Principles of Embedded Software

*Project 1 (**125** Pts) - Due Wednesday 10/4 (Midnight)*
Build Systems and C-Programming Practice

Overview

Description

In this project assignment, you will get more experience with Git Version control, develop some embedded C-programming code, and integrate your code with your build system. You will reuse your version control repository and add some new C-programming functions that manipulate memory. You will test your code on your host machine, but your code should compile for both the target platform and host platform. We will use the target platform in future assignments.

Outcomes

After completing this assignment, you will be able to:

1. Incorporate a C-program application into your Make and GCC build system
2. Write C-program functions that manipulate memory
3. Execute and test your application by simulating it on the host machine
4. Use git Version control to create a repository to version control code.

Guidelines

Projects can be done in teams of 1-2 people. All coding MUST adhere to the C-programming style guidelines posted on D2L. Failure to do so will result in point deductions. **No formal project writeup will need to be handed in for this assignment.** Any report questions/material should be turned in the repository in the form of code documentation and readme support in appropriate files in your repository.

You will need to turn in a code dump report. This should be a single file in a **.pdf** format that will turn into the Project 1 Dropbox D2L. This allows us to check for plagiarism. No other formats will be accepted. If you CD to your top-level directory, this one-liner command should copy all files and folder contents into a single file for you to submit.

```
$ find . \( -name "*.c" -o -name "*.h" \) -exec cat {} \;
```

We would also like you to submit of .zip file of your repository to the Dropbox. This will be used in Homework 4.

Resources:

- Programming Embedded Systems: Compiling, Linking and Locating (on D2L under Reference Material, Reading and Coding). Read the entire document - Programming_Embedded_Systems_Compiling_linking_locating

- The 5 Rule's of Makefiles - http://make.mad-scientist.net/papers/rules-of-makefiles
- GNU Make - https://www.gnu.org/software/make/
- GNU Make Manuals - https://www.gnu.org/software/make/manual/
    - HTML GNU Make Manual - http://www.gnu.org/software/make/manual/make.html
- GNU GCC - https://gcc.gnu.org/onlinedocs/gcc/

Version Controlled Repository

The project code must be turned in via a git repository. You can use your bitbucket/github account and create your own repository in your account. Some of you have already informed the TAs if you are working in a group. If you have not please send them an email with your partners name.

If you created a private repository, add the instructor team as a collaborator to the project. Here are the usernames for the instructor team:

Name	**BitBucket**	**GitHub**
Andrew Pleszkun	arp334	arp334
Vikhyat Goval	vigo9910	vikhyatgoyal
Shuting Guo	pilipalakwok	Embedded-Phelps

Name your repository something informative. Inside your version controlled folder you will need to place your build system files, scripts, embedded software and any other deliverables. You must make a **minimum of 5 commits for project 1 and there need to be commits from both members.** You cannot commit all your code at once. Your commit history must include commits of the following types:

- **First Commit:** This is your initial repository commit. This should contain a README of the repository
- **Design Stubs (C-programming):** This commits represents your project outline. This should include all defined prototypes, commented function descriptions, file descriptions, and empty function definitions. No function implementations
- **Feature Commits:** These commits represent your actual feature developments.
- **Bug Fixes:** These commits should represent any bugs that you found and fixed.

Place a git annotated tag with the name **project-1-rel** where you want the project to be graded. This tag **MUST** be placed before the due date of the project.

- https://git-scm.com/book/en/v2/Git-Basics-Tagging

During the lab practical, you will need to run multiple git commands for the instructor team to evaluate. If you are in a group, both team members must be able to answer command line git questions.

C-Programming Modules

Before you create a build system, you some source files are needed. Along with the current course homework, this section will give you some practice programming with pointers and manipulating data. You will write a few functions and incorporate other source files from the CMSIS libraries and the KL25z libraries. Details about setting-up your build system and **make** are given later in this document.

All pointer operations must use pointer math and pointer dereferencing. You must use pointers and lengths, no use of null terminated strings. **NO ARRAY NOTATION.** Why? Because you need to become extremely comfortable with pointers. Check pointers for NULL values before use. Initialize your pointers to NULL when no value is present.

Every function you write must work on all the three platforms we will be using (Linux VM, BBB, FRDM). In other words, these files must be architecture independent; not different files for different machines. All code **must** use standard data type sizes (int8_t, uint32_t, etc.).

The C-programming functions listed in this section must be written on your own. Many of these functions will be user in later assignments and projects. No online code may be used. Any plagiarized code will result in an honor code violation and a 0% for both group members.

