USC Viterbi
School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 459Lx - Embedded Systems Design Laboratory

Programming the Freescale MC908JL16 in Assembly Language

(for CodeWarrior V6.x)

by

by Allan G. Weber

1 Introduction

This document discusses the use of the CodeWarrior for Microcontrollers, Special Edition software for programming the Freescale HC08 microcontrollers in assembly language. Much if this document is generic and applies to any processor in the HC08 family. However parts that discuss things like memory layout are specific to the processor currently being used in the class. The class is currently using the MC908JL16 (sometimes listed as MC68HC908JL16.) If the CodeWarrior compiler is used for other HC08 processors the information in these areas will have to be adjusted to match the other processor.

2 Installation Instructions

The CodeWarrior installation software can be obtained from the instructor if you wish to install the software on your own system. It should run on any Windows XP or Windows 7 system. To install the software:

- Copy the installation file to your hard drive. Depending on the version it will have a name like "`CW_MCUs_V6_3.exe`".
- Double-click on the file you just copied. There may also be some patch or updates file with it.
- Proceed through the series of questions and agreements posed to you. The program will try to install itself under the Program Files folder of your C drive. Feel free to change that if you desire. When you are asked what type of installation you desire, choose "Complete".
- If there are any update files these can also be installed.
- After the installation is complete, reboot your system.

The CodeWarrior application can be found in the Start menu under "All Programs", "Freescale CodeWarrior", "CW for Microcontrollers V6.x", "CodeWarrior IDE". You should now be ready to run the CodeWarrior development system to create software for your microcontroller.

3 Starting a New Project

In order to create a program for your microcontroller, you must first have CodeWarrior create a project. A CodeWarrior project is a folder that contains all the necessary files and configuration information to build the project. Start CodeWarrior from the desktop icon or the Start menu and follow these steps.

- File…New Project…. This will bring up the new project wizard to gather information about your project.

- On the Device and Connection screen select the microcontroller you will be using. For most of the class projects it should be the MC68HC908JL16. Open up the "HC08" list and then open the "JK/JL Family" list. Click on the microcontroller being used.

  In the "Connections" box, select "P&E Multilink/Cyclone Pro" if you will be using the USB programming hardware, or "Mon08 Interface" to use the Freescale development boards. See Sec. 9 for more information on the programming hardware. Click "Next".

- On the Project Parameters screen, select the programming language. Select "Relocatable assemby" and deselect the other languages. Type a project a name in the text box. CodeWarrior will create a folder of this name under your "My Documents" folder or you can change the location. Click "Next".

- On the Add Additional Files screen you can add source files to the project (this can also be done later.) Select the files to add and then click on "Add". If the "Copy files to project" button is selected the files will be copied into the "Sources" folder in the project folder the new project wizard is creating. Use this option if you are going to create your project by modifying one of the demo or template files. If this option is not selected, the source files are left in their original location.

  One file in the project must contain the starting point of the program. If the "Create main.c/main.asm file" box is checked, CodeWarrior will add a main.asm file to your project containing this entry point. You can later edit this file to add your own code. Alternatively you can uncheck the main.asm box and add your own file that contains a entry point. This method is easier if you are copying a main program from some other project. However it is done there must be one, and only one, file that contains the starting point. Click "Next".

- On the Processor Expert screen, select "None" when it asks about Rapid Application Development Options. Click "Finish".

At this point CodeWarrior will create the project folder and bring up the project window.

4 Navigating Through Your Project

After CodeWarrior creates your project you will see the project window as shown in Figure 1. In the left-hand window pane you will see a folder structure for the files that are part of your project.

- The "Sources" folder contains your source files. In that folder you should see the files you asked to be added to your project. If you didn't add any and selected the option to have CodeWarrior create a main.asm it will show up here and you can either edit main.asm or replace it with another file. If you choose to replace it, use the "Remove" command under the Edit menu to delete main.asm from the project, then use the "Add Files" command under the Project menu to add a different main program file. Other ASM files can also be added to the project. Double-clicking the file will open it in the CodeWarrior editor.

