ECE 362 Lab Experiment 1: "First Blinks"

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 STM32F091 development board to blink. You will also write a few simple assembly language subroutines.

Now that you know that the hardware in your lab kit works, you are comfortable with creating projects, and you've had some experience typing in programs, you will be expected to complete this lab autonomously. If you need help, ask a teaching assistant. In lab, you will receive help for and be checked off for subroutines that you complete. You will also be given a module that will evaluate the subroutines. It will generate a completion code on the serial port as with Lab 0.

Instructional Objectives

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

Table of Contents

Step	Description	Points
0	Prelab Exercises	10
1	Background understanding for program generation	
2	Create a Project	
3	Arithmetic Exercises	40
4	Logical Operation Exercises	20
5	Shift Operation Exercises	20
6	Make the Lights Blink	10
7	Testing	
8	Submit your postlab results	*
	Total:	100

* All the points for this lab depend on proper completion of and submission of your post-lab results.

When you are ready for your lab evaluation, review this checklist.

Step 0: Prelab Exercises:

- Be familiar with lectures up to and including (02) Instruction Sets.
- Read this entire lab document. Complete the prelab exercises and submit your answers **before** attempting the lab experiment
- Read the user manual for the STM32F091 development board.
- Do the prelab exercises and submit them before attempting the lab experiment.

Step 1: Background understanding for program generation

1.1 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 like Figure 1:

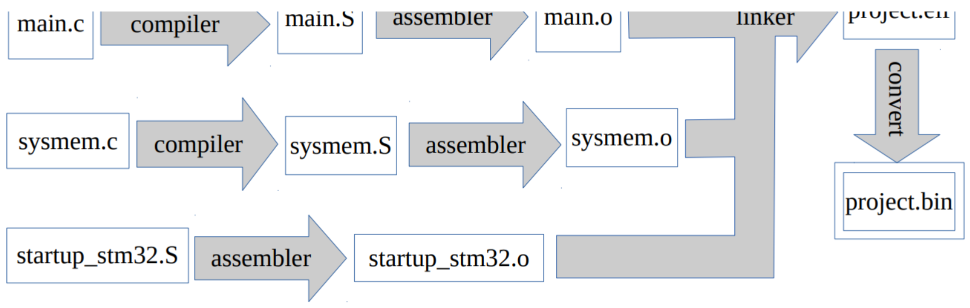


Figure 1: Flow of files during a project build

1.2 ARM Cortex-M0 specific assembler directives

Most of the code that is written in a .s file is meant to be executed on the microcontroller core. However some instructions on the .s file instruct the assembler to perform a specific task. Such instructions are called assembler directives. For now, we describe the four most important ones that are present at the beginning of the .s file.

