ECE 362 Lab Experiment 1: “First Blinks”

1.0 Introduction

Traditionally, the "first steps" taken to verify that a microcontroller works and that you know how to use it, is to make a light blink. In this introductory exercise, you will use an integrated development environment (IDE) to interact with a microcontroller. You will type in, compile, and debug a few programs that we have prepared for you. One of the programs will cause the built-in LEDs of the STM32F0DISCOVERY development board to blink. You will also write a few simple assembly programs.

2.0 Objectives

1. Type in, compile, and debug simple assembly language programs.
2. Analyze the operation of simple assembly instructions.
3. Practice writing assembly language routines which perform simple tasks.

3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

4.0 Prelab

Read section 5 of this document for background information and complete the prelab on the course website.

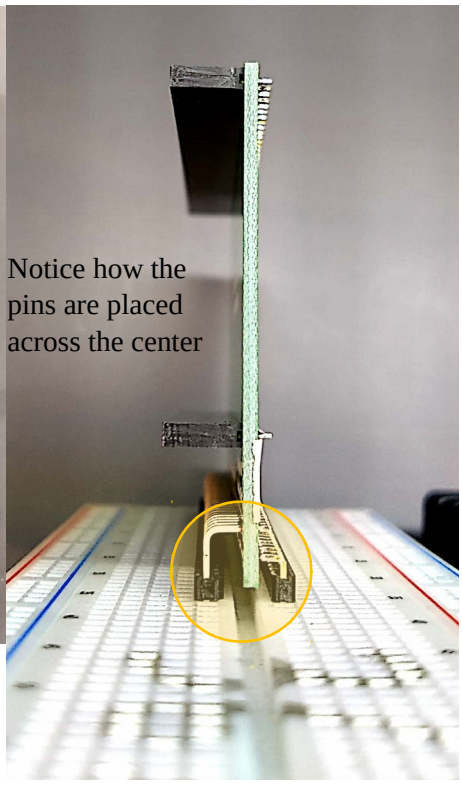
4.1 Assembling your board

Your development kit contains a collection of many parts, however for this lab we will use only:

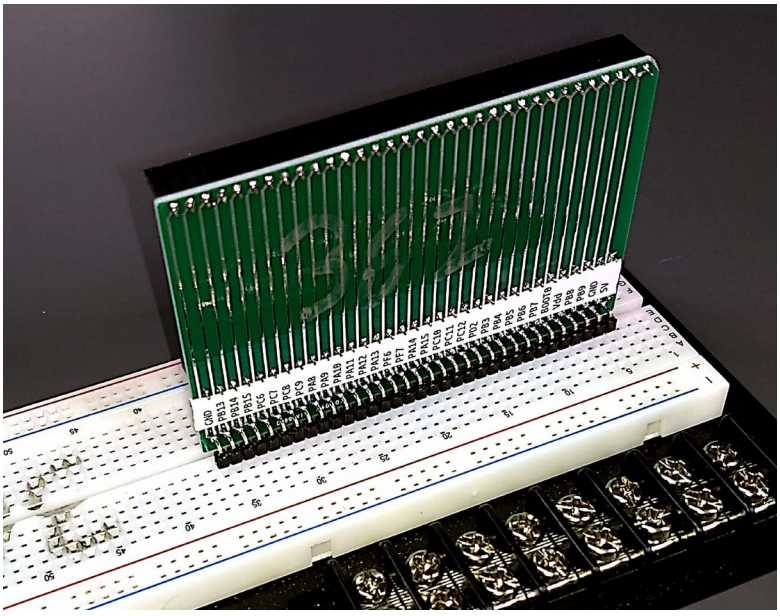
- the SMT32F0 discovery board
- a mini-b USB cable
- a breadboard, and
- the STM32F0 breadboard adapter.

To help you with assembly, images are shown so that you can follow along. It is vital that you follow the following steps, so that you assemble the development kit correctly without damaging the board.