- The "Includes" folder contains files specific to model of microcontroller you are using. Your program should reference the "derivative.inc" file which then references the correct include file for your microcontroller. These files declare variable names you can use to read and write the internal registers of your microcontroller. It also defines bit names that allow you to read, set, or clear any bit in one of the internal registers. You should use these variables rather than trying to create your own system of addressing the internal registers.

- The "Linker Files" folder, contains files used the CodeWarrior program to compile and link your program. The only file you should be interested in is the ".prm" file that specifies the overall memory map of your program and also provides the ability to register your interrupt service routines (ISR's). Initially a new project has a file named "Project.prm" in this folder. As with the main ASM program file, you can either edit this file or remove it and replace it with another. See Sec. 6 for more information.

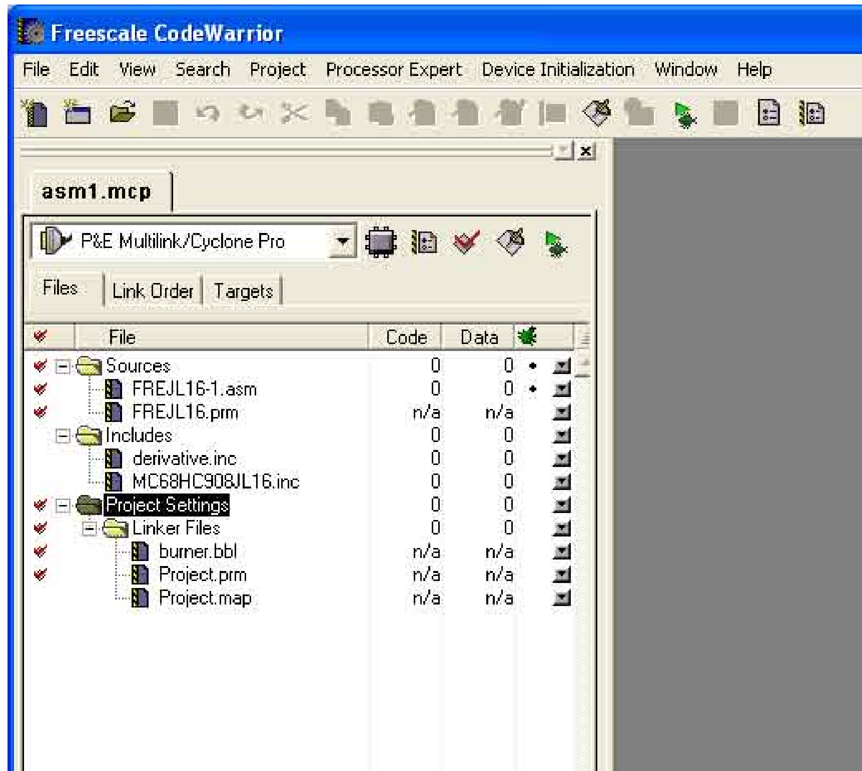

Figure 1: New Assembly Language Project Structure

5 Writing Code

Writing assemby code to be run on a microcontroller requires you to manage some of the low-level details of the system because a microcontroller has no operating system. Not only are you writing the application to be run on the system but also the essential portions of OS code. The low-level details you must manage include reading and writing internal registers, registering interrupt service routines, dealing with memory addressing issues, and some basic optimization issues.

More information can be found in the HC08 Manuals included with CodeWarrior, namely the Compiler reference and Linker reference. Copies of these manuals in PDF format can be found on the EE 459Lx web site.

5.1 Program Initialization

In order to get an ASM program running properly a few things are needed at the start of the program. The default main.asm file contains the code shown below.

```
MyCode:     SECTION
main:
_Startup:
        LDHX    #__SEG_END_SSTACK ; initialize the stack pointer
        TXS
        ;CLI                      ; enable interrupts
        ; Insert your code here
```

Program execution starts at the address of the label "`_Startup`". The linker stores the address of this location in the reset vector ($FFFE:FFFF). When the processor starts up it reads the contents of this 16-bit vector and uses the address stored there as the address of the first instruction to execute.

