# Laboratory 1

## Intro to CodeWarrior

**Concepts:**

- Debugging and simulating with CodeWarrior
- Basic assembly program structure

**Objectives:**

- Simulate the S12 microcontroller using CodeWarrior
- Use trace and breakpoints to monitor a program's execution.
- Read and modify memory and registers during debugging.

**Files Needed:**

- lab01.zip from Blackboard

**Introduction:**

This week's lab is intended to familiarize you with the CodeWarrior IDE. The assignment in the following section uses CodeWarrior to examine some existing programs. It will demonstrate techniques for debugging programs and help you gain some familiarity with the processor.

**Assignment:**

1. Download lab01.zip from Blackboard and unzip the project into a folder. Note the path to the folder.

2. Open CodeWarrior, open the lab01.mcp file in the folder you just unzipped (*mcp* is the extension for a CodeWarrior project).

You should now see a file hierarchy under a tab labeled *lab01.mcp*.

The file *main.asm* contains the assembly code. In today's lab, the program has already been written. In future labs, this is where your code will go.

The file mc9s12g128.inc contains declarations for all the named bits and control registers in the processor. This file won't be used in Lab 1, but it will be referenced quite a bit when working with physical I/O and peripheral devices later in the semester.

The *project.prm* file tells the CodeWarrior assembler where different memory types are located for this particular chip's memory map. Open this file. Notice that the RAM segment is listed as READ_WRITE memory from 0x2000 to 0x3FFF. The assembler will automatically put internal variables into this RAM space.

Also note that ROM_C000 is specified as READ_ONLY. The memory at this range is Flash. As far as an executing microprocessor program is concerned, this space contains only program code and constants and cannot be written to. Special steps are required to write to Flash, so it cannot be used for variables using *store* instructions like RAM.

Near the bottom of the file, notice the "VECTOR 0 Entry" line. This configures the microprocessor to start executing at the address corresponding to *Entry* when the chip is powered up or reset.

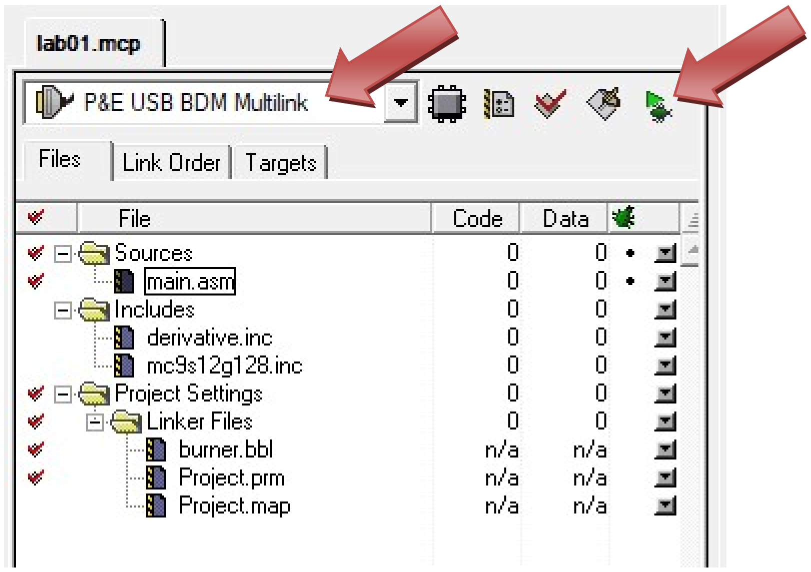
3. Open *main.asm* and read through it.

You should see a brief header comment section, followed by an ";export symbols" section needed by the assembler. Next, there is an "ORG $3000" section. ORG tells the assembler to start placing items in memory starting at that address. Later, we will use this section to specify program inputs and outputs, since we have control over which addresses are used.

The "MY_EXTENDED_RAM: SECTION" will be used for internal variables (those that the program uses for storage, but the user never needs to look at). The assembler will find RAM for items in this section, but we are not concerned about which specific addresses are used. Do not put internal variables in this section.

The "MyCode: SECTION" is where the instructions are. There is a small header that you should not modify. As part of this header, the stack pointer(SP) register is initialized. Although we won't use the SP in this program, it appears in the header so this file can be used as a template, but it would need to be uncommented.

