CDA 3631C Embedded Operating Systems

LAB ASSIGNMENT 2: Keil uVision Debugging and Assembly Programming

Date Posted: 21 September 2019 Date Due: 1 October 2019 (11:59 PM)

Objectives:

- Learning to use the debugging feature of Keil uVision Design Kit.
- Perform basic ARM assembly programming

Files Needed:

- Lab2.s
- Lab2_forms.docx

Assignment:

Create a new project using the following steps.

1. Open the Keil uVision5 development environment.
2. Click "Project", then "New μVision Project…"
3. In the "Create New Project" dialog window, change to an appropriate directory and enter a name for the project file.
4. In the ARM hierarchy, select "STMicroelectronics", "STM32F2 Series", "STM32F207IG", "STM32F207IGHx".
5. In the Manage **Run-Time Environment** dialog window, there are two components that (for now) always need to be added. Click on **CMSIS->CORE**, and click on **Device-> Startup**, then OK. Note that you can open this dialog again at any time to add and remove components from the project.

You should now have a project with several files in it. It is common to have a startup template that can be used to jumpstart a project, such as the **startup_stm32f207xx.s** file. Sometimes the template works as is, but often it needs to be tweaked. Let's skim over it and look at the kind of things it does; we also went over this in class lectures.

- Line 48 – sets up 0x400 bytes for the stack.
- Line 71 – sets up vector addresses for all exceptions.
- Line 183 – sets up the reset handler code. This is where the processor first starts executing when it resets. The name of this code is not important, just that this is the address stored in the reset interrupt vector on Line 77.
- Line 197 – All vector addresses must point somewhere. For lack of something more intelligent to do if an exception occurs, the startup file creates a separate infinite loop for each main system exception, and one **Default_handler** for all I/O device exceptions starting at line 239.

It is often helpful to have separate handlers for debugging. If the processor does encounter an unexpected/unimplemented exception, the processor may be halted. The value of the program counter will indicate which loop it is stuck in, which can show which exception's infinite loop was running. Also, in a well-crafted system, many of the default handlers would be replaced by user code.

Look back at the reset handler on line 184. It does two things. First, it calls the function SystemInit (automatically provided in the system_stm32f2xx.c file), then it jumps to the label __main (and does not expect to return). You will always need to write a **__main** function, and for this lab we will use our own initialization code.

6. Copy the file **lab2.s** from Canvas to your project directory. This may be the same directory as the project file (*.uvprojx), or any subdirectory. The code is shown on the next page and contains the **__main** function. It also has a simple (basically empty) initialization function called **mySystemInit**.

7. Add **lab2.s** to the project by right-clicking on "**Source Group 1**" and selecting "**Add Existing Files to Group** "**Source Group 1**…". Note that you may need to change the extension filter to see the assembly file. The dialog box also allows you to keep adding files, so you will need to click "**Add**", then "**Close**".

8. To make the project use the custom initialization function instead of the default, change the labels in lines 186 and 189 of **startup_stm32f207xx.s** to **mySystemInit**.

```
AREA MyData, CODE
array1
DCD 0x87654321, 0x23456789, 0x3456789A
array2
DCD 0xAABBCCDD, 0x7FCCDDEE, 0xCCDDEEFF
AREA MyAnswers, DATA
array3
SPACE 12
AREA MyCode, CODE, READONLY
EXPORT __main
EXPORT mySystemInit
mySystemInit
BX LR
__main
ENTRY
LDR R0, =array1
LDR R1, =array2
LDR R2, =array3
LDR R3, [R0], #4
LDR R4, [R1], #4
ADDS R4, R3, R4
STR R4, [R2], #4
LDR R3, [R0], #4
LDR R4, [R1], #4
ADDS R4, R3, R4
STR R4, [R2], #4
LDR R3, [R0]
LDR R4, [R1]
ADDS R4, R3, R4
STR R4, [R2]
; LDR R0, =0x22000000
; LDR R1, =32
;loop
; LDR R2, [R0], #4
; SUBS R1, #1
; BNE loop
B .
```

9. Right click on "**Target1**" in the Project tab, click "Options for Target '**Target1'**…", and click the "**Use MicroLIB**" checkbox in the "Code Generation" box. If the board is being used, make sure that the "**ULINK2/ME Cortex Debugger"** is selected in the Debug tab. This lab will demonstrate some concepts that are difficult to accomplish with the physical board. For this lab, select the "**Use Simulator**" option (this allows you to use the debug feature without actually connecting the debugger)

10. Build the project by clicking on the "**Build**" icon in the upper left of the IDE window.

11. Enter the debug mode by clicking the "**Start/Stop Debug Session**" icon. It is a red "d" inside a small magnifying glass. The IDE will warn you that only 32k of code is supported, which is fine.

