COE538 Microprocessor Systems
Lab 1: Using the *CodeWarrior* IDE
and
Introduction to Assembly Programming[1]

Peter Hiscocks
Ken Clowes

Department of Electrical and Computer Engineering
Ryerson University

phiscock@ee.ryerson.ca
kclowes@ee.ryerson.ca

Contents

[1] This lab was adapted to be used with the HCS12 microcontroller by V. Geurkov.

1 Objectives

This tutorial introduces the basic lab environment used in the course. Once you have completed the lab, you will be able to:

1. Use the Serial Monitor program and CodeWarrior software for basic operations, such as examining and modifying memory locations and registers in the microprocessor system.
2. Using the 'hand assembly method', create and run a tiny program on the microprocessor system at full speed (with or without breakpoints).
3. Using a CodeWarrior's text editor, create a complete assembly language program, assemble it, load it into the microprocessor system, and run it.
4. Use breakpoint and memory dump techniques to debug problems with a source code program.

2 Preparation

Read these lab notes to get a general idea of the concepts and work required.

3 Requirements

To get credit for the lab, you must:

1. Setup the required environment.
2. Be able to demonstrate knowledge of how to use the basic Serial Monitor commands.
3. Demonstrate an assembly language program to multiply two integers. This should include a copy of the source file and may require demonstration of breakpoints.

4 Theory

A little bit of theory can help you understand the lab environment you will be using.

The *microprocessor board* and the *EEBOT robot* are the ultimate targets of the embedded systems you will design in the course. The HCS12 microprocessor[2] used in these boards is set up to run a special program, called the *Serial Monitor*, when it is turned on or the reset button is pushed. The Serial Monitor software reads user commands, executes them and displays information back to the user.

The Serial Monitor is designed to communicate with the user through a serial port connected to a dumb terminal. In the lab, the HCS12 board is connected to a serial port on the bench workstation which emulates a dumb terminal in a command-line window. The terminal program that runs on the workstation is implemented as a part of a software package - an Integrated Development Environment (IDE). The IDE that we will use in the course is called the CodeWarrior.

[2]Strictly speaking, the HCS12 chip is a *microcontroller* which in addition to a microprocessor comprises some extra circuitry.

Typing a command into the terminal emulator causes it to be sent to the microprocessor development system, where the command is executed by the Serial Monitor program. For example, typing in 'DB $4000' instructs the monitor program to do a 'Display Byte', i.e., send the contents of memory location at $4000.

The monitor program is a low-level (i.e., very simple and primitive) method of sending programs to the microprocessor development system and executing various debugging commands on it. As such, it is a very basic tool that is used in the early stages of getting software to work on the microprocessor development system.

The HCS12 system has a direct access to memory space of 64 Kbytes. Memory locations are specified with 16-bit numbers expressed in hexadecimal. Consequently, the memory space range is 0x0000–0xFFFF.[3] Using memory paging, the HCS12 can access even more space.

The complete memory map of the microcomputer development system is shown in figure 1 on page 6. Some of the memory locations are occupied by bytes of RAM, others by EEPROM or device registers, and some memory locations are not used at all. It is important to note that not all memory locations behave in the same way.

For example, if you write 0x1A to a byte of RAM and then read that location back, you will of course read what was just written. However, if you write to protected EEPROM, the contents of the byte will not be changed. Locations mapped to device registers can behave even more strangely.

4.1 Number Representations

When you interact with the Serial Monitor, byte contents are always input and displayed in hexadecimal format. As a programmer, however, it is often more convenient to think in terms of decimal numbers (signed or unsigned), binary representation or ASCII character codes. To use the Serial Monitor effectively, you should be able to translate between these different representations.

For example, suppose that a byte contains 0xB6. As a binary number, this is expressed as 0b10110110. (Note that we use the convention of numbering the individual bits of an n-bit binary number from n-1 to 0 where bit-0 is the least significant bit. In this case, we can say that bit-7 is a 1 and bit-0 is a 0.) If the number is meant to represent an unsigned quantity that is normally thought of a decimal number, you would convert 0xB6 to the decimal number 182. If, however, it was meant to represent a signed quantity, then you would convert it to -74.

As another example, a programmer may want to see if a 16-bit variable that should be 61466 really has that value. If the number is encoded into two sequential memory locations, the Serial Monitor allows you to see that the most significant byte is 0xF0 and the least significant byte is 0x1A. Simple conversion shows that, indeed 0xF01A (unsigned) is 61466. (Similarly, as a signed number, 0xF01A is equivalent to -4070.)

If you have forgotten how to convert between numbers of different bases or how the twos-complement method for representing signed numbers works, this is a good time to review them.