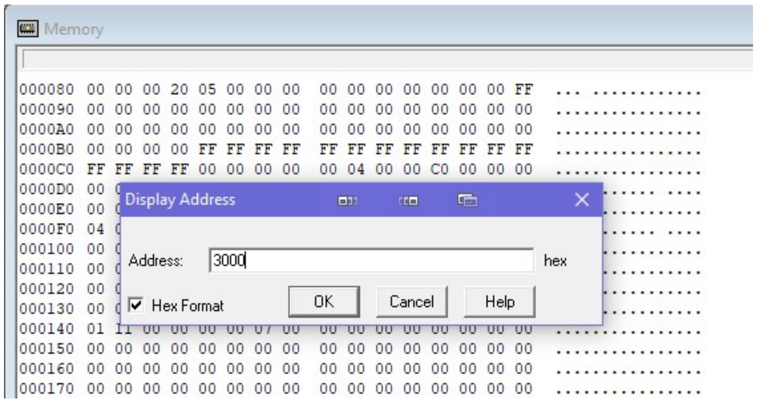
4. In the lab01.mcp tab, make sure "P&E USB BDM Multilink" is selected from the drop-down list, shown in Figure 1. The development board must be plugged into your computer's USB port. Then, click on the "debug" button, depicted as a green arrow and a tiny bug. This will assemble the program and open a second window. You may get a "Loader warning" pop-up window. Click OK to clear the window.



| File | Code | Data |
|---|---|---|
| Sources | 0 | 0 |
| main.asm | 0 | 0 |
| Includes | 0 | 0 |
| derivative.inc | 0 | 0 |
| mc9s12g128.inc | 0 | 0 |
| Project Settings | 0 | 0 |
| Linker Files | 0 | 0 |
| burner.bbl | n/a | n/a |
| Project.prm | n/a | n/a |
| Project.map | n/a | n/a |

Figure 1

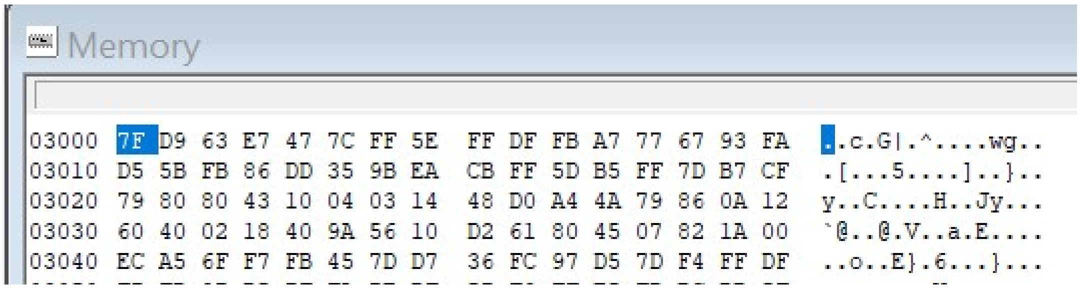
5. One of the windows in the “True-Time Simulator & Real Time Debugger”, which we’ll call just the “debugger”, is the Memory window. Go to memory address 3000h by right-clicking in the window, and selecting “Address...”. As shown in Figure 2, you will get a pop-up box that allows you to type in an address, shown below.



```
000080  00 00 00 20 05 00 00 00  00 00 00 00 00 00 00 FF  ... .............
000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000B0  00 00 00 00 FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0000C0  FF FF FF FF 00 00 00 00  00 04 00 00 C0 00 00 00  ................
0000D0  00
0000E0  00
0000F0  04
000100  00
000110  00
000120  00
000130  00
000140  01 11 00 00 00 00 07 00  00 00 00 00 00 00 00 00  ................
000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Figure 2

Once you click on OK, you’ll see a section of memory with random, shown in Figure 3. Your window will probably have different numbers, and the numbers may be different every time.