(Figure 1) STM32F0 breadboard adapter



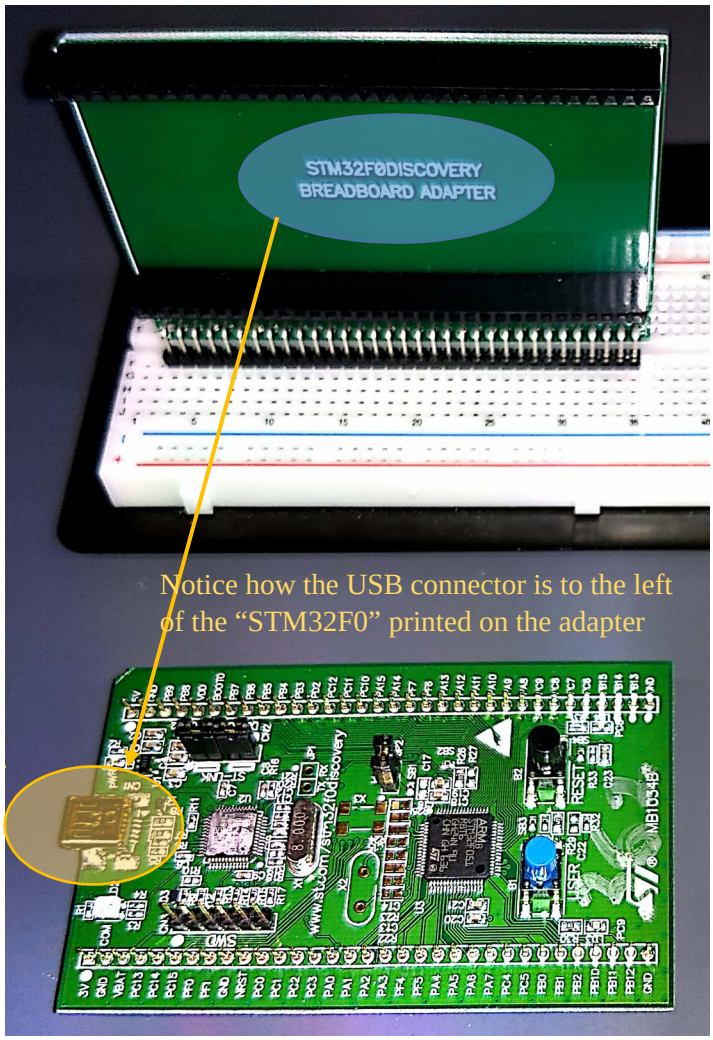
(Top, Figure 2) Adapter on the breadboard

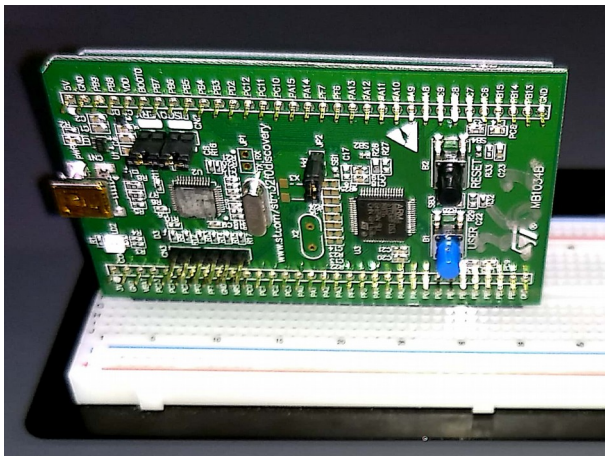


(Left, Figure 3) After placing the adapter on the board

4.1.1 Orienting an placing the adapter

The image above (Figure 1) shows the breadboard adapter for the STM32F0 board. We will place this on the breadboard. It takes a significant amount of force so to push it into place. Make sure to place the pin headers of the adapter across the center divider (a.k.a DIP support) as shown in Figure 2. Once this is done, your assembly should look like Figure 3.





(Top, Figure 5) Development board after being inserted into the adapter

(Left, Figure 4) Orientation of the board and the adapter

4.1.2 Placing the development board

The next step in assembly is inserting the development board into the adapter. It is very important to note the orientation in Figure 4 and Figure 5. Place the board and the adapter as shown in Figure 4. Keeping the same orientation, insert it into the adapter, and you should end up with Figure 5.

Your STM32F0DISCOVERY board comes with an extra printed circuit board with nothing but plated holes on a .1" grid. **You should remove that from the STM32F0DISCOVERY before plugging it in to the adapter board.** Put it back in your kit box. You may want to use it later for prototyping.

4.1.3 Placing labels on pins

Now that everything is assembled, place the label on the backside of the adapter as seen in Figure 6 making sure that the 5V label ends up near the USB connector. Finally connect the development board to the computer and you are ready to go!