The first instructions executed load the stack pointer with the address of the top of the processor stack. Once the stack has been initialized the program can use interrupts and subroutines. On the JL16 the default location for the top of the stack is at location $150 and the stack grows downwards from there. Since the RAM memory on the JL16 occupies locations $060 to $25F, the default location for the stack has the effect of dividing the RAM into two regions. If you need to have a larger contiguous block of RAM space, the stack can be moved to the very top of RAM with the instructions below.

```
        ldhx    $25F
        txs
```

The HC08 processors have a "Computer Operating Properly" (COP) timer that will cause the processor to execute a reset operation if the COP timer isn't cleared before it overflows. This is to prevent the chip from becoming essentially dead due to a runaway program.

For the programs in EE 459 the COP function isn't normally needed so the COP timer should be disabled. Put another way, the COP timer **MUST** be disabled unless your program includes the instructions to reset the timer periodically. Disabling the COP timer can be done with the instruction shown below that stores a 1 in the `CONFIG1` register.

```
        mov     #1,CONFIG1      ; Disable COP reset
```

This should be done at the very beginning of the program, preferably just after the code to load the stack pointer.

5.2 Program Termination

Unlike writing a program on a larger computer, the microcontroller does not have an operating system that can take over control once the program is finished and exits. This means the program should never exit. It should always be doing something, such as running in an endless loop.

```
done:   bra     done            ; loop forever
```

If you are not using interrupts, you can also include the assembly language instruction "`WAIT`" at the end of the program which essentially makes the processor stop and wait for the next interrupt to occur.

5.3 Accessing RAM and ROM Memory

The RAM memory of the microcontroller is laid out in a way that can complicate things. The JL16 microcontroller has 512 bytes of RAM ($060 - $25F). The 160 bytes from $060 to $0FF is in the "direct page" or "zero page" since the upper byte of the address is zero and can be accessed using 8-bit addresses, known as "direct addressing". The rest of the RAM (from $100 to $25F) needs to be accessed with 16-bit addresses, known as "extended addressing". Memory accesses that use the direct addressing mode are faster than those using extended addressing and the instructions use fewer bytes of program space.

When allocating variables in RAM, they should usually be put in the direct page in order to take advantage of the more efficient direct addressing mode. To declare variables in the direct page, the following directive is used.

```
MY_ZEROPAGE: SECTION  SHORT
```

Any variables declared after this directive are allocated into the available zero page RAM. The first byte would be allocated to location $060, the second to $061, etc.

To switch RAM allocation back to the default (extended addressing) part of RAM, use the following directive.

```
DEFAULT_RAM : SECTION  SHORT
```

These directives can be use as many times as needed to determine where variables are allocated.

Some instructions only implement direct addressing mode and not extended addressing. These instructions can not be used to access RAM locations outside of the zero page. The following instructions can only access memory with direct addressing: ASL, ASR, BCLR, BRCLR, BRSET, BSET, CBEQ, CLR, COM, DBNZ, DEC, INC, LSL, LSR, MOV, NEG, ROL, ROR, TST. For example, the following instruction is illegal.

```
        mov     #1,$0140
```

The MOV instruction can only address memory locations in the zero page. To move a value into a RAM location outside the zero page, use a pair of load and store instructions.

```
        lda     #1
        sta     $0140
```

In general, if the instruction does not implement extended addressing the way to operate on data outside the zero page is to first move the data to the accumulator (`LDA`), operate on it in the accumulator, and then move it back to memory (`STA`). Most of the instructions listed above can also access memory outside of the zero page by using the index register. See the programming manual for more information.

