Getting Started with the HCS12 IDE

B. Ackland

June 2015

This document provides basic instructions for installing and using the MiniIDE Integrated Development Environment and the Java based HCS12 simulator. These two programs will allow you to enter a program in HCS12 assembly language, assemble your code into a S19 machine code file, load the machine code into the instruction level simulator and simulate the execution of your code. The MiniIDE is also used as the front-end for your lab work. Code assembled by this program can be downloaded to the lab EVB boards for execution on a real HCS12 processor as part of your lab work. The following instructions assume you are installing the software in a Microsoft Windows environment.

1. Installing MiniIDE

Download the file *HCS12_IDE.zip* by clicking on the **HCS12_IDE** link on the course website. Extract the contents of the zip file. Move the extracted HCS12_IDE files to a convenient place in your file system. Execute the installation file *miniide.msi*. Choose the **Typical** setup type. This will install the MiniIDE software on your machine.

Create a new folder *HCScode* (or any other name you would like to give it) in a convenient place in your file system. This will be your working directory (folder) in which you will write and assemble code for the HCS12. Copy the four files *equates.asm*, *dbug12.asm*, *dbug12.lst* and *DBUG12.S19* from the HCS12_IDE folder to your *HCScode* folder.

2. Installing 68HCS12 Simulator

Return to the *HCS12_IDE* folder. The simulator *simhc12.jar* is provided as a Java executable. Move the file *simhc12.jar* to your desktop or some other convenient place for execution. Note that in order for this program to execute, you will need to have *Java* running on your PC. If you do not currently have a version of *Java* on your machine, you can load *Java* by going to *www.java.com/download* where you can get a free download of the latest version of *Java*. Once you have *Java* installed, the file *simhc12.jar* will display a Java icon. If you double-click on it, a pop-up simulator window will appear (as shown in Figure 2).

3. Using the MiniIDE

Launch the *MiniIDE* from the *START* menu of your PC. When first started, the *MiniIDE* appears as shown in Figure 1. The top half of the window is the area in which you will edit and view your assembly code. Below this area you may see up to two small sub-windows. The upper one is used for diagnostic output from the assembler. The lower window provides output for a terminal emulator when used with an evaluation board. This is the window you will use in your lab sessions to communicate with the EVB. If you do not have an assembler output sub-window showing, click on *Windows* in the **View** pull-down menu and turn on the *Output* window.

3.1 Entering your program

Click **New** on the *File* pull-down menu. This will load a “blank sheet” into the main window with an active cursor ready to receive your code.