5 Lab Exercise: Setting up

1. Create a directory called `coe538` and change to that directory. The rest of the lab should be conducted from `~/coe538`.

   It is essential that you be well organized in this course, so we recommend that you

   - Create a separate subdirectory for each lab
   - Number in sequence each version of any program you write, so that you always know which is the most recent. (The time and date stamp on the file will also help with this.)
   - Carefully back up any work to USB memory stick (or use some other method of backing it up, such as emailing it to yourself.)

[3] The prefix `0x` is the *C style* method of indicating *hexadecimal* format. Sometimes the `$` is used for the same purpose. You should be familiar with both formats. In the case of the Serial Monitor program, it assumes that all numerical values are in hexadecimal, and no preface is used at all.

5.1 Using the CodeWarrior IDE

In order to be able to enter and execute Serial Monitor instructions, the workstation must run the CodeWarrior IDE in the *HCS12 Serial Monitor* mode. In addition to this mode, the CodeWarrior software supports a *Full Chip Simulation* mode that allows exercising the Serial Monitor instructions without using the real microprocessor and Serial Monitor program at all. The simulation mode has many advantages, one of which is that you can write, run and debug a microprocessor program at home, without connecting the real microprocessor board to the workstation.

We will start using the CodeWarrior IDE in a simulation mode first:

1. Launch the VMWare's Windows XP (to simulate Windows on a Unix machine) and start the **CodeWarrior**.
2. Click on the **Create New Project** button in the start-up screen.
3. Under **HCS12D** family, pick the **MC9S12DG128B** derivative and select the **Full Chip Simulation** connection. Click on the **Next** button.
4. Uncheck the **C** language and check the **Absolute assembly** language checkbox. Enter a project name (for example, **lab1.mcp**) and set the project directory location to your home directory, coe538. Click on the **Finish** button.
5. CodeWarrior has created the project directory and basic files necessary for a start of the development. The files in the project are listed in **Files** tab of the project panel. The **Sources** folder contains the **main.asm**, which is our source assembly program. Open the **main.asm** file by double-clicking on it's name.
6. The default main routine initially contains a sample code counting Fibonacci numbers in never-ending loop.[4] The main file includes the file **derivative.inc** that provides a binding to all necessary derivative-specific definitions of the control registers. The **derivative.inc** and the **mc9s12dg128.inc** (the MCU specific include file) can be found in the **Includes** folder of the project panel Files tab.
7. Make the project by clicking on the **Make** icon.
8. Start the *True-Time Simulator* for the project using the **Debug** icon. This will open several **windows** (such as **Source**, **Assembly**, **Command**, **Memory** windows, etc.).

   Note that the microprocessor board has not been communicating with the PC yet; it has just been fully simulated by the CodeWarrior IDE. For more details refer to chapters 3.2 and 3.8 of the textbook [4].

5.2 HELP command

You can now type commands to the (simulated) Serial Monitor using the **in**> line of the **Command** window. To execute a command, the appropriate character string must be followed by <carriage return>.

The command HELP, for example, causes the microprocessor system to list the available commands, as shown below:

```
VER         Shows the version of all loaded commands
LOAD        Loads an application (Code & Symbols)
EXIT        Terminates this application
RESET       Resets the target MCU
HELP        Displays available commands. To get help about a specific
            command, use the command and '?', e.g. 'LOAD ?'
BS          Sets breakpoint
STEPINTO    Step Into
STEPOUT     Step Out
STEPOVER    Step Over
RESTART     Restart execution
BC          Clears breakpoint
BD          Displays breakpoint(s)
G or GO     Starts execution (Go)
```

[4] We will modify the contents of this program shortly, so at this stage it's not important to understand what is going on in it.

```
S or STOP   Stops execution (Halt)
P           Executes an instruction (Flat step)
T           Executes CPU instruction(s)
RD          Displays registers
RS          Sets registers
DB          Displays byte(s) from target memory
DW          Displays word(s) from target memory (2 bytes)
DL          Displays long(s) from target memory (4 bytes)
WB or MS    Writes byte(s) into target memory
WW          Writes word(s) into target memory (2 bytes)
WL          Writes long(s) into target memory (4 bytes)
MEM         Displays Memory map
DASM        Disassembles target memory
SREC        Loads of Motorola S-records from a specified file
CALL        Executes commands in the specified command file
SPC         Shows the address given as argument
SMEM        Shows the memory range given as argument
FILL        Fills a memory range with the given value
DUMP        Dumps the content of the data component to the command line
CLR         Clears the Command window
PROTOCOL    Show communication protocol for Serial Monitor
```

Try it.