File Requirements

For this part , you will add new files and new code to some existing files. All code provided must be your own work. You may move your makefiles or add multiple makefiles if you choose to do so. You may not use any online example code or prewritten code. Each Implementation file requires an associated header file. Each function needs a declaration with a good function comment including information on description, inputs, and returns. See the documentation section.

KL25z Platform Files

A zip of these files are available on D2L in the project folder under project1.zip

memory.c/memory.h

You need to add the following memory manipulation functions:

*uint8_t * my_memmove(uint8_t * src, uint8_t * dst, size_t length);*

1. This function takes two byte pointers (one source and one destination) and a length of bytes to copy from the source location to the destination.
2. The behavior should handle overlap of source and destination. Copy should occur, with no data corruption.
3. All operations need to be performed using pointer arithmetic, not array indexing
4. Should return a pointer to the destination (dst).

*uint8_t * my_memcpy(uint8_t * src, uint8_t * dst, size_t length);*

5. This function takes two byte pointers (one source and one destination) and a length of bytes to

copy from the source location to the destination.
6. The behavior is undefined if there is overlap of source and destination. Copy should still occur, but will likely corrupt your data.
7. All operations need to be performed using pointer arithmetic, not array indexing
8. Should return a pointer to the destination (dst).

*int8_t * my_memset(uint8_t * src, size_t length, uint8_t value);*

1. This should take a pointer to a source memory location, a length in bytes and set all locations of that memory to a given value.
2. All operations need to be performed using pointer arithmetic, not array indexing
3. Should return a pointer to the source (src).
4. You should NOT reuse the **set_all()** function

*uint8_t * my_memzero(uint8_t * src, size_t length);*

1. This should take a pointer to a memory location, a length in bytes and zero out all of the memory.
2. All operations need to be performed using pointer arithmetic, not array indexing
3. Should return a pointer to the source (src).
4. You should NOT reuse the **clear_all()** function

*uint8_t * my_reverse(uint8_t * src, size_t length);*

1. This should take a pointer to a memory location and a length in bytes and reverse the order of all of the bytes.
2. All operations need to be performed using pointer arithmetic, not array indexing
3. Should return a pointer to the source.

*int32_t * reserve_words(size_t length);*

1. This should take number of words to allocate in dynamic memory
2. All operations need to be performed using pointer arithmetic, not array indexing
3. Should return a pointer to the allocation if successful or a NULL pointer if not successful

*void free_words(int32_t * src);*

1. Should free a dynamic memory allocation by providing the pointer src to the function
2. All operations need to be performed using pointer arithmetic, not array indexing

conversion.c/conversion.h

These files should do some very basic data manipulation.

*uint8_t my_itoa(int32_t data, uint8_t * ptr, uint32_t base)*

1. **Integer-to-ASCII** needs to convert data from a standard integer type into an **ASCII** string.
2. All operations need to be performed using pointer arithmetic, not array indexing
3. The number you wish to convert is passed in as a signed 32-bit integer.
4. You should be able to support bases 2 to 16 by specifying the integer value of the base you wish to convert to (base).
5. Copy the converted character string to the u*int8_t** pointer passed in as a parameter (ptr)
6. The signed 32-bit number will have a maximum string size (Hint: Think base 2).
7. Function should return the length of the converted data (including a negative sign).

    a. Example my_itoa(ptr, 1234, 10) should return an ASCII string length of 4 or 5 if you include a null terminator.
8. This function needs to handle signed data.
9. You must place a null terminate at the end of the converted c-string
10. You may not use any string functions or libraries

*int32_t my_atoi(uint8_t * ptr, uint8_t digits, uint32_t base)*

1. **ASCII-to-Integer** needs to convert data back from an ASCII represented string into an integer type.
2. All operations need to be performed using pointer arithmetic, not array indexing
3. The character string to convert is passed in as a uint8_t * pointer (ptr).
4. The number of digits in your character set is passed in as a uint8_t integer (digits).
5. You should be able to support bases 2 to 16.
6. The converted 32-bit signed integer should be returned.
7. This function needs to handle signed data.
8. You may not use any string functions or libraries

*int8_t big_to_little32(uint32_t * data, uint32_t length)*

1. Convert an array of data in memory from a big endian representation to little endian.
2. Should return an error if the conversion fails for any reason.

*int8_t little_to_big32(uint32_t * data, uint32_t length)*

1. Convert an array of data in memory from a little endian representation to big endian.
2. Should return an error if the conversion fails for any reason.

debug.c/debug.h

Write a function called ***print_memory()*** so that you can enable/disable debug printing with a compile time switch. This compile time switch should be enabled with a VERBOSE flag in the makefile (**-DVERBOSE**). If enabled, the print_memory function should print as normal. If not defined, nothing should print.