Figure 6. Placing labels on the pins, notice the USB connector on the right and the 5V pin

5.0 Background

5.1 Programming, Compilation, and Assembly

Microcontrollers can be programmed in a manner similar to that of a general purpose computer system. However, due to memory size constraints of most microcontrollers, the software development environment must be hosted on a separate computer system. A program written in a high-level language such as C or C++ can be translated into assembly language by a compiler. In the case where the host system CPU differs from the target system CPU, this translator is called a cross-compiler. The compiler used for ECE 362 is the same as the one you used for your introductory programming classes, gcc. This version of gcc has been configured as a cross-compiler to produce assembly language specifically for the ARM Cortex M0 architecture. Assembly language is changed into machine code (also known as object code) by an assembler. In the past, you have seen gcc apparently produce object code directly as object files with a “.o” suffix. It does this by invoking the assembler automatically in a way that you never see the intermediate assembly language. GCC also knows how to directly accept assembly language files

with “.s” suffixes and pass them to the assembler to generate an object file without ever needing initial C code.

An object file contains raw machine instructions and other data in chunks known as segments. Two of these segments are the text and data segments. The object file format is known as the Executable and Linking Format (ELF). An ELF object file produced for your C or assembly language program cannot be directly executed by a computer because it lacks various hidden setup and configuration procedures. For instance, there is a special procedure that is responsible for calling the main() function as well as setting up its arguments. These procedures are held in other object files. These object files are combined into an executable by a linker. The linker searches each object file to find references to symbols that remain undefined in that file. For instance, object file containing startup code refers to main(), but that function is not defined in that file. The linker combines the text, data, and other segments of multiple object files so that all of those references are defined. The resulting file is called an ELF executable which can be executed directly by a general purpose computer. Unfortunately, the microcontroller on your development board has neither an operating system nor any kind of loader to interpret an ELF executable. A final step of the program compilation, assembly, and linking process involves converting the ELF executable into a raw binary image of the desired memory configuration that has no segments. This binary file is written directly into the flash ROM of your development system. All of the steps of compiling, assembling, linking, and converting are automated and hidden by the integrated development environment. You might look at the project directories that are created in the course of this lab exercise to find the object files, executables, and binary images that are produced. The resulting flow of files for a project might look as follows:

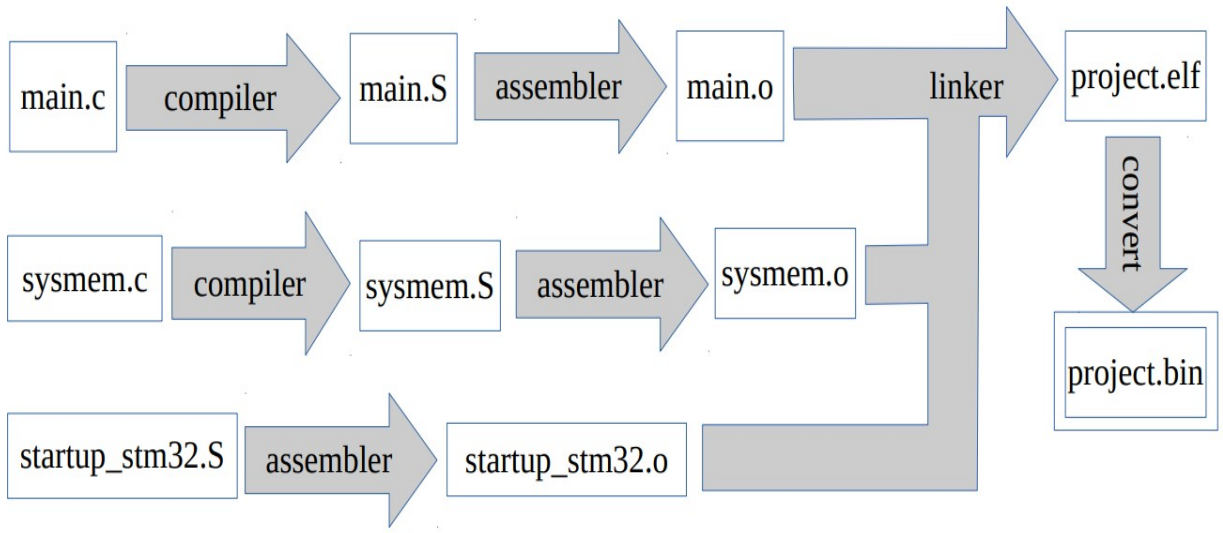


Figure 7. Flow of files during a project build

6.0 Experiment

This lab has 4 segments where you are to type in assembly code. Before you begin you must create a project (see 6.1), then you must use the “main.s” template file available on the website and add your code in the corresponding section (see 6.2 and 6.3).