Notes

1. The commands are case-insensitive (so HELP and help are equivalent.)
2. The commands may be abbreviated by using only enough letters to make the command unambiguous. (For example, the only command starting with the letter `G` is `GO`; it may be entered as the single letter `G`. If you enter this command, you will run the program. To stop the execution, enter the letter `S`.)
3. Hitting the **Up Arrow** and **Down Arrow** keys scrolls through the list of previously typed commands. To invoke a command, press the **Return** key.

5.3 Display Byte(s) from target memory (DB)

One of the most commonly used commands allows the user to examine the contents of memory. The DB command has the following syntax: DB Addr, N

The address *Addr* (expressed as four-digit hexadecimal number) and the number *N* give the range of memory locations to look at. Each memory location is read and displayed as a two-digit hex number. Each line displays the contents of 16 sequential bytes of memory. The initial 4-digit number is the address of the first byte displayed.

Example

The command `db $4000,32` gives the following (or equivalent) output:

```
4000: CF 40 00 10 EF 86 FF 7A 02 44 7A 02 62 41 7A 02 .@.....z.Dz.bAz.
4010: 42 B6 02 41 41 7A 02 60 20 F7 FF FF FF FF FF FF B..AAz.` .......
```

Notes:

1. In this case, the first byte displayed 0xCF is located at address 0x4000. This is equivalent to the unsigned decimal number 207 or, if interpreted in twos-complement format, to the signed decimal number -49.
2. The byte at location 0x4003 is 0x10. This is equivalent to decimal 16 (in both signed and unsigned formats).
3. Note that following the 16 hex numbers are an additional 16 hex numbers of single characters (one of which is the space character). For example, the contents of 0x4013 is 0x41 which corresponds to the ASCII code for the character `A`.

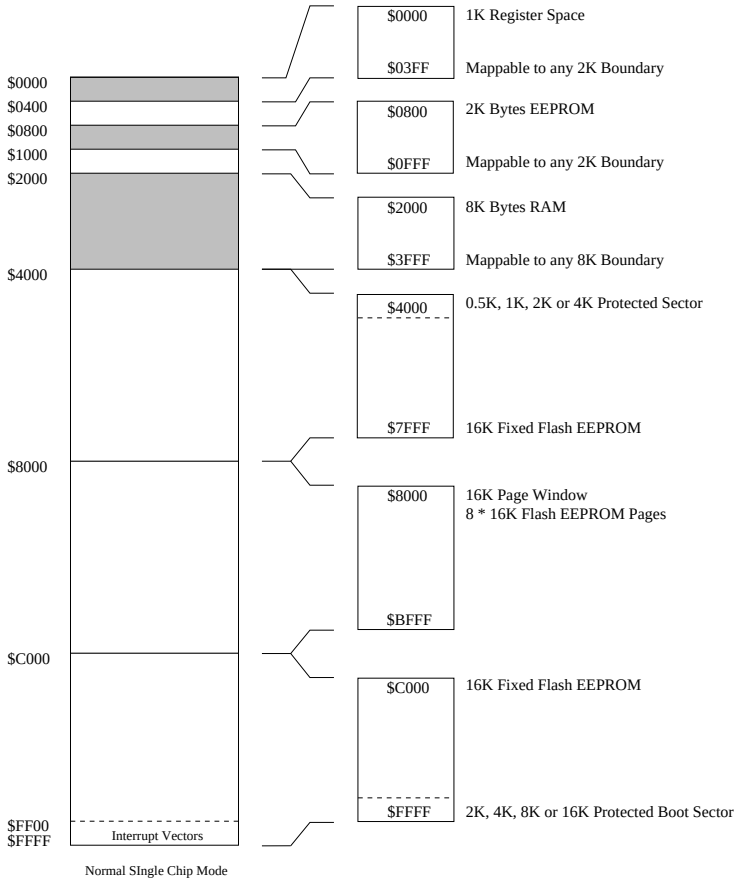

Figure 1: Memory Map

The complete memory map is shown in figure 1. Notice the Register, RAM and EEPROM areas. The 2-Kbyte Serial Monitor program begins at memory location $F800 and ends at location $FFFF. (The '$' sign indicates hexadecimal notation).

Immediately following **reset**, control is passed to the Serial Monitor which makes few changes:

1. It relocates 2K EEPROM to the address range 0x0800 to 0x0FFF from the default location of 0x0000 to 0x07FF (see figure 1).
2. It relocates 8K RAM to the address range 0x2000 to 0x3FFF from the default location of 0x0000 to 0x1FFF.
3. Fixed Flash Memory space (16K each) is not relocated: 0x4000 to 0x7FFF and 0xC000 to 0xFFFF (the space 0xFF00 to 0xFFFF is reserved for interrupt vectors).
4. The Register space (1K) is not relocated: 0x0000 to 0x03FF.
5. It changes the clock rate from default 4MHz to 24MHz.