*void print_memory(uint8_t * start, uint32_t length)*

1. This takes a pointer to memory and prints the hex output of bytes given a pointer to a memory location and a length of bytes to print.

platform.h

This file should help you create an independent layer allowing you to switch between platform specific lower-level functions. Printf functionality needs to be disabled for the KL25z platform and enabled for BBB and HOST platforms. A compile time switch can be used for this.

NOTE: Only for these first assignments will I advocate that you use printf.

main.c

Your main function will be very simple. You just need to call a function that is defined in the project1.c source file. However, you must use a compile time switch that wraps this function. This way we can

have a simple main() function that can switch to the project deliverable we wish to call by specifying the **-DPROJECT1** compile time switch. Future projects will use such switches to allow code from this project to be easily used.

```
#ifdef PROJECT1
     project1();
#endif
```

project1.c/project1.h

These files are provided for you and will provide some simple testing conditions for the project 1 functions. These are a part of the project1.zip

Documentation

You may want to try using a documentation system named Doxygen. This system uses known tags to indicate meta-information that can be easily searched and extracted from a code base. Whether you or not you use this system, your source files and functions should include this information.

Functions documented using Doxygen should include the following for each function:

- @brief - short description
- Long description
- @param - if applicable
- @return

Each file should also have the following documentation:

- @file - name of file
- @brief - short description
- long description
- @author
- @date

Build System using GCC and GNU Make

As we have learned in class, we can develop our own build system by directly by using make and creating our own makefile to specify our compiler options, build operations, and source files. You should strive to make our makefile as architecture and platform independent as possible. Command-line flags, can be used to switch between the native compiler or one of the cross compilers to build for one of multiple target platforms. In addition, we will use our host system and other target platforms for simulating and emulating our software on a host system.

Deliverables and Instructions

Write a makefile that can compile multiple source files and support three platform targets.

Your directory structure should look as follows:

- Top Level Folder project: Contains two subdirectories and some files
- 'platform' folder: Contains non-source files for build config
    - **MKL25Z128xxx_flash.ld** - The linker file you are to use for linking
- 'src' folder : contains multiple source files (*.c)
    - **Makefile** - The makefile you are to edit for the assignment
    - **sources.mk** - The source file you are to edit for the assignment
    - ***.c** - c implementations files
    - ***.s** - Assembly implementations files
- 'include' folder : contains the three directories of supporting header files
    - **common -** Contains common headers for both platform targets
    - **kl25z -** Contains KL25z platform headers
    - **CMSIS** - Contains ARM architecture specific headers "core_*.h"

Platforms and Flag Support Guidelines

You need to support three target platforms and their own specific compilers. These three platforms are the HOST, BeagleBone Black (BBB) and the KL25Z. The host embedded system will use the native compiler, **gcc**. The kl25z target embedded system will use the cross compiler, **arm-none-eabi-gcc**. The kl25z target embedded system will use the cross compiler, **arm-linux-gnueabi-gcc** or **arm-linux-gnueabihf-gcc**.

Compilation needs to work on both the Host machine and on the BeagleBone Black natively. On the host machine you should be able to cross compile both the KL25z and BBB builds. The BBB should be able to sync the repository and compile natively. The BBB does not need to support any cross compilers.

Use the **PLATFORM** keyword in the Makefile to conditionally assign the appropriate compiler flags, linker flags, and architecture flags. The target platform must be provided at the command line with the make command to set the platform for which you are compiling. Example:

```
$ make build PLATFORM=KL25Z
$ make build PLATFORM=BBB
$ make build PLATFORM=HOST
```

Upon completion of a build, provide a build report of code size using the gcc **size** tool. Note, you will need to select the right GCC Toolchain Size application. The output executable that gets built should be called **project1.elf.** This would be executed using dot-slash notation:

```
$ ./project1.elf
```

The makefiles need to have a minimum set of variables defined. Those are listed below:

- **CC** - Compiler that will perform the build (Native or Cross)
- **CFLAGS** - C-programming flags for gcc
- **CPPFLAGS** - C-Preprocessor flags for gcc
- **LDFLAGS** - Linker Flags
- **PLATFORM** - The target platform you are compiling for (Platform Specific)
- **SOURCES** - The list of sources files that will need to be compiled (Platform specific)

- **INCLUDES** - The list of include directories
    - Hint: Use the -I flag

Generate the files for each complete build:

- **project1.map** - Map file for the full build
    - Use the -Map=<FILE> option
- ***.d**- Dependency Files for each source file (main.dep, memory.dep, etc)
    - Use the -M option
- ***.o** - Individual object files (main.o, memory.o, etc)
- **project1.elf** - Output Executable file