6.1 Create a project

You should have created a project in Lab 0, you can use the same project or create a new one. If you are creating a new project, refer to Lab 0 on steps to create a new project. Even though it is not required for this lab, it is highly recommended that you create a project with the Standard Peripheral Library (StdPeriph). This will be helpful in future labs.

6.2 Writing Code Segments

With some familiarity of the STM32F0 assembly instruction set developed from the lectures, it's now time to put these skills into practice, in the form of developing functional code segments. These code segments should perform a specified function, subject to provided constraints. This is an important exercise for students, as they will often have to write code which interfaces with code written by others in a specific and well-defined way. For your assistance, a template file is available on the ECE362 course website (sections are reproduced below, for reference). The description of each program is contained in the header section. Download a copy of the file and “fill in” the code segments where indicated. For testing, run the code segments within Eclipse, similar to that done in the previous experiment. Students are to demonstrate code segments to their lab instructors in order to receive credit. Note: Each of the code segments you create will be written in the same main.s file. Write each code segment in the space indicate by the comment **“/* Student code goes here */”**.

6.3 Some Assembly Required

Since the purpose of ECE362 is to build a comprehensive understanding of microprocessors, the first few labs will require you to program in assembly language. Eclipse relies on the GNU C Compiler (gcc), so assembly language programming in this course will focus around the GNU Assembly language, as opposed to ARM Assembly, used by other compilers and assemblers. In the C/C++ Editing Perspective, find the file main.c, located in the src/ folder of the project being used in this experiment, in the Project Explorer pane. Right click on the file, and select **Resource Configurations >> Exclude from Build**. When prompted for configurations, exclude the file from all configurations. This instructs the compiler to ignore this file during the compilation process.