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp
```

- **`.cpu cortex-m0`**: Lets the assembler know that the target CPU core for this code is a Cortex-M0 core (i.e., the core supports the ARM Cortex-M0 instruction set). Different types of ARM CPUs, and even different types of ARM Cortex CPUs support different groups of instructions. It is important for the assembler to know which CPU we want to target so that it does not accept an instruction that is not supported.
- **`.thumb`**: Specifically accept and generate only the Thumb 1 and Thumb 2 instructions supported by the target CPU.
- **`.syntax unified`**: Use the "unified" instruction names like `ADDS` rather than the ones listed in the ARM Architecture Reference Manual (e.g., `ADD`).
- **`.fpu softvfp`**: Select the *type* of floating-point unit to assemble for. The ARM Cortex-M0 lacks floating point hardware, and we won't be using floating point in assembly language. If you wanted to specify a global variable in assembly language and set it to a floating point value, you would need this directive for it to be understood by C-compiled module.

Step 2: Create a Project

This lab consists of multiple segments where you are expected to type in assembly code. Before you begin you must create a project, then you must use the "main.s" template file available on the website and add your code in the corresponding sections.

You created a project in Lab Experiment 0, and you will need to do so for each lab experiment in the course. Create a "lab1" project that uses the Standard Peripheral Library (StdPeriph). Although you won't be using any of the features of the Standard Peripheral Library in this lab, the `autotest.o` module provided for you relies on it.

A `main.c` is automatically created in the lab1/src directory. We won't be using it in this lab, so **delete it** by right-clicking (or option-clicking) on the `main.c` file and selecting "Delete". Then create a new file named "main.s" to hold your assembly language code. At the top of the file, add the four directives explained in section 1.2, followed by items that identify you and *invoke* the steps of the experiment you will build, below.

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.global login
login: .asciz "xyz"

.align 2
.global main
main:
    //bl   autotest // Uncomment this ONLY when you're done.
    movs r0, #1
    movs r1, #2
    movs r2, #4
    movs r3, #8
    bl   example // invoke the example subroutine
    nop

    movs r0, #35 // replace these values with examples from the prelab
    movs r1, #18
    movs r2, #23
    movs r3, #12
    bl   step31 // invoke Step 3.1
    nop

    movs r0, #10 // replace these values with examples from the prelab
    movs r1, #3
    movs r2, #18
    movs r3, #42
    bl   step32 // invoke Step 3.2
    nop

    movs r0, #24 // replace these values with examples from the prelab
    movs r1, #35
    movs r2, #52
    movs r3, #85
    bl   step33 // invoke Step 3.3
    nop

    movs r0, #29 // replace these values with examples from the prelab
    movs r1, #42
    movs r2, #93
    movs r3, #184
    bl   step41 // invoke Step 4.1
    nop

    movs r0, #0x11 // replace these values with examples from the prel
    movs r1, #0x22
    bl   step42 // invoke Step 4.2
    nop

    movs r0, #0 // unused as an input operand
    movs r1, #16
    movs r2, #2
    movs r3, #3
    bl   step51 // invoke Step 5.1
    nop
```

```
        movs r0, #5
        bl   step52 // invoke Step 5.2
        nop


        bl   setup_portc
    loop:
        bl   toggle_leds
        ldr  r0, =500000000
    wait:
        subs r0,#83
        bgt  wait
        b    loop

    // The main function never returns.
    // It ends with the endless loop, above.

    // Subroutine for example in Step 3.0
    .global example
    example:
        // Enter your code here
        bx   lr

    // Subroutine for Step 3.1
    .global step31
    step31:
        // Enter your code here
        bx   lr

    // Subroutine for Step 3.2
    .global step32
    step32:
        // Enter your code here
        bx   lr

    // Subroutine for Step 3.3
    .global step33
    step33:
        // Enter your code here
        bx   lr

    // Subroutine for Step 4.1
    .global step41
    step41:
        // Enter your code here
        bx   lr

    // Subroutine for Step 4.2
    .global step42
    step42:
        // Enter your code here
        bx   lr

    // Subroutine for Step 5.1
    .global step51
    step51:
        // Enter your code here
        bx   lr

    // Subroutine for Step 5.2
    .global step52
    step52:
        // Enter your code here
        bx   lr

    // Step 6: Type in the .equ constant initializations below

    .global setup_portc
    setup_portc:
        // Type in the code here.
        bx   lr

    .global toggle_leds
    toggle_leds:
        // Type in the code here.
        bx   lr
```

Each step in this lab experiment will involve the completion of a *subroutine*. Unless you have moved fast-forward through the lectures, you have not learned how subroutines work in assembly language, and that is not a problem. Here, we have set up the subroutines for you. You can think of each **`BL`** instruction as a "call" for a named subroutine, and the **`BX LR`** is the "return" from that subroutine. Your job is to fill in the code between the label of the subroutine (e.g., `step31`) and the `bx lr`.

The `main` label also defines a subroutine which is called by the code in `startup/startup_stm32.s`. We never return from the `main` subroutine for the code we write in this class, so you might not think of `main` as a subroutine.

Each step invocation in the `main` subroutine is preceded by one or more **`MOVS`** instructions that initialize a few registers. You may change any of these MOVS instructions to do your own testing of the assembly code that you write. In the debugger, use the "Step Into (F5)" button to step into each function. When the function returns, the code arrives at a **`NOP`**. This is a "no operation" instruction. That we will use as a visual indication that this is the point where you should examine the register values that were produced by the operations you wrote in the subroutine. At every point, you should be able to watch the register values in the "Registers" tab of the information window in the upper right corner of the debugger.

When debugging, you may place up to four breakpoints anywhere in your assembly code by double-clicking in the area just to the left of the line number of main.s. (If you attempt to place more than four breakpoints, the debugger will complain, and it will demand that you to remove the excess breakpoints before it runs the program. Once you set a breakpoint, you can click the "Resume" button on the debugger interface to quickly execute code without single-stepping through instructions. In this way, you can skip over exercises that you have already completed while you work on later exercises that you need to debug.

