EE475 Lab #2 Fall 2004

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 MC68HC912BC32 processors on the Axiom CME-12BC32 single-board computers.

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 MC68HC912**BC**32 derivative
   b. Select C language support
   c. No PC-lint support
   d. No floating point support
   e. Small memory model
   f. Metrowerks Full Chip Simulator

   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.

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;
  }
  _asm("swi");
}
```

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. In this case we are inserting a software interrupt (`swi`), and this will cause the program to turn control back to CodeWarrior.

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.

Simulating/debugging the program:

Fix the program until it compiles correctly, then press the green **Debug** button to launch the simulator/debugger application. The debugger app automatically opens a variety of windows to show the source code, disassembly, register contents, etc.

→ In the debugger, press the **green arrow** (start/continue) to run the program using the software simulator. The simulator will stop when it hits the `"swi"` (software interrupt) instruction. 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.
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 `Load…` from the **Simulator** menu, find the file `Simulator.abs` in the `bin` folder, load it, then start single stepping the program until you reach your `main()` routine.

→ Demonstrate your use of the simulator/debugger for the instructor.

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 sign the verification sheet for each working assignment that is completed.

Problem #1: In addition to the software simulator, the CodeWarrior debugger app will monitor the HC12 development boards in the lab. Setup the following:

1. Make sure the Axiom board is connected with a serial cable, and the cable switch box is set properly to use that cable.
2. Make sure the Axiom board is powered up. Press Reset on the board.
3. In the debugger app, go to the **Component** menu and choose `Set Target…` In the target selection window, select the `HC12` processor and the `D-Bug12 Target Interface` and press OK. The software should locate the hardware via the serial port and load the code into the hardware processor's memory. Make sure the CPU derivative type is set to **MC68HC912B32**.

Try running the code, setting breakpoints, etc., using the serial connection.

Once you are familiar with how the hardware debugger behaves—essentially just like the software simulator—go back and do the following:

1. Get a copy of the `dbug12.h` file from the EE475 course web site (Notes page) and place it in the Sources folder of your project (e.g., `\eeclasses\ee475\Lab2xxx\Sources`). `dbug12.h` contains the definitions and declarations needed for your C program to use the Dbug12 ROM monitor functions.

2. In the CodeWarrior project window, right-click on the **Sources** group and select `Add Files…` Browse to find the `dbug12.h` file, and add it to the project.

3. Edit `main.c` to add `#include "dbug12.h"` on the line below the other `#include` files.

4. Now just above the `_asm("swi");` line, insert the statement:

   ```
   val = DBug12FNP->SetUserVector(RAMVectAddr,(Address) 0 );
   ```

   This complicated-looking statement uses the Dbug12 monitor routine to report where the interrupt vector table is located in memory.

Compile the code, fix any errors, then re-launch the debugger and run your program.

→ Set the **Data** window in the debugger to display the variable `val` in hex format, and show the instructor where the vector table can be found.

→ For your memo report, include an explanation of what the `SetUserVector` routine does: refer to the Dbug12 application notes on the EE475 course web site. We will be using this routine to install interrupt service routines in future labs.

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

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. Save the file, then go ahead with the next problem.

Problem #2: Create a C program that sequentially blinks each of the 8 LEDs on the I/O board, one at a time, in 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 `MC68HC912BC32.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) Port P (`PORTP`, 0x0056) is connected to the LEDs through a 74LS373 tri-state latch. Set the Port P data direction register (`DDRP`, 0x0057) so that all eight bits are outputs. To enable the '373 latch, set bit 5 of the CAN port data direction register (`DDRCAN`, 0x013F) to '1', and set bit 5 of the CAN port (`PORTCAN`, 0x013E) to '1'.

(d) The Port P bits are connected to the cathodes of the diodes (active low).

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 P to be outputs using
 * the data direction register DDRP
 */
  DDRP = 0xFF;

/* Set bit 5 of the CAN port to be an output using DDRCAN */
  DDRCAN5 = 1;

/* Set bit 5 of the CAN port to be '1' to enable the '373 latch
*/
  PCAN5 = 1;

…etc…
}
```

It is advisable to start simple: just make a program that turns on a pattern of LEDs by writing to Port P. Incrementally add complexity until you have the LEDs flashing one by one continuously.

→ Demonstrate 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 2004

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.