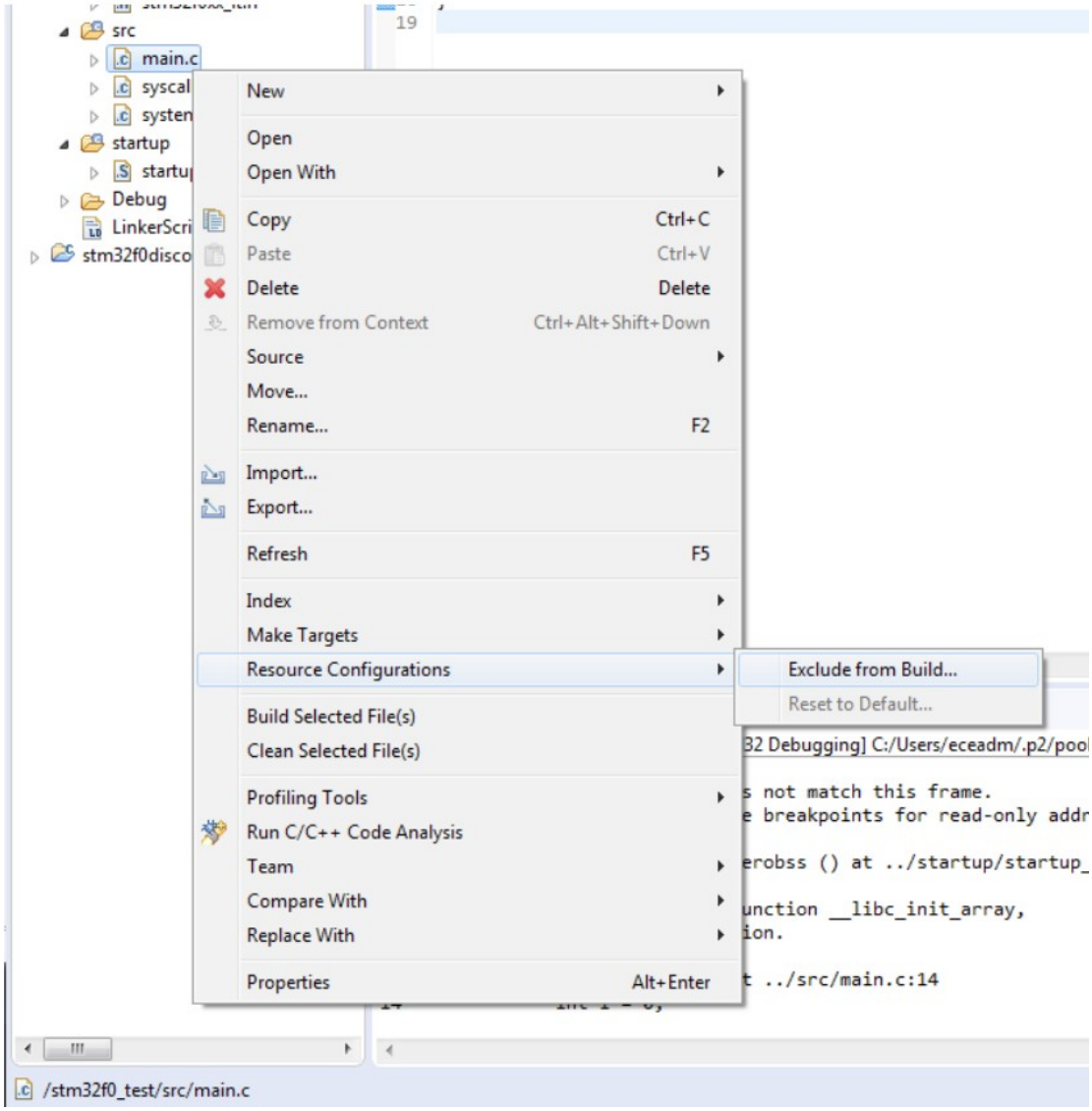


Figure 8. Excluding a file from the build

In place of main.c, a new assembly source code file must be created. Right click on the src folder in the Project Explorer pane, and select **New >> File**. Call the file main.s. Copy the contents of the main.s file from the ECE 362 website.

6.4 Arithmetic operations

```
/*******************************************************************************
* Code Segment 1
* Arithmetic operations.
*
* Description:
*   Write an assembly program to set the following register values using the
*   MOVS instruction.
*      R0 = 0x11
*      R1 = 0x33
*   Set R2 to be the sum of R0 and R1.
*   Set R3 to be the difference of R0 and R1.
*   Set R4 to be the product of R2 and R3.
*
* Useful Instructions:
*   Look at the ADD, SUB, and MUL instructions (it is up
*   to you to look up the mnemonics for these instructions)
*
*******************************************************************************/
codeSegment1:
            /* Student code goes here */

            /* End of student code */
```

This is the first code segment that you will need to complete for this lab. For this section you will write an assembly program to set a particular bits in registers R0 – R4.

Please type your code under the *set_bit: /*Student code goes here*/*. **DO NOT** modify code outside the */*Student code goes here*/* and */* End of student code */* section. Please use **6 instructions** to implement this code. Do not use something like “LDR R0,=...” to implement this code.

6.5 Initializing values

```
/*****************************************************************************
* Code Segment 2
*
* Description:
*   Initialize r0 to 0xFFFFFFFE (a lower)
*   Initialize r1 to 0x7FFFFFFF (a upper)
*   Initialize r2 to 0x2        (b lower)
*   Initialize r3 to 0x0        (b upper)
*   When you write this code, do not use “LDR R0,=0xFFFFFFFE” to load R0.
*   Hint: for initializing r2 and r3, use the MOVS instruction.
*   Hint2: Use the SUBS instruction to set R0.
*   Hint3: Use the LSRS instruction to set R1.
*
* Useful Instructions:
*   add, subtract, move, logical shift right, and logical shift left
*   (it is up to you to look up the mnemonics for these instructions)
*
*****************************************************************************/
codeSegment2:

        /* Student code goes here*/

        /* End of student code*/
```

To complete this section you will have to write your assembly code **under codeSegment2** (refer the main.s template file on the website). The code should initialize the values of R0 to 0xFFFFFFFE, R1 to 0x7FFFFFFF R2 to 0x2, and R3 to 0x0.

Hint: A smart way to accomplish this with less code is to initialize one of R2 and R3 then initialize R0. (How are values of R2 and R0 related? Can you perform one operation on R2 to get value of R0?). Then use R2 to build R3.

The “move”, “add”, “logical shift right” and “logical shift left” (it is up to you to look up the mnemonic for these instructions) are some useful instructions that you may use to accomplish this step. Note: You may not need to use all the instructions, you may be able to accomplish this with only a subset of the stated instructions.

Question 6.5.1: Why is it not possible to use “movs r2, #0x7fffffff” or “movs r0, #0xfffffffe”? Provide a brief explanation in your post-lab write-up.