For compilation and linking a variety of flags need to be supported. These will include general flags, platform specific flags and architecture specific flags. These are listed below:

- General Flags (Both Platforms)
    - -Wall
    - -Werror
    - -g
    - -O0
    - -std=c99
- Platform Specific Flags (KL25Z/HOST)
    - Linker File Flag **(KL25z)**
    - Platform Target Compile Time Switch **(All)**: HOST, KL25Z, BBB
- Architecture Specific Flags for ARM Architectures (KL25z)
    - -mcpu=**???**
    - -mthumb
    - -march=**???**
    - -mfloat-abi=**???**
    - -mfpu=fpv4-sp-d16
    - --specs=nosys.specs

You may **NOT** use wildcard searches for finding source files. You must directly define the files you want to compile and the directories you want to include based on the platform. To do this, you will need to define the sources and include list differently based on the target platforms. Use a conditional check to switch between these. The example below provides a hint on how to do this:

```
ifeq ($(PLATFORM),HOST)
      CC = <Add the appropriate compiler>
      # etc
else
      CC = <Add the appropriate compiler>
      # etc
endif
```

Build Target Guidelines

You will need to support a number of build rules and target files. Any rule with a prerequisite list must have only dependent files and dependent targets listed. For any prerequisite that is in another build target prerequisite list, that target dependency needs to execute those rules before running the initially

provided target. Finally, you do **NOT** need to go from source file to preprocessor file to assembly file to object file to relocatable file to executable file for this build. You can have build rules that directly compile a file.

Below are the targets which need to be supported in the makefile:

- %.i
  - Generate the preprocessed output of c-program implementation files (–E flag).
  - You can do this by providing a single target name:
    - Example: `$ make main.i`
- %.asm
  - Generate assembly output of c-program implementation files (-S flag)
  - You can do this by providing a single target name:
    - Example: `$ make main.asm`
- %.o
  - Generate the object file for all source files (but do not link) by specifying the object file you want to compile.
    - Example: `$ make main.o`
- compile-all
  - Compile all object files, but **DO NOT** link.
  - Needs to have .PHONY protection.
    - Example: `$ make compile-all`
- build
  - Compile all object files and link into a final executable.
  - Needs to have .PHONY protection.
    - Example: `$ make build`
- clean
  - This should remove all compiled objects, preprocessed outputs, assembly outputs, executables and build output files.
  - Needs to have .PHONY protection.
  - This includes but is not limited to .map, .out, .o, .asm, .i etc.
    - Example: $ make clean

Testing Your System

You can intermittently test your build system targets as you write them. A few examples are listed as follows:

```
Example 1:          $ make memory.o PLATFORM=KL25Z
Example 2:          $ make build PLATFORM=HOST
Example 3:          $ make main.asm PLATFORM=HOST
Example 4:          $ make memory.i PLATFORM=HOST
```

Each of these build commands will produce one or more output files. To show the generated files, use Linux's **ls** command to list the current files in the directory. You should be able to remove those files

with the “make clean” command. Running the **ls** command once more, should show that these files have been removed. For example:

```
$ ls -la
$ make clean
$ ls -la
```

Finally, the host code should be able to run natively on your system. You can test that this file works properly by running **./project.elf.** The cortex build will not be able to run natively. This should throw an error if you try to run natively. This should output the string ***“aXy72_L+R”***.

Extra Credit

Library (2 Pts)

Add support into your build system to build the Homework/project sources files into a library/archive for project1. This should not include the main.c file. Then you must show that you can produce this archive, link with that library and run those functions. That target must be called:

- build-lib

This should generate a library of your platform independent functions into an archive called libproject1.a

Linker (2 Pts)

Directly invoke the linker and not through GCC. This requires extra options and includes to get it LD to work.

Project Demonstration Procedures

Your project will be graded during a demo session with the instructor team. The following items will be asked of your group for the project during this demo:

1. Students will be asked to run certain commands and observe the output. These commands can include coverage on Linux, Git, Make, or GCC.
2. Documentation on your source files, makefiles, and functions will need to be shown (just open them to view during the demo).
3. Successful running of your compiled files from project1() function without any errors.
4. The instructor team will ask you to add a new feature during the demo.

Plan the demonstration practical to take 20-30 minutes per group. Students in groups will be asked to work on tasks in parallel.

Campus students will be asked to meet in person with the instructor team. Distance students will use Screen share and a Zoom session to demo.

A poll to signup up for a demo time will be provided here: [To Be posted]