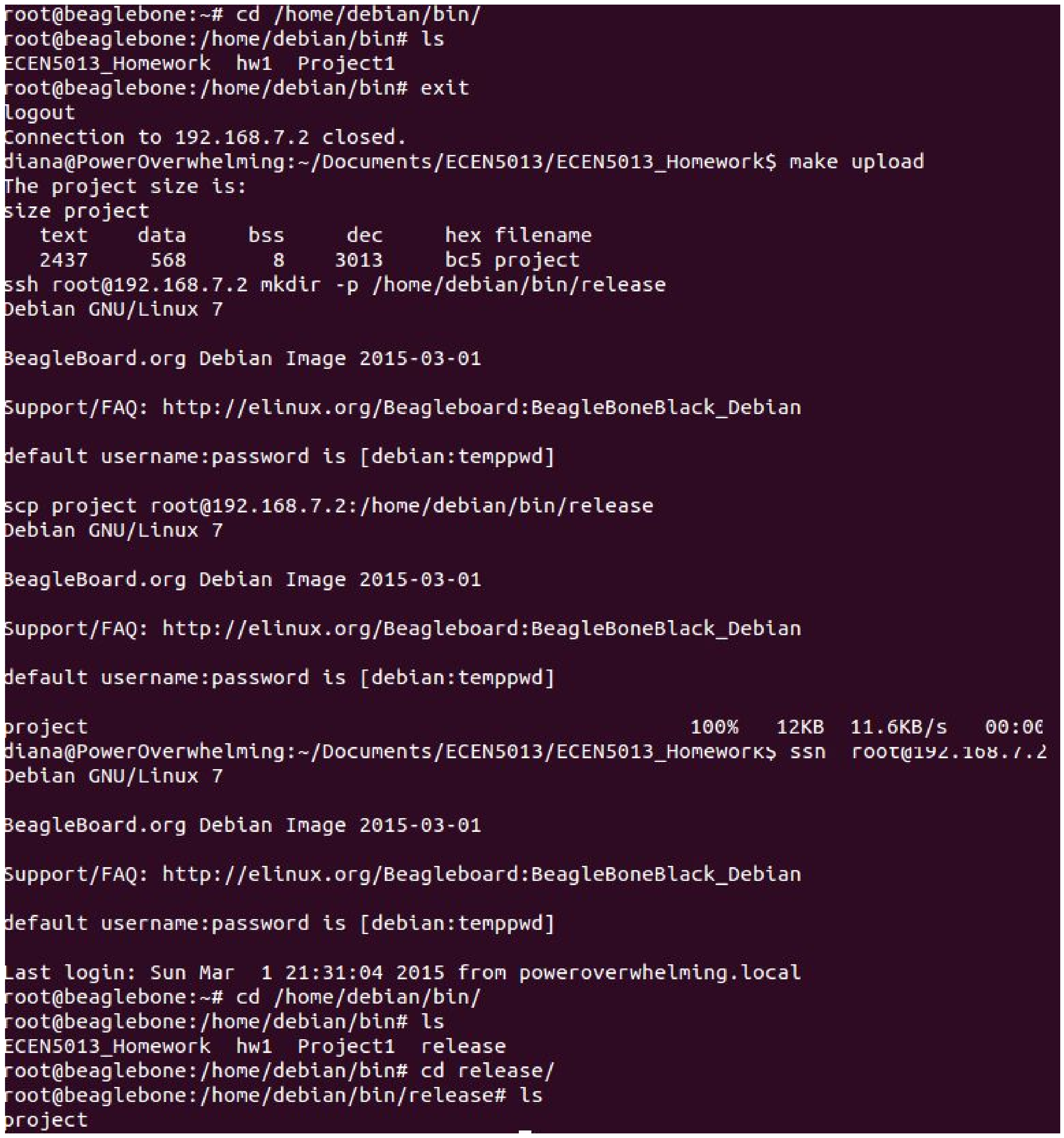
There was a lot of time invested in completing this project. Being that this was the first time I've created a makefile from scratch, there were many small, seemingly trite details that ended up taking much more effort than expected in order to figure out. For example, attempting to place a conditional statement inside a target's command block: as innocuous as that sounds, there were multiple different online sources saying multiple different ways in how to get this accomplished, none of which seemed to work!

