EE475 Lab #2 Fall 2005

Creating C Programs With the CodeWarrior IDE

In this lab you will use the Metrowerks CodeWarrior compiler to create, build, and run several simple C programs. CodeWarrior is a code development environment that supports a wide variety of processor *targets*, including the HC08 and HC12. We will be using it with the MC9S12C32 processors on the Axiom MCU Project Boards in the lab.

Logging On:

1. Wake up the computer (or turn on the power if it is off).
2. Begin by pressing Ctrl+Alt+Delete (Windows XP).
3. Enter your username and password, and select the proper domain.

Launching :

Start the development application:

> Programs | Metrowerks CodeWarrior | CW12 V3.1 | CodeWarrior IDE

Setting up the CodeWarrior software project:

1. From the CodeWarrior File menu, select the **New…** option. The **New Project** window should appear.

2. Fill in a project name (e.g., `Lab2xyz`, where `xyz` are your initials), and use the **Set…** button to locate a subdirectory you can read and write: `c:\eeclasses\ee475` is a good choice. Choose the **HC(S)12 New Project Wizard** and press **OK**.

3. Using the Wizard prompts:
    a. Select the **MC9S12C32** derivative
    b. Select C language support
    c. No PC-lint support
    d. No floating point support
    e. Small memory model
    f. Check the boxes to select **Metrowerks Full Chip Simulator** AND **P&E Hardware Debugging**

    and then press **Finish**.

The project framework has now been created.

In the left pane, open all the file browse folders by pressing the "+" boxes to see all the supporting files. Also, make sure that “P&E ICD” is the target selected in the drop-down box at the top of the folders pane.

Compiling a simple program:

Using the browse view, locate the `main.c` file in the **Sources** group. Double-click to launch the file in an editor window. This dummy `main()` routine just enables interrupts and then traps itself in an infinite loop. Keep in mind that embedded software generally does not "return" to a calling routine, so an infinite loop somewhere in the code is not unusual.

Now replace the `main()` routine with the following:

```
void main(void)
{
/* This is a test program.
 * The results are meaningless.
 */
  int j;
  volatile int val;

  val=10;

  for(j=0;j<6;++j){
    val=val+j;
  }
  for(;;); /* endless loop */
}
```

Note that `volatile int` is used to prevent the compiler from eliminating the "useless" code involving `val`. Also, note that the `_asm()` function can be used to insert an in-line assembly language instruction, such as `_asm("nop");`

Save the `main.c` file, then select **Make** from the Project menu (or press the "make" icon on the task bar).

Any compiler errors will appear in a pop-up window.

Ask for help if any of the preliminary steps did not work properly.

→ Once the program compiles correctly, try making some deliberate errors (missing semicolons, misspelled variable names, etc.) and see what the compiler errors look like.

Connecting to the SLK board

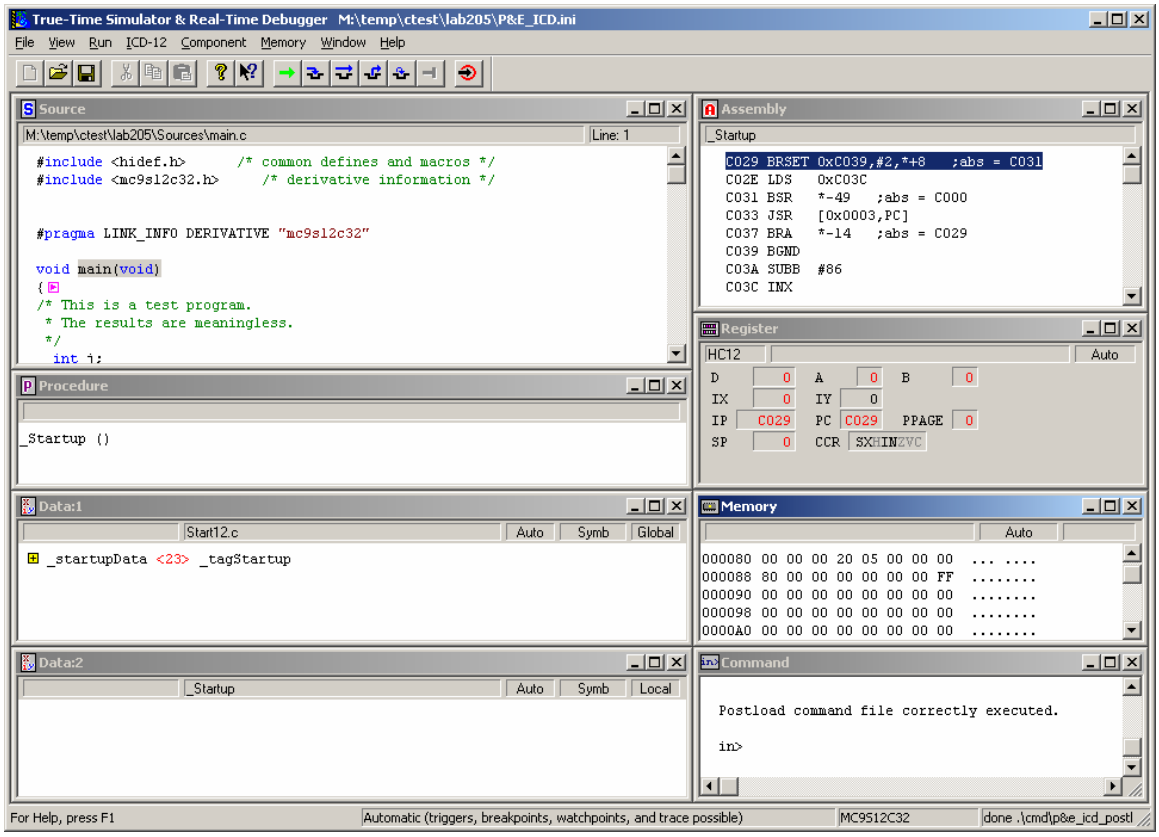
Carefully connect the USB cable between the computer and the shiny USB receptacle on the edge of the SLK board opposite to the microcontroller daughterboard. The USB power LED should turn on, and the VDD LED on the microcontroller daughterboard should light up.