Step 3: Arithmetic Exercises

The subroutines for this step will exercise your knowledge of the arithmetic instructions. Each subroutine is preceded by MOVS instructions that initialize registers R0 through R3. The instructions you write should use only registers R0 through R3.

3.0: example

Fill in the subroutine body for `example` with instructions that will set the value of R0 to R0+R1+R2+R3. For instance, if you set the values R0 through R3 like this:

```
movs r0, #1
movs r1, #2
movs r2, #4
movs r3, #8
```

You should expect that the R0 register will contain the value 15 when execution reaches the `nop` following the subroutine invocation. It does not matter what values are left in R1, R2, and R3 after the return from `example`. Remember to use only the registers R0 through R3 when you write your instructions.

Solution for example

The operation cannot be implemented with a single instruction. You must compose multiple instructions to produce the result.

```
.global example
example:
    adds r1, r0, r1 // now, r1 = r0 + r1
    adds r1, r1, r2 // now, r1 = r0 + r1 + r2
    adds r1, r1, r3 // finally, r1 = r0 + r1 + r2 + r3
    movs r0, r1     // put the result into r0
    bx lr
```

You should copy this into the example subroutine in the file, and trace through the execution with the debugger to make sure you understand how it works.

Another solution for example

There are usually many ways to write the same high-level operation in assembly language. The fewer instructions you can use, the faster the code will run to completion. Here is another solution for the previous problem that has fewer instructions:

```
.global example
example:
    adds r0, r0, r1 // now, r0 = r0 + r1
    adds r2, r2, r3 // now, r2 = r2 + r3
    adds r0, r0, r2 // finally, r0 = (r0 + r1) + (r2 + r3)
    bx lr
```

It may be a little more difficult to understand. You should study it to discover how it works. For today's exercises, it does not matter how slowly your solution works (within reason). What does matter is that no registers other than r0, r1, r2, or r3 are modified by the code. The reasons for doing so may seem a little arbitrary, but you'll understand someday when we talk about the ARM Cortex Application Binary Interface (ABI) specifications.

3.1: step31

Fill in the subroutine body for `step31` with instructions that will set the value of the R0 register to the expression described in section 3.1 of the prelab.

Use the four **`movs`** statements before the subroutine is called to set up example values. You should expect that the R0 register will contain the expected result when execution reaches the `nop` following the subroutine invocation. It does not matter what values are left in R1, R2, and R3 after the return from `step31`. Remember to use only the registers R0 through R3 when you write your instructions.

When the subroutine is working as expected, show your TA.

3.2: step32

Do for **`step32`** as you did for **`step31`**.

When the subroutine is working as expected, show your TA.

3.3: step33

Do for **`step33`** as you did for **`step31`**.

When the subroutine is working as expected, show your TA.

Step 4: Logical Operation Exercises

4.1: An easy one

Fill in the subroutine body for `step41` with instructions that implement the logical expression specified in the prelab for step 4.1, where

- | represents the bitwise OR operator in C
- & represents the bitwise AND operator in C
- ^ represents the bitwise XOR operator in C
- ~ represents the bitwise inverse operator in C

You should expect that the R0 register will contain the expected result when execution reaches the `nop` following the subroutine invocation. It does not matter what values are left in R1, R2, and R3 after the return from `step41`. Remember to use only the registers R0 through R3 when you write your instructions.

When the subroutine is working as expected, show your TA.

A few notes:

The basic logical instructions are ANDS, ORRS, BICS, EORS, and MVNS. Read the Architecture Reference Manual to familiarize yourself with their operation.

4.2: Using temporary registers to hold values

Fill in the subroutine body for `step42` with instructions that will set the value of R0 to the expression found in the prelab. (Since the logical operations do not support immediate operands, you will need to load the constant integer values in the expression into temporary registers.) For instance, if you want to AND the value 0xff with r0, you might say something like:

```
movs r1, #0xff
orrs r0, r1
```

You might need to use different register names to implement the expression given.

You should expect that the R0 register will contain the expected value when execution reaches the nop following the subroutine invocation. It does not matter what values are left in R1, R2, and R3 after the return from `step41`. Remember to use only the registers R0 through R3 when you write your instructions. Note that you should implement an operation like (Rx & ~Ry) with the **`BICS Rx, Ry`** instruction.