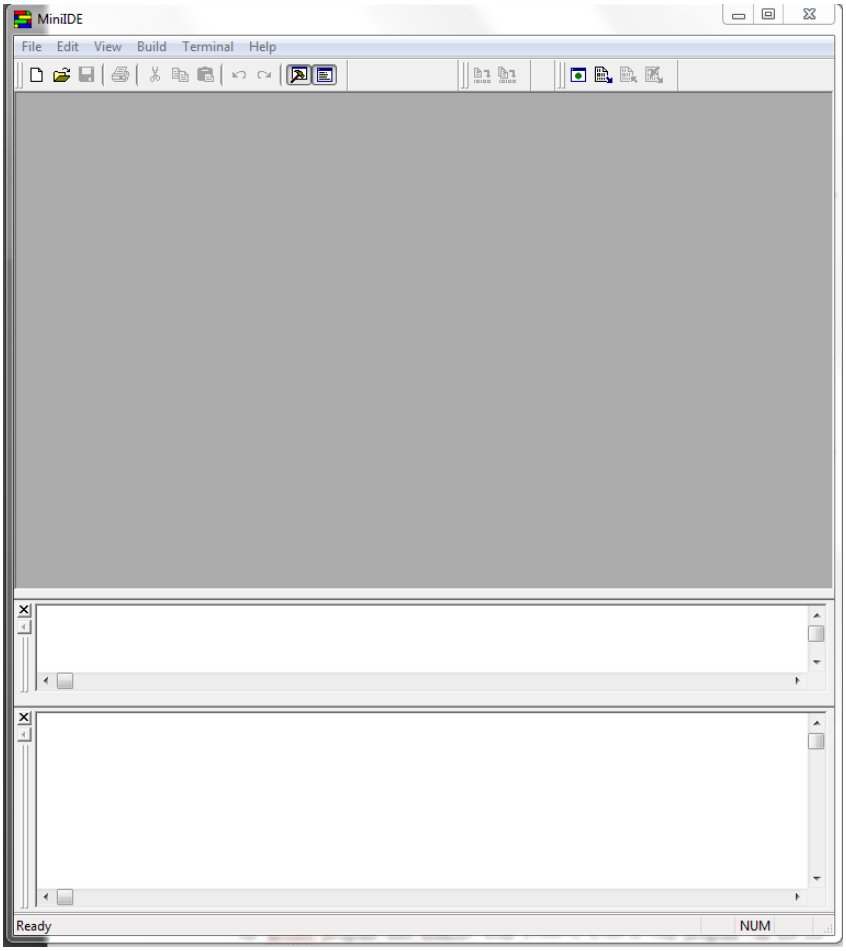

Figure 1. MiniIDE on startup

Enter the code shown on the following page:

```
        ORG   $5000
Adata:  dc.b  $23          ; byte initialized to 23 (hex)
Bdata:  dc.b  $5A          ; byte initialized to 5A (hex)
Rdata:  ds.b  1            ; one byte set aside to receive result
        ORG   $4000
Start:  ldaa  Adata        ; load A with contents of A data
        ldx   #Bdata       ; load X with address of Bdata
        ldy   #Rdata       ; load Y with address of Rdata
        adda  0,X          ; add data pointed to by X to acc A
        suba  #10          ; subtract 10 (decimal) from result
        staa  2, Y+        ; store final result
        swi
        END
```

This is a simple program to exercise the assembler and simulator. It adds the two numbers stored in memory locations *Adata* and *Bdata*, subtracts 10 (decimal) from the result and then stores the answer in memory location *Rdata*. The SWI instruction causes a software interrupt which will stop execution when running on the simulator. Use the **Save As** command on the *File* pull-down menu to save your program as *simple.asm* in your working folder *HCScode.* Make sure you understand what this code is supposed to do. What do you expect the values of accumulator A, registers X and Y and memory location *Rdata* to be once this program has executed?

3.2 Assembling your program

Click **Build simple.asm** on the *Build* pull-down menu. This will run the assembler on your program. The assembler will print out a diagnostic message in the assembler output sub-window indicating how many errors were found. If there are errors, you may have mistyped when entering the code. Scroll up on the output window to see the details of the error(s) and correct them. If you do not have any errors, you may want to deliberately introduce one just to see how the assembler responds. Once you have corrected any errors, save the file and run the assembler once again to produce an error-free output.

In addition to the diagnostic output message, the assembler produces two new files in your working folder. The first will be *simple.S19.* This is a text file that contains the machine code (in a special format) that will be loaded into the simulator or into the physical microprocessor on the EVB board. The second is a listing file that will be named *simple.lst*. Open this file in MiniIDE using the **Open** command on the *File* pull-down menu. Set the *Files of Type* pull-down field in the *Open* pop-pup menu to **Listing Files (*.lst).** Select *simple.lst* and click **Open**.

The listing file shows your program text together with the address and machine codes of each instruction in your program. Each line consists of:

(a) The line number for your original code. This is very useful if trying to locate an assembler error (identified by line number) in a large program.

(b) The memory address of the first byte of this instruction

(c) The instruction opcode (1-2 bytes)

(d) The instruction operand address bytes (0-5 bytes)

(e) The original assembler line of code

The listing file also shows any memory locations (along with their labels) that have been initialized with data (*Adata* and *Bdata* in this case) or set aside to hold variables during execution (*Rdata* in this example). At the bottom of the file is a summary of all the labels that were defined as part of the assembly process. Again, this is useful in a large (multi-page) program where it might be difficult to visually locate a label.

The listing file can be very useful in debugging a program. Knowing the memory address of each instruction and the actual machine code data can help identify where in the instruction sequence something is going wrong.

4. Using the Simulator

Launch the simulator *Simhc12.jar* from the desktop (or wherever you stored it). The simulator control panel appears in window as shown in Figure 2.

On the left of the window are fields showing the values of accumulators A, B and D and registers X, Y, PC and SP. You can change the contents of any register by simply typing into these fields. The D register is changed by writing to A and B (the upper and lower bytes of D). To the right of the registers are the 8-bits of the condition code register. A tick indicates that a bit is set to '1'. The absence of a tick indicates this bit is set to '0'. These bits can be toggled by simply clicking the box.

At the bottom of the window is a Memory Display area when you can display and modify the contents of memory locations. Memory is displayed four lines at a time. Each line corresponds to 16 memory addresses. The first line starts at the address specified using the *Address* field and the **Show** button at the top of the memory display area. Each line shows the contents of each of its 16 8-bit memory locations as a 2-digit hex number. It also displays (on the right side of the window) the line as a string of 16 ascii characters. If the value in a particular memory location does not represent a printable ascii character, it is simply shown as a blank space.