6.6 Adding two 64-bit numbers

```
/*****************************************************************************
* Code Segment 3
*
* Add two 64 bit numbers:
*
* Description:
*   Write a program to add 2 64-bit numbers.
*   The operation is a + b, where each a and b are 64 bits.
*   Given that:
*     a's lower 32 bits are stored in r0,
*     a's upper 32 bits are stored in r1,
*     b's lower 32 bits are stored in r2,
*     b's upper 32 bits are stored in r3,
*   use add with carry instruction to propagate the
*   carry from the lower 32 bit's (lower word) to the upper word
*
* Useful Instructions:
*   add and add with carry (it is up to you to look up the
*   mnemonic for these instructions)
*   Think: ask yourself why you need the <s>
*
*****************************************************************************/
codeSegment3:
        /* Student code goes here*/

        /* End of student code */
```

To complete this section you will have to write your assembly code **under codeSegment3** (refer the main.s template file on the website). Assume that registers are already set to the right values. You will write an assembly program to add two 64-bit numbers.

Since the ARM Cortex-M0’s registers have a bit width of 32, a single add instruction is not sufficient to add two 64 bit numbers. The operation is “a” + “b” where both “a” and “b” are 64 bits. The lower 32 bits of “a” are stored in R0, and the upper 32 bits of “a” are stored in R1. “b's” lower 32 bits are stored in R2, and upper 32 bits are stored in R3. The result of the addition should be stored in R0 and R1, where R0 stores the lower 32 bits of the result and R1 stores the upper 32 bits of the result.

Even though this looks daunting, rest assured this can be achieved in **2 instructions**.

Question 6.6.1 in the previous section we had you initialize the values of the registers, the values we choose were tailored so that we could test if you got this section right. What should be the result of the addition?

6.7 “First Blinks”

For this section, type in the following code near the top of your main.s file right under the first comment about */* Student code goes here*/ /* Put the .equ here*/.*

```
/*****************************************************
* ECE 362 Lab expirement 1
* "First Blinks" with assembly
*****************************************************

/* Student code goes here */
/* Put the .equ here */
.equ    RCC,        0x40021000
.equ    AHBENR,     0x14
.equ    GPIOCEN,    0x00080000
.equ    GPIOC,      0x48000800
.equ    MODER,      0x00
.equ    ODR,        0x14
.equ    PIN8MOD,    0x00010000
.equ    PIN8,       0x00000100
/* End of student code */

.global main
main:
```

Figure 9. Some .equ initializations needed for blinking the LED

Next type the assembly code shown on the next page (Figure 10) **under codeSegment4.** Some of the code shown below uses syntax and instructions not yet covered in class, but rest assured that it will be demystified soon.

When you run this assembly program you should observe the blue LED blinks. Note this will work if and only if you got 6.4 right, if you are having trouble with this section seek help from a TA.

Question 6.7.1: What happens if the .global directive is omitted? Remove it and rebuild your project to find out. Provide a brief explanation in your post-lab write-up.

```
103 codeSegment4:
104     /* Student code goes here */
105     // Enable clock to the GPIOC peripheral
106     ldr     r2, =RCC
107     ldr     r3, =AHBENR
108     ldr     r1, =GPIOCEN
109     ldr     r0, [r2,r3]
110     orrs    r0, r1
111     str     r0, [r2, r3]
112
113     // Set the mode of pin8 as output
114     ldr     r2, =GPIOC
115     ldr     r3, =MODER
116     ldr     r1, =PIN8MOD
117     ldr     r0, [r2, r3]
118     orrs    r0, r1
119     str     r0, [r2, r3]
120
121     // Turn on LED
122     ldr     r2, =GPIOC
123     ldr     r3, =ODR
124     ldr     r1, =PIN8
125     ldr     r0, [r2, r3]
126
127     ldr     r4, =PIN8
128     orrs    r0, r4
129     str     r0, [r2, r3]
130
131 loop_inf:
132     bl      delay
133     eors    r0, r4
134     str     r0, [r2, r3]
135     b       loop_inf
136
137 delay:
138     ldr     r1,=6000000
139 delay_loop:
140     subs    r1,r1,#1
141     bne     delay_loop
142     bx lr
143     /* End of student code */
```

Figure 10. Code to blink the LED

6.8 Demonstrate

Demonstrate each of the previous steps 6.3, 6.4, 6.5 6.6 and 6.7 to your lab TA. Your TA will register your work for this lab.