5.4 Reading and Writing Internal Registers

Registers for the various microcontroller functions such as the I/O ports are located in the address range $0000 through $003F and are accessed in the same manner as RAM memory. For example, the Port B Data Direction Register is at location $0001 and the following code stores the accumulator contents in this register

```
        sta     $0001
```

To avoid having to use numerical addresses, symbolic names for the register addresses are declared in one of the header files that is included at the beginning of the program. The above operation is then written as

```
        sta     DDRB
```

Names for the individual bits within the registers are also declared in the header file. These can be used in BSET, BCLR and other instructions that require a bit number. The following code sets bit number 2 in Port A.

```
        bset    PTA_PTA2,PTA
```

5.5 Declaring Variables and Constants

Variables that need to be in RAM for reading and writing are declared with the "`DS`" directive. This directive reserves memory for use by the program but does not initialize it to any value. Different versions of the `DS` directive can be used depending on size of the variable element being reserved.

```
x:      ds.b    1               ; allocates one byte for X
y:      ds.w    4               ; allocates four 16-bit words for Y
z:      ds.l    2               ; allocates two 32-bit longs for Z
```

Note that the `DS` directive only allocates space for the variable. It does not initialize the variable to any value.

Sections of ROM that need to contain constant values or strings can be allocated and initialized using the "`DC`" and "`DCB`" directives. The `DC` directive is used to allocate space for constants and initialize them to specified values.

```
a:      dc.b    "This is a string"
b:      dc.b    1,2,3,4,5       ; initializes B as five byes
c:      dc.w    $1234           ; initializes C as a 16-bit value
```

Unlike strings in C programs, strings that are allocated in this manner do not contain a zero byte as the last byte in the string. If the program requires having a zero byte to mark the end of the string this has to be added to the code allocating the string.

```
a2:     dc.b    "This is another string"
        dc.b    0
```

The `DCB` directive is used to allocate a block of ROM space and initialize the block to a specified value.

```
p:      dcb.b   40,$ff          ; initialize P to 40 bytes of $FF
q;      dcb.w   20,0            ; initialize Q to 20 words of zero
```

6 The PRM File

The PRM file controls where procedures and data are put in memory. It defines "SEGMENTS" which identify portions of memory and your code and variables can be placed in those segments. One or more code segments and data segments are defined to cover the areas supported by the microcontroller (0x60-0x25F for data and 0xBC00 - 0xFBFF for code). The example below from a PRM file defines two areas for variables (Z_RAM and RAM) and one for code (ROM). The "Z_RAM" segment covers the range 0x60 to 0xff where data can be accessed using only an 8-bit address. The remainder of the RAM area (0x100 to 0x25f) is covered by the "RAM" segment. The ROM segment starts at 0xBC00 and all the program code is stored there.

```
/* Here all RAM/ROM areas are listed. Used in PLACEMENT below. */
SEGMENTS
  Z_RAM      = READ_WRITE 0x0060 TO 0x00FF;
  RAM        = READ_WRITE 0x0100 TO 0x025F;
  ROM        = READ_ONLY  0xBC00 TO 0xFBFF;
END

/* Here all predefined and users segments are placed
   into the SEGMENTS defined above. */
PLACEMENT
  DEFAULT_RAM,               INTO RAM  /* non-zero page variables */
  PRESTART,                  /* startup code */
  STARTUP,                   /* startup data structures */
  STRINGS,                   /* string literals */
  DEFAULT_ROM                INTO ROM;
  _DATA_ZEROPAGE,
  MY_ZEROPAGE                INTO Z_RAM;  /* zero page variables */
END
```

The "PLACEMENT" section of the PRM file defines the names are associated with each segment. In your C code you can use the "`#pragma`" statements to indicate which segment code should be placed into (see the examples). It is probably a good idea just to use the example as a template for your PRM file and your allocation of code. More information on using `#pragma` statements for memory allocation can be found on page 536 of the Compiler Manual. Information for the PRM file can be found in the Linker Manual.

7 Building your Application

You can assemble and link your project by using the "Make" command under the "Project" menu. This will compile any source files that have changed since the last time the project was built, and then link it together. Whenever you successfully "make" your application, a file called "Project.abs.s19" will be generated in your project directory in the "bin" subfolder. This file contains the binary data that must be programmed into the microcontroller.