The Debug IDE has several important windows:

- Registers – These show the main 16 registers along with the program status register which can be expanded to show individual fields.
- Disassembly – This shows the address, machine code, and assembly code being executed, regardless of what format the source code was written in.
- Source code – This window (which has only tabs and no window name) shows the source files being executed. You can make changes to the files here without having to exit debug, but the changes won't take effect until the project has been rebuilt.
- Call Stack + Locals / Memory – monitor variables (once they are created in C) or monitor memory locations. To look at memory contents, you need to click on the Memory 1 tab and enter the starting location, such as 0x20000000.

The windows can be undocked and rearranged if desired. Feel free to explore the debug interface. Most buttons should be fairly straightforward. One handy note is that a breakpoint may be set by clicking in the grey left-hand margin in any code window, i.e. the disassembly, the source window, or the code window outside of the debugger (as demonstrated in class lectures).

Complete the instruction trace in **Table 1** (lab2_ form.docx) by single stepping through the program. The first row of the table should have the register values immediately after the processor is reset (it is reset when the debug session starts). For the following rows, do a single-step and only fill out the box if the value of that register has been affected (it may be affected even if it doesn't change). Note: do not uncomment the section at the end. You will run this code in a later step. The Bytes column refers to the number of bytes of that row's instruction, i.e. the instruction that generated the values for that row.

Question 1
What values are in memory locations 0x20000000 through 0x2000000B when the program finishes?
Question 2
Compare the branch destination in R0 before the "BX R0" is executed in the **Reset_Handler** (after pressing RST in debug window) to the value in the PC immediately after the branch is executed. Explain the discrepancy between the values.

12. Select "**Debug**", "**Execution Profiling**", and check "**Show Time**". Rerun the program from the beginning and notice that the time spent execution each instruction is listed on the left.

Question 3
How long does the program take to execute? Count only the time in "__main".

The sums should show up in the bit-band region of memory. However, uVision doesn't like to show the bit-band alias region in debugging. Uncomment the following code near the end of the file. LDR R0, =0x22000000
LDR R1, =32
loop
LDR R2, [R0], #4
SUBS R1, #1
BNE loop

This code will loop through the first 32 aliased locations. Complete **Table 2** by placing a breakpoint at the SUBS instruction on line 45, rebuild the program, and repeatedly run the program.

Question 4
How do the values captured in **Table 2** relate to the answer array (in term of Bit banding)?

Question 5
How long does the program take to execute now? Count only the time in "__main". How is the time reported for the lines 44-46?

Comment/delete these lines out again for the remainder of the lab.

13. The first AREA directive in **lab2.s** is actually misleading. The arrays of data should be in data memory, not code memory. Modify line 1 in lab2.s from "AREA MyData, **CODE**" to "AREA MyData, **DATA**".
14. Rebuild the program. Place a breakpoint at line 41 in lab2.s, the infinite loop, and rerun the program.

Question 6
What addresses did uVision use for **array1** and **array2**?
Question 7
What data is in these arrays when the processor is reset (i.e. the debugger is first started)?

The issue is that variables in SRAM have random values (or often 0's) when the processor comes out of reset, and the code has to manually initialize them all. According to standard programming practices, initializing the variables is one of the operations that should be performed in the "__ main" function. The "__main" function is also supposed to do some other setup operations, and then branch to a "main" function where the application code starts. Let's make the project a little more standard.

15. Create another assembly file. Move the entire **"__main"** function from **lab2.s** to the new file and rename it "**main**". Create a new **"__main**" function in **lab2.s** that initializes the two arrays with the required data (look at MOVW.W and MOVT.W) and then branches to the "**main**" function in the other file. You will need to add appropriate directives to import and export labels. Test the new setup to verify that your **"__main"** function correctly initializes data and the "**main**" program correctly calculates the answer. Finally, rewrite the "**main**" function so that it executes a loop that iterates **three** times to add the arrays instead of performing the addition with serial code.

Question 8
How much time is required by the new "**main**" function to add the values using a loop?

Deliverables:
Upload an electronic copy of the deliverables to Canvas per the lab submission guidelines. Please note: **ALL** deliverables must be present in the single PDF report. Program-specific files, like the assembly source files, must **also** be submitted in case they need to be verified using their respective programs.

- Typed answers to questions
- Completed Table 1
- Completed Table 2
- Lab2.s file with variable initialization code
- Assembly file with addition code that loops