When the subroutine is working as expected, show your TA.

Step 5: Shift Operation Exercises

5.1: An easy one first

Fill in the subroutine body for `step51` with instructions that will set the value of R0 to (R3 << R1) >> R2, where

- >> represents the right shift operator in C
- << represents the left shift operator in C

A left shift by n bits represents a multiplication by 2^n, and a right shift by n bits represents a division by 2^n. For instance, if you set the values R0 through R3 like this:

```
movs r0, #0 // unused as an input operand
movs r1, #16
movs r2, #2
movs r3, #3
```

You should expect that the R0 register will contain the value 49152 ($3*2^{16}/2^2 = 3*2^{16-2} = 3*16384$) when execution reaches the nop following the subroutine invocation. Of course, truncation of the result stored in R0 will occur if the the initial value in R3 is too large, or either of the shift values in R1 or R2 are too large. For instance, (3 << 48) >> 47 is zero because any left shift more than 32 will always shift all of the bits out of the register. It does not matter what values are left in R1, R2, and R3 after the return from `example`. Remember to use only the registers R0 through R3 when you write your instructions.

When the subroutine is working as expected, show your TA.

5.2 Combine With Logical Operations

Fill in the subroutine body for `step52` with instructions that will set the value of R0 to ((R0 & ~0x1) << 3) | 5, where '~' represents the bitwise inverse of the operation. Note that there is a special instruction that implements something like Rx & ~Ry. You might like to use that one. For instance, if you set the value R0 this:

```
        movs r0, #5
```

You should expect that the R0 register will contain the value 37 when execution reaches the `nop` following the subroutine invocation. It does not matter what values are left in R1, R2, and R3 after the return from `example`. Remember to use only the registers R0 through R3 when you write your instructions.

When the subroutine is working as expected, show your TA.

Step 6: Make the Lights Blink

You have not yet learned how to use general-purpose input/output (GPIO), but we can give you some code that will do the necessary steps to manipulate the pins of the microcontroller and let you fill in some additional assembly language to do interesting things. Four pins of Port C (PC6,PC7,PC8,PC9) are connected to four LEDs on the development board. Setting any one of these pins "high" will illuminate the appropriate LED. Soon, this will seem like a very easy thing for you to do, but we'll try to give you a gentle introduction and encouragement to learn more.

Our tradition is to give you code with the expectation that you will type it in. Do not ask someone else to type it in for you, and do not share your code with anyone else. This typing is not only practice for the assembly language constructs that you will be using, but also practice for debugging the mistakes that you will inevitably make when you do all that typing. It is natural to make these mistakes, and you will continue to do so through the semester (your instructor still makes mistakes typing things like this), so the ability to *recognize* errors and *recover* from them quickly is essential.

Initialize some constants

Type in the following .equ definitions just below the comment in `main.s` for Step 6.

```
// Step 6: Type in the .equ constant initializations below
.equ RCC,          0x40021000
.equ AHBENR,       0x14
.equ GPIOCEN,      0x00080000
.equ GPIOC,        0x48000800
.equ MODER,        0x00
.equ ODR,          0x14
.equ ENABLE6_TO_9, 0x55000
.equ PINS6_TO_9,   0x3c0
```

Figure 2: Some .equ initializations

Configure GPIO

Type in the following assembly language subroutine to initialize the RCC clock for Port C and set up the mode register for Port C to configure four pins for output.

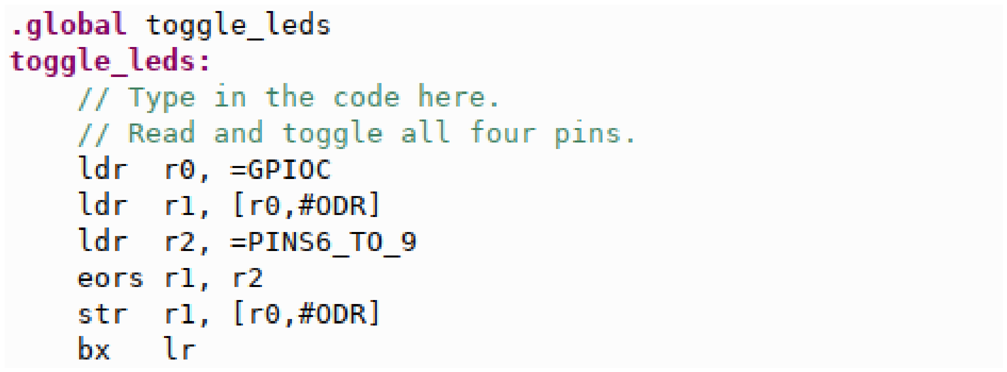
```
.global setup_portc
setup_portc:
    // Type in the code here.
    // Enable the RCC clock for the GPIOC peripheral.
    ldr  r0, =RCC
    ldr  r1, [r0,#AHBENR]
    ldr  r2, =GPIOCEN
    orrs r1, r2
    str  r1, [r0,#AHBENR]

    // Set pins 6 through 9 to be outputs.
    ldr  r0, =GPIOC
    ldr  r1, [r0,#MODER]
    ldr  r2, =ENABLE6_TO_9
    orrs r1, r2
    str  r1, [r0,#MODER]
    // Return
    bx   lr
```