8 Sample Programs

On the computers in the EE 459 lab, in the "`JL16 Samples`" folder in the EE 459 account are the following files that may be of interest to those writing assemby language programs. These programs are also available on the EE 459 library web site.

jl16.asm This is a template file with the required declaration and initialization code. It can be used as a starting point for writing a program in assembler.

jl16-0.asm A very simple program for showing the micro is working. It loops forever turning bit zero in port A (PTA0) on and off as rapidly as possible.

jl16-1.asm This program reads a switch input and turns an LED on and off.

jl16-2.asm This program counts up and down on a seven-segment display. The program uses nested loops to implement the counter delay.

jl16-3.asm Similar to jl16-2.asm but uses an internal timer and interrupts to implement the delay. Use linker parameter file `CWJL16-3.prm` for this sample program.

jl16-4.asm Demonstrates interfacing to an LCD display using an 8-bit interface. Puts up a short message on the display.

jl16-5.asm Similar to jl16-4.asm but uses a 4-bit interface to the LCD

9 Programming the JL16 from CodeWarrior

Programming the microcontroller can be done from within the CodeWarrior software using the debugger functions. The EE 459 lab has a collection of programmers that consist of a blue USB module and a small section of PC board with a ZIF socket (Fig. 2). The blue module connects to one of the computer's USB ports and a short section of ribbon cable connects the module to the PC board with the programming socket. The following steps explain the process of programming the microcontrollers using these devices.

1. If CodeWarrior is not running start it from from the desktop icon and open the project. It can actually be started any time during the first three steps but must be running in order to do step 4.
2. Connect the programmer's USB cable to one of the computer's USB ports. If connecting to a Macintosh running CodeWarrior via the Parallels virtual machine software, the programmer must be connected to a USB 1.1 hub first and then the hub is connected to the Macintosh. The programmer will only work with the Macs if the programmer is connected as a USB 1.1 device and the hub acts to establish this type of connection.
3. Lift **up** the lever on the zero-insertion force (ZIF) socket and insert the chip into the socket. Pin one should be closest to the lever. Push the lever to the **down** position to lock the chip in. The lever must be in the down position when the chip is in the socket to make good electrical contact with the pins.
4. Start the CodeWarrior debugger by selecting "Debug" from the Project menu or by clicking on the icon of a green arrow and bug at the top right above the project box. This will bring up a dialog box (Fig. 3) that defines the connection between CodeWarrior and the development board.

   The first time this is done in a project CodeWarrior has to be told how to establish the connection to the programmer. At the top, under "Interface Details" CodeWarrior may have figured out that the USB programmer is present and it will say "P&E USB MON0 Multilink on USB1" or something similar. If CodeWarrior has not found the USB programmer the connection interface will be blank so click on "Add a Connection". In the "Interface Selection" dialog box, select "P&E USB MON0 Multilink on USB1" and then click "OK".

   Back on the Connection Manager screen in the "Power/Clock Detail" area, set the Device Power to "5 Volts, Provided by P&E Interface". Also set Device Clock to "Clock Driven by P&E Interface on Pin 13".
5. Once the connection settings are correct, click on "Contact Target with These Settings..." and it should open a connection to the programmer and chip.
6. If the debugger finds your chip it should put up dialog saying "Load image contains flash memory data. Erase and Program flash?" This means the debugger is ready to write the new binary program data you created when building the application into the microcontroller. Click on "OK" and it will go through several steps automatically programming and verifying the data.
7. When the programming is complete, exit the debugger.
8. Lift up the lever on the ZIF socket and remove the chip.