5.4 Exercises

1. Display bytes from memory locations at 0x4020 to 0x4027.
2. What are the contents (in hex, binary, decimal both signed and unsigned and ASCII) of memory location at 0x4024?

5.5 Write Byte(s) into target memory (WB)

The WB command allows modifying the contents of one or more memory locations. The syntax for the command is: WB Addr Data

For example, typing the command:

```
>wb $3000 $41 (then hit the enter key)
```

changes the contents of memory location at $3000 to $41

Example: If you type in `wb $3000 $41 $42 $43`, the contents of memory locations at $3000 to $3002 will change to $41, $42, $43.

A db command then confirms that the locations have been modified as shown below:

```
>db $3000,3
3000: 41 42 43                                          ABC
```

Therefore, multiple bytes may be modified in one `wb` operation

5.6 Questions

Determine the ASCII character codes for the first four (4) letters of your UserID (i.e. the user name you type in response to the "Login:" prompt on the workstations.) Modify memory locations 0x3000-0x3003 so that they contain the 4 ASCII codes and ensure that the next memory location (0x3004) contains 0x00. Show the sequence of commands required and the resulting memory contents.

6 Devices: Hardware Input and Output

The DB and WB commands can be very useful in debugging hardware that is attached to the microprocessor. Often, the hardware signal lines of an external device are 'mapped' into the memory address space.

Hardware Inputs

For an input device (signals from an external device into the microprocessor), this means that each logic signal appears as one logic bit in some location of memory. Using the DB command, one can read that location to determine whether a particular electrical signal is being read as a logic 0 or 1. The DB command produces an output in the form of a hexadecimal byte, so it's necessary to convert the data to binary format to interpret it properly.

For example, if a particular hardware data input location read as $AA, then it's binary value is 10101010. This tells us that bit zero is read as zero volts, bit 1 as +5 volts, and so on.

Hardware Outputs

When hardware output lines are mapped into the address space at a particular location, the logic level on these particular lines may be changed from zero volts to 5 volts by changing the contents of some memory location from logic zero to logic 1. For example, in the lab robot, the starboard motor enable is bit (something) of location (something). A logic 0 turns the motor off, a logic 1 turns it on.

6.1 Register Set (RS)

The HCS12 microprocessor contains a number of 'machine registers'. A diagram of these registers, called the 'programming model', is shown in figure 2. (See the 'HCS12 Reference Manual').

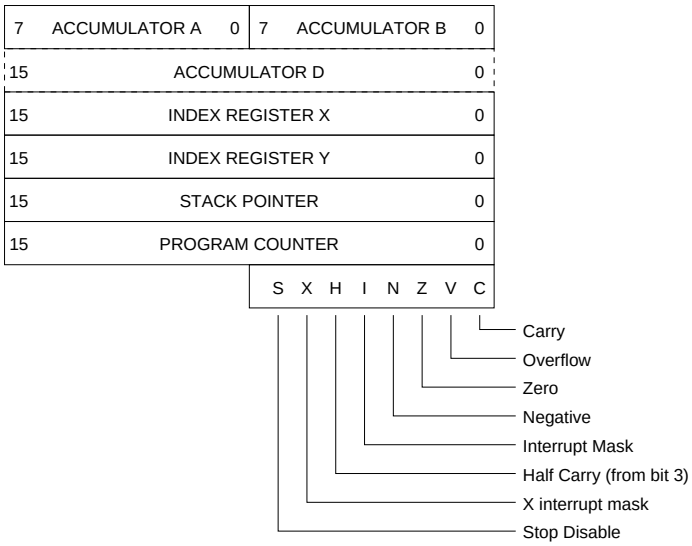

Figure 2: HCS12 Programming Model

This diagram represents the registers that are internal to the central processing unit of the microprocessor. It ignores for example, input and output registers that control external hardware.

The registers have various functions:

Accumulator A (8 bits):	Math operations and general purpose storage
Accumulator B (8 bits):	Math operations and general purpose storage
Accumulator D:	Operations that take place on both A and B are treated as taking place on the 16 bit D register. D may be regarded as a 16 bit accumulator.
Index X (16 bits):	Pointer operations
Index Y (16 bits):	Pointer operations
Stack pointer SP (16 bits):	Pointer into the stack region
Status Register (8 bits):	Various device flags
Program Counter PC (16 bits):	Pointer to the address of the next instruction to execute

For our purposes at this point, the registers of interest are the `ACCUMULATOR A` and the `PROGRAM COUNTER`.