```
03000 7F D9 63 E7 47 7C FF 5E  FF DF FB A7 77 67 93 FA  ..c.G|.^....wg..
03010 D5 5B FB 86 DD 35 9B EA  CB FF 5D B5 FF 7D B7 CF  .[...5....]..}..
03020 79 80 80 43 10 04 03 14  48 D0 A4 4A 79 86 0A 12  y..C....H..Jy...
03030 60 40 02 18 40 9A 56 10  D2 61 80 45 07 82 1A 00  `@..@.V..a.E....
03040 EC A5 6F F7 FB 45 7D D7  36 FC 97 D5 7D F4 FF DF  ..o..E}.6...}...
```

Figure 3

Note that the leftmost location is 3000h. The second location from the left (with D9h) is 3001h, and so on. There is a larger space between the xx07 and xx08 columns. For example, 3018h is CBh.

6. This lab will add up 5 one-byte numbers. Since the microprocessor powers up with random values in RAM, we need to enter specific values so that we can predict the results.

   Enter the six values from Table 1 into memory at the addresses shown. Double-click on the memory location, and you will see "Edit Mode" in the Memory window. Type in the value that you want. Press "enter" to exit the edit more, changing that one location. If you press "space" instead, the debugger will conveniently move to the next memory byte.

| **Memory Address** | **Contents** |
|---|---|
| 3000h | 6Bh |
| 3001h | 47h |
| 3002h | EEh |
| 3003h | 85h |
| 3004h | 9Ch |
| … | |
| 3008h | 00h |

Table 1. Initial Memory Values

Figure 4 shows the assembly language program from *main.asm*. It adds a list of five one-byte numbers beginning at address 3000h, and it stores the one-byte result in address 3008h.

7. ***Before*** running the program, add a breakpoint at the "endmain BRA endmain" instruction by right-clicking on the instruction and selecting "Set Breakpoint". Notice that the breakpoint appears in both the Source and Assembly panes.

| Code Line | Memory Address | Contents | Assembly Code |
|---|---|---|---|
| 1: | C000h | B6 | LDAA 0x3000 |
| | C001h | 30 | |
| | C002h | 00 | |
| 2: | C003h | BB | ADDA 0x3001 |
| | C004h | 30 | |
| | C005h | 01 | |
| 3: | C006h | BB | ADDA 0x3002 |
| | C007h | 30 | |
| | C008h | 02 | |
| 4: | C009h | BB | ADDA 0x3003 |
| | C00Ah | 30 | |
| | C00Bh | 03 | |
| 5: | C00Ch | BB | ADDA 0x3004 |
| | C00Dh | 30 | |
| | C00Eh | 04 | |
| 6: | C00Fh | 7A | STAA 0x3008 |
| | C010h | 30 | |
| | C011h | 08 | |
| 7: | C012h | 20 | BRA endmain |
| | C013h | FE | |

Figure 4 – Sequential Program to Add Five Numbers

The Assembly pane may hide the machine code by default. Right-click in the pane and select "Display", then "Code", and you should see the contents above. Note that the figure shows the byte-by-byte storage of the instruction vs the instruction-by-instruction format in the Assembly window. Finally, just to show that consistency is not important to some people, the Assembly window displays hexadecimal using a *0x* prefix instead of a *$* prefix or *h* suffix. You CANNOT use the *0x* notation when writing assembly.

8. Click on the green arrow button to run the program. The debugger should halt with the "endmain" line highlighted.

**Question 1**: What answer does the program calculate (i.e. what is the one-byte value in 3008h?)

**Question 2**: If the numbers are added as unsigned bytes and the number of digits in the answer was not limited to one byte, what is the correct sum of the five hexadecimal numbers?

9. Reset the S12 to the beginning of the program by clicking the "Reset Target" button. Note that this DOES NOT reset memory addresses, so you may need to restore some of them according to the table above.

10. Fill in the values for Figure 7 by tracing through the program instruction by instruction. The command for this is the "Single Step" button (or press F11) as shown in Figure 5.

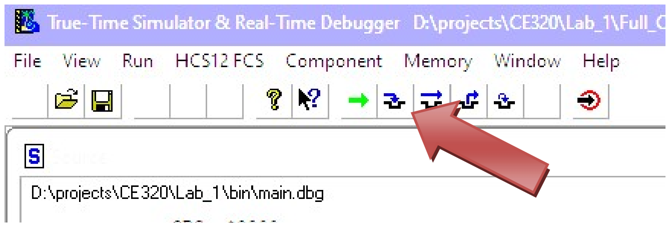
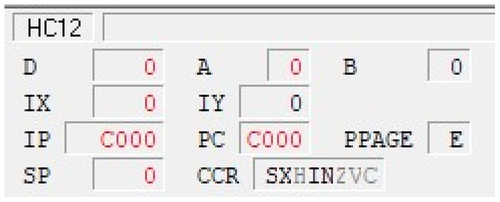

True-Time Simulator & Real-Time Debugger D:\projects\CE320\Lab_1\Full_C
File View Run HCS12 FCS Component Memory Window Help
D:\projects\CE320\Lab_1\bin\main.dbg

Figure 5 - Single Step button

Each line in the figure must be completed with the values ***after*** the line has been executed. The first row shows the initial value of the PC (set to the first instruction's address), with all other registers unknown since the program has not affected them yet.

*Note that after an instruction executes the CCR bits are black if they are 1, and they are light gray if they are 0. In* Figure 6, *the S, X, I, and N bits are 1, the H, Z, V, and C bits are 0.*