Seek help if the board does not power up as expected.

Debugging the program

After the program compiles and links correctly, press the green **Debug** icon to launch the simulator/debugger application. The debugger app automatically opens a variety of windows to show the source code, disassembly, register contents, etc. You may get a warning message indicating that the non-volatile memory on the chip will be erased: this is OK. The LEDs on the daughterboard should turn off, except for the green power LED.

The Debugger app should show a bunch of windows, including the source code of your `main.c`, the current contents of the processor registers, a disassembly of the instructions, and some memory and data viewing windows (see below).



→ In the debugger, press the **green arrow** (⇨, start/continue) to run the program using the debugger. After a few seconds, press the red (⊣, stop) button. Observe the updated views in each window.

→ Now go to the **Source** window in the debugger and scroll until you can see the `val=10;` line. Set a *breakpoint* on this line by pointing with the mouse, doing a right-click to bring up the context menu, and selecting "**Set Breakpoint**". A red arrow should appear on that line in the **Source** window.

Select **Restart** in the **Run** menu, and observe that the simulator restarts the code and stops at the breakpoint line. Now use the **Single Step** button to go through the program one line at a time.

Things to keep in mind:

1. Each line of the C program usually executes several assembly language instructions. Observe the assembly instructions in the Assembly window. Note that you can use the **Assembly Step** button to execute the machine instructions one by one.

2. The variables are updated in the **Data** window after each step.

3. The compiler generates special startup and initialization code that is executed before turning control over to your `main()` routine. To see this, select `Reset` from the **ICD-12** menu, then start single stepping the program until you reach your `main()` routine. What is the startup and initialization code for?

Running the software simulator

In addition to the hardware debugging connection, the CodeWarrior debugger app can also perform a *simulation* of the chip (no hardware required). Setup the following:

1. Close the debug app (True-Time) and return to the CodeWarrior project.

2. In the drop-down box at the top of the project file pane, select **Simulator** instead of P&E ICD.

3. Recompile and link your program.

4. Now re-launch the debugger app (green bug arrow).

Try running the code, setting breakpoints, etc., using the simulator.

→ Demonstrate your use of the simulator/debugger for the instructor. Can you think of some reasons that you might choose to use the software simulator instead of the hardware debugger?

Problems to do in the lab:

For the following lab problems you need to demonstrate a working program to get credit for it. Have the instructor initial the verification sheet for each working assignment that is completed.

Problem #1: (BE SURE TO SWITCH BACK TO USING THE P&E ICD HARDWARE DEBUGGING)

(a) → Add `sizeof()` expressions to your `main()` program so that you can determine the byte sizes used for the types examined last week in Lab #1:

```
char                unsigned char
signed char         short
short int           int
long int            float
double
```

One way to do this is to assign the result to a variable, then single-step the program to see each result, e.g.,

```
val=sizeof(char);
val=sizeof(unsigned char);
val=sizeof(short);
…
```

(b) → Determine if the HC12 storage is *little endian* or *big endian* by saving an integer value in your program and then observing the storage using the **Memory** window.