Individual memory locations can be changed by clicking on the current hex data in the memory display window. A red cursor will appear. Simply type in the new hex character (0-F) and this will overwrite the old value. The cursor will then move on to the next memory location. You can explore a larger region of memory using the scroll buttons on the right of the memory display area.

The three fields at the very bottom of the window allow you to fill a section of memory with the same constant value. Simply set up the *Start* and *End* address along with the data *Value* you want entered and click the **Fill** button.

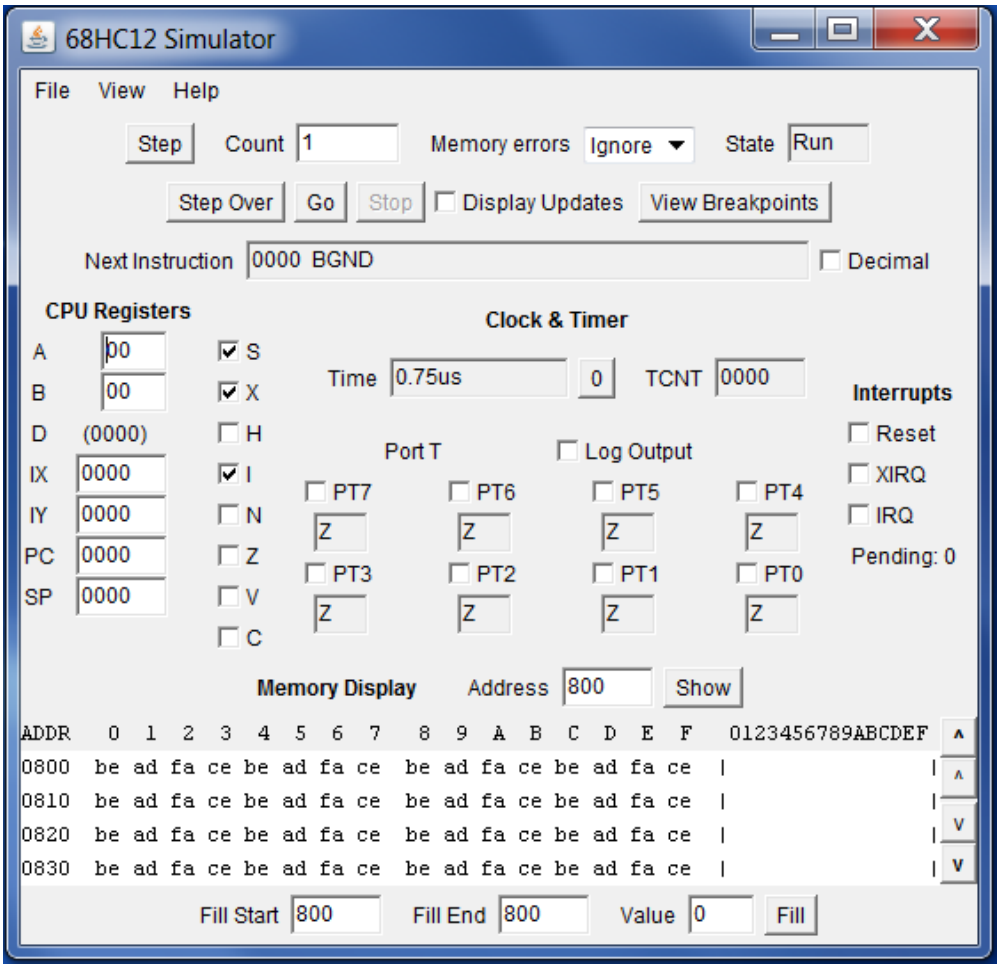

Figure 2. Simulator on startup

4.1 Loading and executing DBUG12

Click **Load** on the *File* pull-down menu. Browse to your *HCScode* folder and select *DBUG12.S19*. This will load the program DBUG12 into the simulator. This program is used to set up the interrupt vector table, set the stack pointer, initialize the clock simulator and set up the serial interface. It is not required to run simple programs, but it is a good idea to include it routinely with all user programs as it allows the simulator to be better behaved when unexpected conditions occur.

Click **Reset** on the *File* pull-down menu. This is equivalent to pulsing the reset pin on the microprocessor. Note that this sets the PC to point to the address of the first instruction of the DBUG12 code (FF02). Now click the **GO** button. This will run the DBUG12 code. When done, this program uses a BGND instruction to return control to the user. A pop-up window appears. Click **OK**. Note that the next instruction is a jump to location $4000, the address at which we normally place the start of user code.

4.2 Loading your program

Click **Load** on the *File* pull-down menu. Browse to your *HCScode* folder and select *simple.S19*. This will load the machine code of your program into the simulator.