In return, a lot of knowledge was gained in exactly how a makefile works, in the particular language syntax it uses. I have not had recent experience coding in C so the functions we were responsible for toko a little bit of refreshing as well.

This project was well worth it, for all the headache and frustration it caused. I feel confident that future projects will be easily assimilated into the final makefile and that this exposure will be reinforced further on in the course.

Appendix

A.1 Screenshot of successful upload to BeagleBone



```
root@beaglebone:~# cd /home/debian/bin/
root@beaglebone:/home/debian/bin# ls
ECEN5013_Homework  hw1  Project1
root@beaglebone:/home/debian/bin# exit
logout
Connection to 192.168.7.2 closed.
diana@PowerOverwhelming:~/Documents/ECEN5013/ECEN5013_Homework$ make upload
The project size is:
size project
   text    data     bss     dec     hex filename
   2437     568       8    3013     bc5 project
ssh root@192.168.7.2 mkdir -p /home/debian/bin/release
Debian GNU/Linux 7

BeagleBoard.org Debian Image 2015-03-01

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

scp project root@192.168.7.2:/home/debian/bin/release
Debian GNU/Linux 7

BeagleBoard.org Debian Image 2015-03-01

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

project                                           100%   12KB  11.6KB/s   00:0
diana@PowerOverwhelming:~/Documents/ECEN5013/ECEN5013_Homework$ ssh  root@192.168.7.2
Debian GNU/Linux 7

BeagleBoard.org Debian Image 2015-03-01

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

Last login: Sun Mar  1 21:31:04 2015 from poweroverwhelming.local
root@beaglebone:~# cd /home/debian/bin/
root@beaglebone:/home/debian/bin# ls
ECEN5013_Homework  hw1  Project1  release
root@beaglebone:/home/debian/bin# cd release/
root@beaglebone:/home/debian/bin/release# ls
project
```

A.2 Results from tests on memory.h functions:

Below are the results as seen printed from the test executable file. The tests are conducted on 32-length byte array. The data stored in each position of the array is printed to the screen from the beginning to the end of the array in rows of 8. A brief description is shown to describe what action was taken just before the array was printed.

```
Initial Array Data: Prepare for memcpy tests
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32


Case 1: src < dst, no overlap: memcpy test, 8 bytes from 0 to 16
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32


Case 2: src < dst, overlap: memcpy test, 16 bytes from 0 to 8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32


Case 3: src > dst, no overlap: memcpy test, 8 bytes from 24 to 8
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32


Case 4: src > dst, overlap: memcpy test, 16 bytes from 16 to 8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32
25	26	27	28	29	30	31	32


Re-initial Array Data: Prepare for memmove tests
```

```
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
```

Case 1: src < dst, no overlap: memmove test, 8 bytes from 0 to 16

```
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32
```

Case 2: src < dst, overlap: memmove test, 16 bytes from 0 to 8

```
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
25	26	27	28	29	30	31	32
```

Case 3: src > dst, no overlap: memmove test, 8 bytes from 24 to 8

```
1	2	3	4	5	6	7	8
25	26	27	28	29	30	31	32
9	10	11	12	13	14	15	16
25	26	27	28	29	30	31	32
```

Case 4: src > dst, overlap: memmove test, 16 bytes from 16 to 8

```
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
25	26	27	28	29	30	31	32
25	26	27	28	29	30	31	32
```

Memzero test from 16 to 32

```
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
```

Reverse test of entire string

```
0	0	0	0	0	0	0	0
```

```
0       0       0       0       0       0       0       0
16      15      14      13      12      11      10      9
8       7       6       5       4       3       2       1
```