When the monitor program starts, it copies the machine registers to a storage area in desktop computer's RAM. Using the monitor Register Set instruction `RS <Register>=<value>`, the programmer may modify the values of these various stored register contents. When the programmer executes the GO instruction, the monitor program rewrites these new register values into the machine registers and starts program execution at the memory address specified by the contents of the Program Counter register. To set the contents of the Program Counter to e.g. $1234, you must type `RS PC=$1234` (You can also start a program at the location $1234 by entering the command GO 1234).

If you only need to observe the contents of a register, you can similarly use the Register Display instruction `RD <Register>`.

- Execute the `RD` instruction and examine the contents of each machine register listed above.

Note that the **Register** window immediately reflects all updates of the register contents.

Now we have enough information to enter and run a small program, using a method called 'Hand Assembly'.

7 Hand Assembly

Using a method called 'hand assembly', it is possible to use the Write Byte and Register Set commands to install a program on the development system. We'll do this with a program to increment (increase by 1) the contents of the accumulator register. In symbolic form the program is:

```
INCA    ; Increment the A accumulator
INCA    ; ditto
SWI     ; break to the monitor program.
```

Each INCA instruction increases the contents of the accumulator A by 1. If we initialize ACCA to 00, it should contain 02 when the program breaks to the monitor.

SWI stands for SoftWare Interrupt. For now, it's only required to understand that an SWI command causes the machine to terminate the current program and switch to the monitor program. The details of how this occurs require some understanding of the stack mechanism and interrupts, and is in the Technical Manual.

7.1 Exercise

Manually convert each instruction into its corresponding operation code, and then enter the program into the microcomputer. In detail:

- Use the HC12 Reference Manual to look up the 'operation code' for the INCA instruction and the SWI instruction. These instructions have a 'one byte' op-code, which keeps things simple. Other instructions could require more bytes.
- Decide on a suitable starting address (the 'Origin') for the program. The RAM location $3000 is a suitable choice.
- Use the `WB` command to enter the sequence of three bytes, starting at the chosen origin.
- Use the `DB` command to verify that the instructions have been stored correctly.

Next, you need to set up the machine registers to run the program:

- Use the `RS` command to set the accumulator A (also referred to as ACCA) to 00.
- Use the `RS` command to set the PC to the value of the Origin address
- Use the `RD` command to examine the register contents and ensure that they are correct.

Now run the program by executing the Go (`G`) command.

The computer should break to the monitor with the program counter pointing at the next byte that would be executed. The accumulator ACCA should contain the expected result. Check that this is the case.

Note that the original program that calculates Fibonacci Numbers still resides in the memory and can be invoked in the same way.

Now you can close the *True-Time Simulator & Real-Time Debugger* window.

This 'Hand Assembly' process is far too tedious and error-prone to use for entering anything more than a few lines of program code. It is however useful for modifying an existing program during debugging, a process known as 'patching' the code.

8 Writing an Assembly Language Program

An assembly language program is the symbolic description of a program that is to be translated into machine language and loaded onto the microprocessor. For a high level language, such as C, one instruction usually corresponds to many machine instructions. For assembly language, each instruction in the source code corresponds to one machine instruction. The resultant translation into machine language is referred to as 'the program binary'.

The process is as follows.

Doubleclick on the `main.asm` file in the upper left part of the *Freescale CodeWarrior* window. This file contains the assembly language source program. In the main.asm file delete the following portion of the automatically generated code for Fibonacci numbers:

```
            LDS   #RAMEnd+1         ; initialize the stack pointer
            ...
            ...
            RTS                     ; result in D
```

Copy the following code lines into the file `main.asm`:

```
            LDAA FIRSTNUM           ; Get the first number into ACCA
            ADDA SECNUM             ; Add to it the second number
            STAA SUM                ; and store the sum
            SWI                     ; break to the monitor
```

Make the following extra changes to `main.asm` and save it. The final version should look like:

```
*****************************************************************
*    Demonstration Program                                      *
*                                                               *
* This program illustrates how to use the assembler.            *
* It adds together two 8 bit numbers and leaves the result      *
* in the 'SUM' location.                                        *
* Author: Peter Hiscocks                                        *
*****************************************************************
; export symbols
            XDEF Entry, _Startup    ; export 'Entry' symbol
            ABSENTRY Entry          ; for absolute assembly: mark
                                    ; this as applicat. entry point
; Include derivative-specific definitions
            INCLUDE 'derivative.inc'
*****************************************************************
* Code section                                                  *
*****************************************************************
            ORG   $3000 Go to $3000

FIRSTNUM    FCB   01                ; First Number  Save 01 under firstnum
SECNUM      FCB   02                ; Second Number Go to 3001, save under secnum
SUM         RMB   1 how many byte   ; Result of addition reserve space for later use
*****************************************************************
* The actual program starts here                                *
*****************************************************************
            ORG   $4000 Go to $4000 and stay
 Entry:
_Startup:
            LDAA  FIRSTNUM          ; Get the first number into ACCA load first sum into ACCA
            ADDA  SECNUM add to A   ; Add to it the second number
            STAA  SUM stay          ; and store the sum
            SWI                     ; break to the monitor
*****************************************************************
* Interrupt Vectors                                             *
*****************************************************************
            ORG   $FFFE
            FDB   Entry             ; Reset Vector
```