Figure 3: Configure Port C

Toggle LEDs

Type in the following assembly language subroutine that will *toggle* all four of the pins connected to the LEDs.



```
.global toggle_leds
toggle_leds:
    // Type in the code here.
    // Read and toggle all four pins.
    ldr  r0, =GPIOC
    ldr  r1, [r0,#ODR]
    ldr  r2, =PINS6_TO_9
    eors r1, r2
    str  r1, [r0,#ODR]
    bx   lr
```

Figure 4: Toggle the LEDs

If you have typed everything in correctly, you should see all four LEDs on the development blink at a rate of 1 Hz. You can feel free to modify the `toggle_leds` subroutine to try different values. (For instance, consider what happens if you change the line that says

```
ldr r2,=PINS6_TO_9
```

to say, instead,

```
ldr r2,=0x40
```

Change `toggle_leds` all you want. It's for your own amusement. But the other subroutines must all work as described.

When the subroutine is working as expected, show your TA.

Step 7: Testing

Once you have tested each of the subroutines to implement for this lab, you can install the precompiled autotest.o module for the lab as you did for Lab 0 and uncomment the call to `autotest`. It will call each of the subroutines you implemented with many variations of register values to make sure they work exactly as they should. **You must be sure to set the `login` variable to the proper string in order to initialize the testing.** In past semesters, your instructor has been a little too helpful in supplying an `autotest` subroutine. Students said, "I'll just let autotest check everything for me, so I don't have to," and this *deprived* them of an opportunity to develop their debugging skills. We wouldn't want to *deprive* you of any excellent learning experience. Therefore, the autotest module will check each one of the subroutines you wrote and tell you how many subroutines are correct. (Wire the serial port just like you did for lab 0.) The subroutines tested are `example`, `step31`, `step32`, `step33`, `step41`, `step42`, `step51`, `step52`, and `setup_portc`. If you cannot get the `example` subroutine (which we provided for you) to work, you won't get any points.

If `autotest` tells you that only 7/9 of the subroutines work properly, you can intentionally make a subroutine do the wrong thing by adding a **`movs r0,#0`** just before the `bx lr` instruction. This will force the result to be always be zero. If you insert it into the `step41` subroutine, and, if the score remains 7/9, you know that `step41` is one of the two subroutines that does not work correctly. Using this process of elimination, you can find all of the subroutines that do not work and repair them. **However, it will be easier, in the long run, if you carefully test each one as you write it.**

8 Submit your postlab results

Once your lab experiment works to your satisfaction, and you see how many parts work correctly on the serial terminal, get the completion code for your work. This code is a cryptographic confirmation of your completion of the lab experiment as well as your identity and your score. (Remember to change the "login" variable from "xyz" to your login.) It is normal for the code to change every time you run it. Choose one of the codes and carefully enter all 32 characters into the postlab submission for this lab experiment.

For this lab, and most to follow, you must also submit the program that you wrote so that it can be checked by the course staff. Either upload the file or copy it from SystemWorkbench and paste it into the text box. Make sure that your entire program is shown there.

Lab Evaluation Checklist

Normally, we'll have a checklist of things that you should review before going into your evaluation. There will be no evaluation for this lab experiment.

- [ ] You did not share your microcontroller with anyone else, right? We will know if you did.
- [ ] Implement step31.
- [ ] Implement step32.
- [ ] Implement step33.
- [ ] Implement step41.
- [ ] Implement step42.
- [ ] Implement step51.
- [ ] Implement step52.