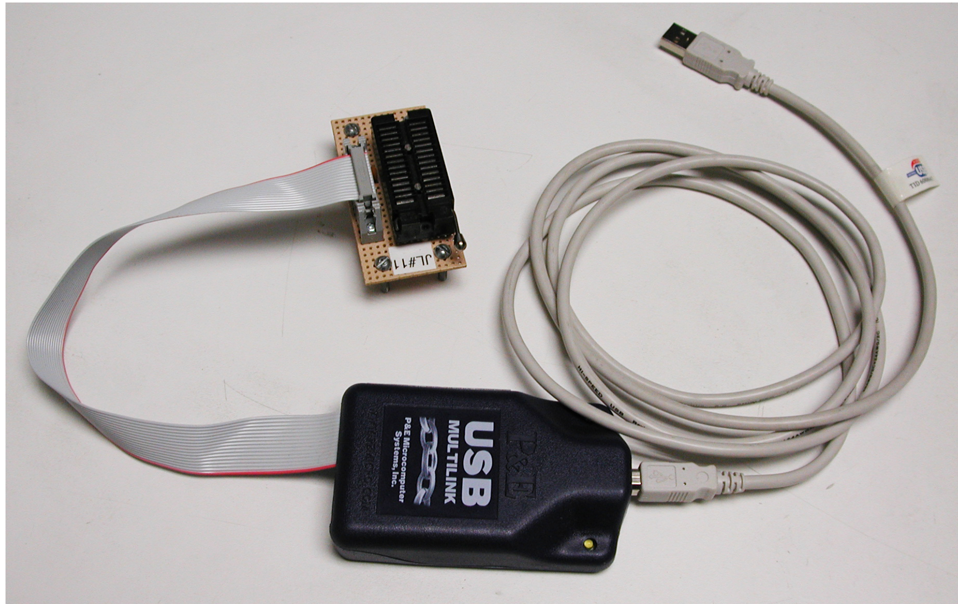

Figure 2: USB Programming Devices

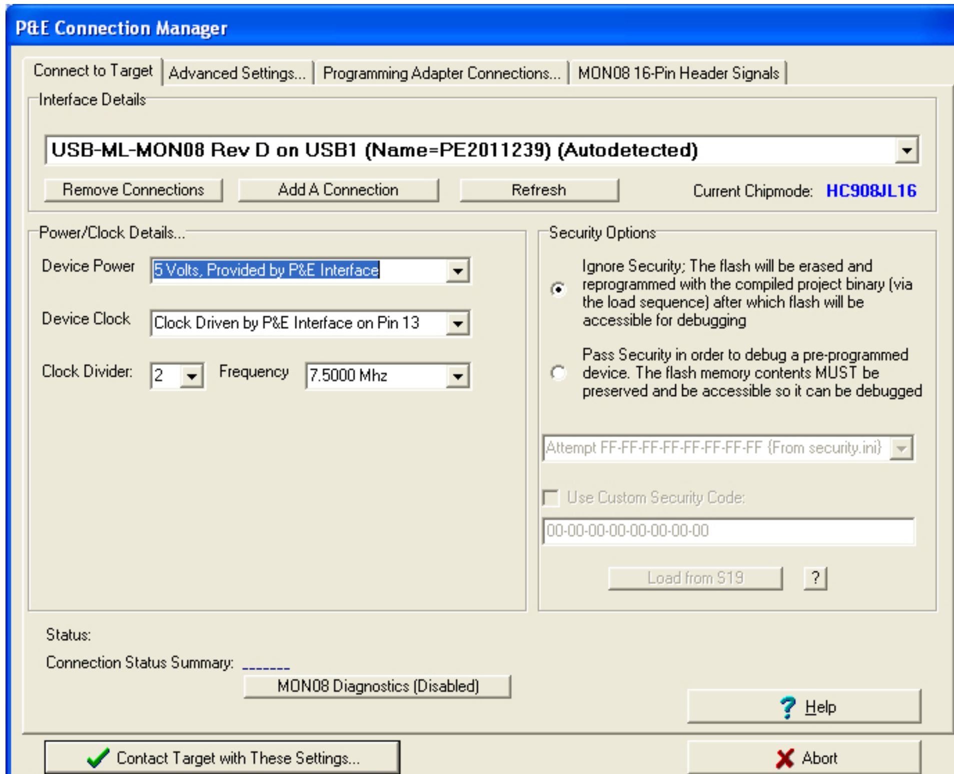

Figure 3: CodeWarrior Connection Manager for USB Programmers