(c) → Next, observe what happens if you remove or comment-out the endless loop at the end of your program (`for(;;);`). Do a single-step and see what the **Assembly** window shows.

(d) → Finally, observe what happens if you change the declaration

```
volatile int val;
```

to just

```
int val;
```

Recompile the program and single-step it through the code. What did the compiler do differently this time? Explain.

→ Demonstrate your program and results for the instructor.

Before moving on to Problem #2, keep a copy of the Problem #1 `main.c` *file using the following procedure:*

First, re-edit the file so that it has the `volatile int val` *declaration and the infinite loop at the end.*

In the CodeWarrior window, save the `main.c` *file under the name (*`Save As…`*)* `main1.c`*, then right-click on the file name in the project browser and remove it from the Sources folder. Don't be alarmed by the warning message: the file itself will not be deleted from the disk folder, only from the list of project files.*

Next, add the `main.c` *file back into the Sources group, open it in the editor, and delete the statements within the* `main()` *routine. Re-save the main.c file, then go ahead with the next problem.*

Problem #2: Create a C program that blinks the two LEDs on the microcontroller board. Write the code as a continuous loop—you might want to include a delay (do-nothing software loop) to make each blink last longer.

To implement this program you will need to realize that:

(a) The `mc9s12c32.h` header file includes C definitions of the various processor I/O ports (`PORTA`, `PORTP`, etc.) and the individual register bits.

(b) The bits in each register are numbered from zero (least significant bit) through seven (most significant bit).

(c) The Bit 0 (zero) of Port A (`PORTA`, 0x0000) is connected to the cathode of LED1. Set the Port A data direction register (`DDRA`, 0x0002) so that bit 0 is an *output*.

(d) The bit 4 of Port B (`PORTB`, 0x0001) is connected to the cathode of LED2. Set the Port B data direction register (`DDRB`, 0x0003) so that bit 4 is an *output*.

(e) Locate the system copy of `mc9s12c32.h` in order to see all the defined values. The include files should be somewhere like:

```
C:\Program Files\Metrowerks\CodeWarrior CW12_V3.1\lib\HC12c\include
```

Put statements in the `main()` routine in order to implement the flashing LEDs. A partial skeleton is shown below:

```
void main(void)
{
/* Set the bits of Port A and Port B to be outputs using
 * the data direction registers DDRA and DDRB
 */
  DDRA = 0x01;
  DDRB = 0x10;

for(;;){
  PORTA=0x00;
  PORTB=0x00;
…delay somehow…
  PORTA=0x01;
  PORTB=0x10;
…delay somehow…
}/* end of endless for loop */
} /* end of main() */
```

It is advisable to start simple: just make a program that turns on LED1 by writing to Port A. Then do something to turn on LED2. Incrementally add complexity until you have both LEDs flashing.

The code fragment in Problem #2 is rather casual about the register usage: all of the bits in Port A and Port B are affected each time there is an assignment statement like `PORTA=0x01;` If those other bits were connected to other hardware features then it wouldn't be a good idea to change them.

Instead, consider two options for setting and clearing *single* bits in a register

1. Use a sequence of read-modify-write commands, e.g., `PORTA = (PORTA | 0x01 )` to set bit 0 of Port A, leaving the other bits unchanged, and `PORTB = (PORTB & 0xEF)` to clear only bit 4 of Port B.

2. Use the defined bits from the `mc9s12c32.h` file, e.g., `PORTA_BIT0 = 1` and `PORTB_BIT4 = 0`.

→ Modify your program to use method 1 and observe what assembly language instructions are created by the compiler. Comment in your report.

→ Finally, modify your program to use method 2 and again observe what assembly language instructions are created. Again, comment in your report.

→ Demonstrate and explain your flashing LED program for the instructor.

BE SURE TO KEEP COPIES OF YOUR CODE AND INPUT/OUTPUT EXAMPLES. Even though you may work in pairs in the lab, each student needs to write his or her own individual report.

Lab Report: Due at THE START OF THE LAB PERIOD NEXT WEEK.

The lab report is to be written up in the Memo format. Each student should submit a separate lab report. For each problem, write a short description of what you did to solve the problem. Include **commented** C code *excerpts* for each problem and include them within the memo.

Instructor Verification Sheet

EE475 Lab #2

Fall 2005

Student Name: _____

	Instructor Signature	**Date**
Editor, compiler, simulator skills demonstration.		
Problem #1 runs on the HC-12 hardware.		
Problem #2 flashes LEDs.		

Note: This verification sheet must be signed by the instructor and submitted with the lab report to get any credit for the lab.