```
HC12
D        0   A       0   B        0
IX       0   IY      0
IP    C000   PC   C000   PPAGE    E
SP       0   CCR  SXHINZVC
```

Figure 6 - CCR bits

**Question 3**: The running sum in accumulator A may experience unsigned overflow (i.e. the correct sum requires 3 digits). Which trace lines show unsigned overflow after adding a value to the running sum in A?

**Question 4**: Which trace lines contain signed overflow after adding a number to the running sum in A? I.e. which additions cause an incorrect change in sign?

**Question 5**: Which condition code register bits correspond to the unsigned overflows and signed overflows in the lines from Questions 3 and 4?

| Trace Line | Code Line | PC | A | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| 0 | - | C000 | - | - | - | - | - |
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |

Figure 7 - Program Trace

Figure 8 shows an assembly language program. This program adds five numbers like the first program, but it uses a loop instead. Close the debugger. Comment out or delete the six lines of code from the first program. Then, enter the assembly code for the second program from Figure 8. You are not expected to know what all the instructions do yet. Run the debugger, and then reenter the values from Table 1 into memory.

**Question 6:** Using the Assembly window, complete the Machine Code column from Figure 8. Recall that the Assembly pane shows all bytes for an instruction in one line. You should enter one byte per row in the figure.

11. Run the new program from the beginning, remembering to set a breakpoint at the “endmain BRA endmain” line.

| Code Line | Memory Address | Machine Code | Assembly |
|---|---|---|---|
| 1: | C000h | | CLRA |
| 2: | C001h | | LDX #3000h |
| | C002h | | |
| | C003h | | |
| 3: | C004h | | LDAB #5 |
| | C005h | | |
| 4: | C006h | | BEQ *+8 |
| | C007h | | |
| 5: | C008h | | ADDA 0,X |
| | C009h | | |
| 6: | C00Ah | | INX |
| 7: | C00Bh | | DECB |
| 8: | C00Ch | | BRA *-6 |
| | C00Dh | | |
| 9: | C00Eh | | STAA 3,X |
| | C00Fh | | |
| 10: | C010h | | BRA * |
| | C011h | | |

Figure 8 – Looping Program to Add Five Numbers

12. Verify that the second program generates the same answer in 3008h as the first program. If it doesn't, fix it before you move on.

13. Reset the simulator to the beginning of the program and complete the program trace in Figure 9 for the first 15 executed instructions. (The program will not reach the end.)

14. Reset the processor to the beginning of the second program again and add a breakpoint at line C006h. Then, run the program from the beginning. The address displayed in the PC should be C006h when the processor halts. Note that when the processor stops at a breakpoint, the instruction that the PC points to has not yet been executed.

15. A breakpoint trace is like a program trace, except that instead of recording the register values after each line of code is executed, we will record the register values when the processor stops at the breakpoint. Complete the breakpoint trace in Figure 10, given that you are currently at iteration 0, and the first time that the program returns to line C006h is considered iteration 1. Clicking on the "Start/Continue" button (i.e. the green arrow button) will resume program execution and halt at the next breakpoint.

**Question 7**: Which line(s) of code in Figure 8 must be changed to add $100_{10}$ numbers instead of 5 numbers?

**Question 8**: Which line(s) of code in Figure 8 must be changed to add numbers beginning at memory location 3200h instead of 3000h?

**Question 9**: Which line(s) of code in Figure 8 store the sum to memory?

| **Trace Line** | **Code Line** | **PC** | **IX** | **A** | **B** | **N** | **Z** | **V** | **C** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | |
| 2 | | | | | | | | | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |
| 7 | | | | | | | | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | | | | | | |
| 13 | | | | | | | | | |
| 14 | | | | | | | | | |
| 15 | | | | | | | | | |

Figure 9 – Looping Program Trace

| Iteration | PC | IX | A | B | N | Z | V | C |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |

Figure 10 - Breakpoint Trace

**Deliverables/Scoring:**

- 20 points - Compliance with posted lab report guidelines.
- 20 points - Answers to questions.
- 20 points - Completed Figure 7 – Program trace
- 20 points - Completed Figure 9 – Program trace
- 20 points - Completed Figure 10– Breakpoint trace

Submit the deliverables according to the lab report guidelines posted on Blackboard. There is no code submission or demonstration for this lab.