The features of this program are as follows:

- A line beginning with an '*' is ignored by the Assembler and may be used for commentary. The comments are extremely important. Assembly language is difficult to understand at the best of times, so header comments are vital.
- Comments may also be added to individual text lines, after a semicolon.

- ORG is an 'assembly directive'. It does not translate directly into machine code, but advises the Assembler. In this case, it specifies where the program is to be located. We have two ORG statements. The first one defines the start of 'working storage' in RAM, where variables will be stored. The second ORG defines where the 'program text' starts.

- Notice that hexadecimal values must be designated with a leading $ sign. Without the dollars sign, the value is interpreted as a decimal value. This is an easy mistake to make, but it's obvious if you check that the program loaded correctly into the memory area you expected.

- FIRSTNUM, SECNUM and SUM are 'symbolic addresses'. They will be translated into specific addresses in the memory space. An assembly language program must adhere to a relatively strict format. In the case of this assembler, it is required that **only symbolic addresses begin in the first column, all other text (such as directives) must be indented at least one space or tab**. Notice that the symbolic names reflect their function in this program. This is an important documentary clue for those unlucky souls (including yourself, perhaps) that have to read and understand this program at a later date. Symbolic names can be up to 13 characters long and can contain the understroke character.

- The FCB (Form Constant Byte) directive reserves one byte in memory and initializes it to some value. Notice how symbolic names, FIRSTNUM and SECNUM, are attached to this reserved space.

- The RMB (Reserve Memory Byte) reserves one or more bytes in memory. They are not initialized to any particular value. Notice how a symbolic name, SUM, is attached to this reserved space.

Two other important directives are FCC and FDB:

FCC Form Constant Character. This directive is used for defining a string message in memory. For example, the following stanza could be used to define a null-terminated string with symbolic starting address 'HELLO':

```
HELLO       FCC 'HELLO WORLD V96'
            FCB $00
```

The characters between single quotes will be translated into their ASCII values and stored, one per byte, in memory.

FDB Form Double Byte. This directive reserves and initializes two bytes of memory. It's often used to store a 16 bit address in memory. For example, the following stanza will initialize the location VECTOR to the 16 bit address HELLO:

```
VECTOR      FDB HELLO
```

8.1 Assemble the Program

We have just created an assembly program that resides in the CodeWarrior environment. This program must next be assembled. But this time, we will execute it on a real microcontroller. So the debugging connection in the project panel has to be changed from the **Full Chip Simulation** to the **HCS12 Serial Monitor** mode. Make sure you have changed the mode and click on **Make** icon in the project panel. This will invoke the assembler. If there are no errors (caused by mistakes in typing, for example), all appropriate files will be created in the Project folder. Otherwise, error messages will appear on screen. The assembler converts the assembly program into the program binary (or machine code). It also generates an encoded form of the program binary, `Project.abs.s19` that is later uploaded to the microprocessor. The format of the encoding is known as 'S-Records'. The S-Records file includes address information that specifies where the binary should reside in microprocessor RAM, and checksums that can be used to ensure that the uploading occurred correctly.

The assembler used in this exercise generates 'absolute code': the binary is assembled to run at its final location. This is relatively simple to use and understand, but it does not support the inclusion of library routines.

8.2 Upload the Program

At this point, you need to power up the connected microprocessor board. Next, start the debugger program by clicking on the project panel **Debug** icon. Select the appropriate COM port and confirm the **Erase and program flash** dialog with the OK button. This will upload the machine codes into the HCS12. The upper left window **(Source)** of the debugger will contain the 'Source code' and the upper right window **(Assembly)** will display the 'Assembly code'. The Assembly window shows where in the memory map the instructions are stored. The memory map appears in the lower right window **(Memory)**.

Click on the **Assembly** window. From the `Assembly` menu of the debugger select `Display` and click on `Code`. Now you can see the listing view of the assembly code. It will look like this:

```
3000 01            MEM
3001 02            INY
3002 00            BGND
...
4000 B63000        LDAA    0x3000
4003 BB3001        ADDA    0x3001
4006 7A3002        STAA    0x3002
4009 3F            SWI
...
```

This is an extremely important information for debugging the computer program.