Click **Code Viewer** on the *View* pull-down menu. A pop-up Code Viewer window appears. Enter 4000 into the address field and click Set Start. The window will display your program correctly loaded at memory location $4000, decompiled back into assembly code. Note that the simulator does not know about the labels used in your original code, so all addresses are now numerical. Close the *Code Viewer* window.

Now enter 5000 into the *Address* field of the Memory Display window and click **Show**. You can now see the values 23 and 5A that you loaded into memory locations *Adata* and *Bdata*. Note that *Rdata* (location $5002) is initially set to 00.

4.3 Executing your program

There are two ways you might execute your loaded program:

(a) Check that the PC still contains the value $F02A and that the *Next Instruction* field shows a jump to memory address $4000. If that is the case, simply clock **GO**. This will execute the jump to $4000 which will start your program.

(b) Alternatively, load $4000 into the program counter and click **GO**. This is the preferred technique as it can be used to re-start your code at any time, and does not rely on you just having run DBUG12.

Your program will run until it reaches the SWI instruction, at which point it will generate a HALT pop-up window. Click **OK**. Use the Memory Display window to examine the contents of *Rdata*. Does it contain the correct result? What about the contents of A, X and Y. Are they what you predicted back in Section 3.1? Unfortunately, you will see that the SWI handler in DBUG12 overwrites the data in the X and Y registers.

If you want to re-run the program and see the correct values of X and Y at the end of the program, replace the SWI instruction with a BGND instruction. You can do this by editing the original assembly program, re-assembling and reloading your machine code into the simulator. Or for a quick fix, you could change the machine code. Use the Memory Display window to change memory location $400F from $3F (the opcode for SWI) to $00 (the opcode for BGND). Set the program counter back to $4000 and rerun the program. Now when the program halts, you should see the correct values in the X and Y registers.

4.4 Single-stepping your program

The simulator provides the facility to single step your program, that is execute your program one instruction at a time, observing the contents of the registers and memory after each individual instruction execution. Load the program counter with $4000. Now click the **STEP** button. The simulator now executes just one instruction – the first instruction in your program (start: ldaa Adata). Note that accumulator A now contains $23. The PC is now set equal to the address of the second instruction $4003. The instruction at $4003 is shown in the *Next Instruction* window. Click **STEP** again. Note that the X register now contains the value $5001, the address of *Bdata*. The program counter has advanced to point to the third instruction. Continue clicking the STEP button, following the data changes to the registers and memory as each instruction is executed until you reach the end of the program.

The number of instructions executed at each step is specified by the number in the *Count* field. This number is one by default. The **Step Over** button operates similarly, except that when it encounters a subroutine call, it will execute a subroutine call as if it was just one

instruction and then proceed to the instruction immediately after the subroutine call, rather than executing the subroutine itself one instruction at a time.

4.5 Setting Breakpoints

Single stepping is very useful for debugging a program that is not behaving as expected. But it is not possible to single step a large program that requires thousands of instructions to be executed. Another mechanism one can use is to set a breakpoint. When a breakpoint is set at the address of an particular instruction, the program stops or breaks whenever the program counter is set to next execute that instruction. If we know that an error is occurring within a small group of instructions, we can set a breakpoint at the beginning of that group and then single step to see exactly what is going wrong.

Click on the **View Breakpoints** button. A *Breakpoints* pop-up window appears. Enter 400B into the small sub-window above the **Add** button. Now click the **Add** button. This adds 400B as a breakpoint in your program. $400B is the address of the "suba #10" instruction. Return to the main simulator window, set the PC to 4000 and click **GO**. This time the program runs until it's is ready to execute the instruction at $400B, It then halts with a pop-up window indicating that a breakpoint has occurred. You can now view the content of the registers right before the "suba" instruction is executed. To resume execution, click **OK** on the pop-up and then click **GO** again on the main simulation window. The program continues, starting with the breakpoint instruction until it reaches the normal end of the program.

4.6 Simulating microcontroller peripherals

The 68HC12 simulator is also capable of simulating some simple I/O features of the HCS12 microcontroller including the H, J and T ports, the Timer Counter module, one serial communication channel SCI0 and one A/D converter ATD0. The serial communications window can be enabled by clicking **SCI Viewer** on the *View* pull-down menu.

Note that the addresses of many of the I/O registers in the simulator are different from the addresses found in the microprocessor on the lab EVB board. The addresses of I/O registers in the simulator are given in the file *equates.asm.* To use the symbolic reference to any of these I/O registers in your program simply insert the line " #include equates.asm" at the beginning of your assembler program.

A detailed description of I/O operations in the simulator is beyond the scope of this document. Please refer to the simulator help and documentation pages for more information.