- In each line, the first number is the absolute address where the instruction or data will be stored. Using this listing file, we know what computer memory locations are used by the program. We can use the `DB` instruction to check these locations to ensure that the correct instructions are at those locations, and to read out the program data.

  Note that you can observe and/or modify memory contents directly in the **Memory** window. To observe a specific memory location, just scroll the screen to that location. Double click the location and modify it, then hit the `Tab` key. The cursor will move to the next location that can be altered in the same way. It is important to keep in mind that you are in the HCS12 Serial Monitor debugging mode. Therefore any modification will apply to the real microprocessor memory. Make sure that you don't corrupt your program (the RAM locations that you can modify range from $2000 to $3FFF, see figure 1).

- The symbolic instructions are converted into op-code and operand. Notice how the instruction `LDAA FIRSTNUM (0x3000)` is converted into a one-byte op-code `B6` and two-byte operand `30 00` that corresponds to the absolute address of FIRSTNUM. Refer to the LDAA instruction in the HCS12 Reference Manual and confirm that the op-code is correct for an absolute address.

  The SWI instruction converts into a single-byte operand.

- If there is an error (which is the usual case when first assembling a program), the assembler will announce it. It usually takes several tries to get an error free assembly of a new program. However, notice that an absence of errors in the assembly code is no guarantee of correctness. A syntactically correct program can represent something that is logical nonsense.

When the source code `.asm` file is finally correct, the assembly operation will produce an `S19` file called `Project.abs.s19`. The `S19` file is an encoding of the assembled program that can be loaded into the microprocessor.

Use an editor to examine the S-record file `Project.abs.s19` that is stored in the `bin` folder of the project directory. It will look something like this:

```
S0400000433A5C4D792046696C65735C52796572...2732D
S1063000010200C6
S10D4000B63000BB30017A30023FF5     S1124000CF400010EFB63000BB30017A30023FE2
S105FFFE4000BD
S9030000FC
```

The S-record format was created for data block transfers by Motorola Inc. In the above notation, the first character, *S*, signifies a start of the record. The next digit defines a type of the data block (*0* - a vendor specific header; *1* - a data block; *9* - an end of a data block). The next two hex digits indicate the number of bytes that follow in the rest of the record. The next four hex digits determine a memory address where the information should be stored. Next follows a sequence of *2n* hex digits, for *n* bytes of the information itself. Finally, the last two hex digits is a checksum, the *1*'s compliment of the *mod 256* sum of the byte count, address and data fields.

For example, in the following *S1* record, `S10D4000B63000BB30017A30023FF5`:

- Hex *0D* indicates that *13* bytes will follow.
- Hex *4000* specifies the destination for the first byte of the data.
- The rest of the record (excluding the last byte) constitutes our program binary (or machine code).
- The last byte is calculated as follows:

$$\overline{0D + 40 + 00 + B6 + 30 + 00 + BB + 30 + 01 + 7A + 30 + 02 + 3F} = F5$$

For our purposes, it is enough at this point to know that the `S19` file exists and it is loaded into the microprocessor.

8.3 Check the Program

Once the program is loaded, you should check that it is actually present on the development system. Use the `DB` command or look at the memory map displayed in the **Memory** window. Compare the contents of the memory with the information shown on the listing (Assembly) window. They should match.

8.4 Run the Program

Now you can execute the program by selecting **Menu** $>$ **Run** and clicking on **Start/Continue**. (Alternatively, you can start the program with the command `G $4000` or just `G`).

The program should break to the monitor and display the contents of the machine registers. Check that the memory location SUM contains the correct result.

Use any method to change the contents of `FIRSTNUM` to `0A` and `SECNUM` to `FF`. Reset and rerun the program; verify the result.

8.5 Questions

1. Memory location 0xFFFF is in the protected flash EEPROM. What happens if you try to change it using the `WB` command to something else? Explain.
2. The HCS12 contains 2 Kbytes of EEPROM memory starting at location $0800. EEPROM is memory cells that can be individually changed, and will contain their contents even when the power is turned off. The EEPROM write cycle is normally quite slow (by computer standards). A write to RAM occurs in one machine cycle. A write to EEPROM requires longer time. However, recent advances in microelectronic industry have allowed implementing an EEPROM that is able to complete writing in one machine cycle. Check whether memory locations in the EEPROM area can be modified by the `wb` command. Try to modify it again. Explain the result.

9 Breakpoints and Debugging

In general, computer programs do not work correctly the first time they are run, and they must be debugged. The monitor Breakpoint instruction is very useful for this.

When you insert a breakpoint in a program, the program will exit to the monitor at that point. Then you can examine memory and the machine registers to determine what went wrong.

If everything seems correct at that point, then you can cancel that breakpoint and insert a new breakpoint farther along in the program.

For example, the command

```
BS $400B
```

would insert a breakpoint at address $400B, and the computer would switch to the monitor when the program counter reached that address, ie, just before the machine executes the instruction at address $400B. Notice that the breakpoint must correspond to the address of the instruction op-code. It can't refer to an address part way through an instruction.

It is possible to insert several breakpoints at the same time, but to keep things clear it's usually best to work with one at a time. However, there is one situation where multiple breakpoints are useful. If one is not entirely sure which branch a computer program is taking, you can put breakpoints in both branches and see which breakpoint is activated.

The command

```
BC $400B
```

removes the breakpoint at $400B.

Insert a program breakpoint so that the program breaks just after the addition of the two numbers. The sum should be in the A accumulator at that point.

In order to know the address at which to insert the breakpoint, you will need a copy of the assembled program.

Rerun the program and verify that the program does break correctly and that the number in the A accumulator is correct.

Note that you can insert a break point by right clicking on the assembly instruction and selecting *Set Breakpoint* from the menu.

9.1 Crash during Breakpoint

The breakpoint mechanism works by replacing the instruction op-code at the breakpoint with an `SWI` instruction. For this reason, the program under test must be in read-write memory (RAM). For example, a program in EPROM cannot be breakpointed.

Now, if there is a breakpoint set in a program under test, then an op-code has been replaced with an `SWI` instruction. The monitor program has a record of this replacement so that it can restore the op-code when the breakpoint is removed using the `BC` command. However, suppose there is a breakpoint set in the program and the program crashes. You then have to press the `RESET` button to get the monitor program back. The `RESET` wipes out any record of the original breakpoint, and the `SWI` instruction is still installed somewhere in the program text. Consequently, the BC command will not work to remove the original breakpoint.

Faced with this situation, you could manually patch the code by replacing the breakpoint `SWI` instruction with the correct op-code information, which can be determined from the listing file. Alternatively, you would simply reload the program.

The moral is this: if you have any doubt at all about the program being corrupted after a crash, reload it.

10 Assignment

This assignment is due in the next week and must be demonstrated in the opening minutes of the lab session in the next week.

Using the previous program as a model and referring to the HCS12 Reference Manual, write, assemble and test a program to multiply two 8 bit numbers together. You may use the `MUL` instruction to do the multiplication operation. The input values should be held in two 8 bit locations named `MULTIPLICAND` and `MULTIPLIER`. The result should be held in a 16 bit location named `PRODUCT`.

The program should be written as an assembly language source code. It should have a descriptive header and include comments in the code. For example, your header should indicate that this is an *unsigned* multiply operation.

Demonstrate this program to your lab supervisor, using the first digit and last digit of your student number as inputs to the program. The lab supervisor will select two other input digits to use in your program.

You should be prepared to show and explain the contents of the listing display, and to demonstrate the use of breakpoints as applied to the operation of this program.

11 Additional Readings

The fastest route to getting an understanding of assembly language is to read code that other people have written. One place to look is in the course library: `/home/courses/coe538/lib`. This directory contains useful example routines which you can use in your own programs without penalty. As well, the Web page of Jim Koch, the Senior Tech Support Engineer for Electrical and Computer Engineering, has a link to the source code of another monitor program (for the HC11 microcontroller) - Buffalo Monittor. This is a very large assembly language program, but there are many useful ideas in the code.

Reverse engineering other people's code is hard work at first, but once you get a basic understanding of the most common instructions and groups of instructions, it will go much more easily.

References

[1] **eebot Technical Description**
Peter Hiscocks, 2002
A complete technical description of the *eebot* mobile robot.

[2] **CPU12 Reference Manual**
Motorola Document CPU12RM/AD REV 3, 4/2002
The authoritative source of information about the 68HC12 & HCS12 microcontrollers.

[3] **68HC11 Microcontroller, Construction and Technical Manual**
Peter Hiscocks, 2001
Technical information on the MPP Board, 68HC11 Microprocessor Development System
Information on programming and interfacing the M68HC11 MPP Board.

[4] **HCS12/9S12: An Introduction to Software and Hardware Interfacing**
Han-Way Huang
Delmar Cengage Learning, 2010
A basic text on the HCS12 microcontroller.

Production Note

This file was created using the LaTeX typesetting program. Diagrams were created with the `xfig` program. Previewing was by `xdvi` and conversion to postscript format by `dvips`. Previewing of the postscript format used `gs` and conversion to `.pdf` format using the script `ps2pdf`.