SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

ET0104 Embedded Computer Systems Laboratory

Laboratory 2 Further features of C compilers

Objectives

- To explore the debugging features of Visual C++.
- To use the features of Visual C++ that encourage modular programming

Introduction

In this session, we will use some of the more powerful features of Visual Studio and Embedded Toolsuite VC/ETS. Specifically, we shall learn :

i) The difference between Debugging and Production modes.
ii) How to set breakpoints and examine variables.
iii) How to set watches.

Assignment

We will revisit the timer program used in the previous lab. Create a new project ***lab2*** in the subdirectory ***ECSLAB/lab2***. Bring in the program ***lab2.c*** and fill in the blanks as before.

Frequently encountered problems in program development

We look at some frequently encountered problems:

i) Why did the program not execute this part?
ii) Why did the value change/ not change?

The first question comes about after the program makes a decision, for example an *if*, *switch* or *while* statement. For this, we want to stop the program just after the decision. We also want to examine the test variable.

In the second case, we normally call a function to perform an operation - perhaps to output to a device. Again, we want to stop just before the function. Then we want to execute just one step after the procedure, or step *into* the procedure to see what went wrong by viewing the offending variable(s) as we execute the function one step at a time.

Breakpoints, stepping and watches

The point at which a program stops is called a “breakpoint". A breakpoint that can stop a program as it is executing at full speed *without* modifying the program itself is a “hardware breakpoint”. As you may guess, many debuggers insert a special instruction into a program to make it stop. This is a “software breakpoint”. In both cases, we can have *conditional* breakpoints, that is, stopping only when a certain condition comes up.

When a program executes one line at a time, it is a "step". To observe the values of a variable as we step is to "watch" it.

Typical embedded systems do not normally provide such debugging features. To perform such tasks in these systems, special hardware is needed.

However the 386 architecture has these advanced features. This allows us to debug programs easily. VC/ETS allow us to control the SBC over the parallel port link.

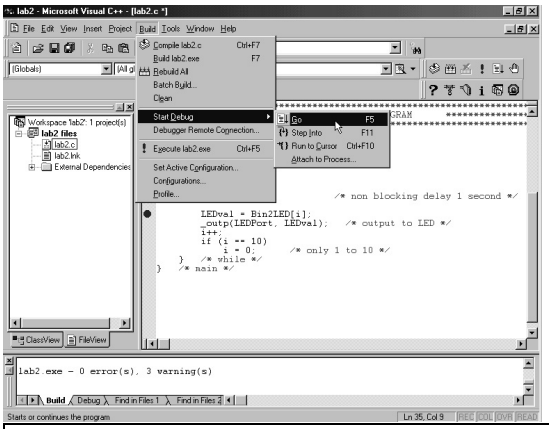
Looking at the LED program, let's say we want to see how the program converts the variable i to its 7 segment form. Also, we want to see the value being output to the port.

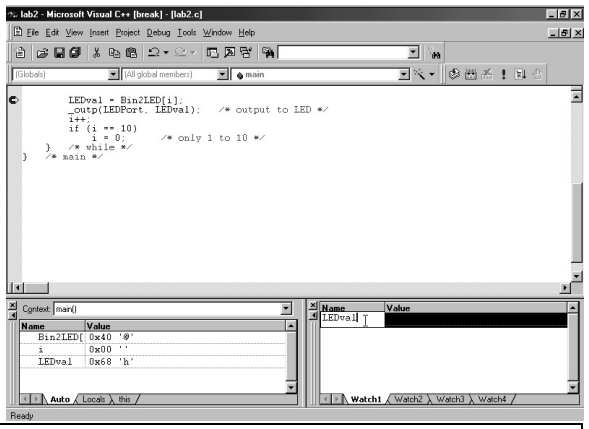
```
while (TRUE)
{
        Sleep(______);              /* non blocking delay 1 second */
        LEDval = Bin2LED[i];
        _outp(LEDPort, LEDval); /* output to LED */
        i++;
        if (i == 10)
                i = 0;              /* only 1 to 10 */
}
```

We want to *break* at the statement: `LEDval = Bin2LED[i];`
After that, we want to *step* through and *watch* the values *i* and *LEDval*.

1. Position the cursor at the statement: `LEDval = Bin2LED[i];`

2. Press ***F9***, a red circle will appear next to the statement.



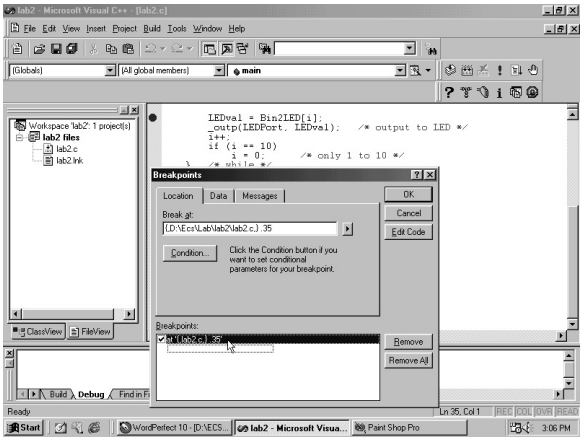


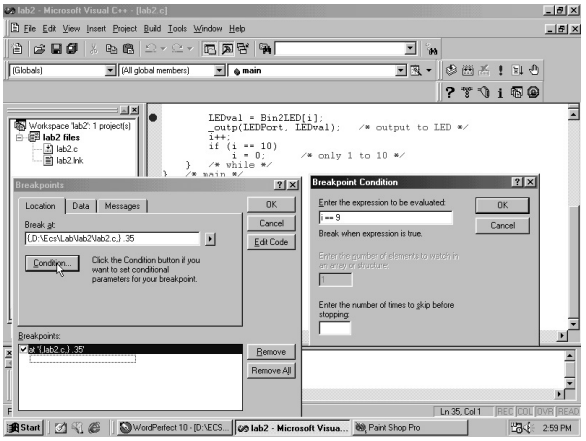
3. In the previous lab, we *executed* the program straightaway. To debug, we have to: i) Enable the debug option, do: ***Build*** >***Set Active Configuration*** click on ***Win32 Debug*** ii) To *debug* the program, ***Build*** >***Start Debug*** > ***Go (F5)***	4. In the lower left window pane is a list of *all* the variables in the program. It is quite small here. To select only certain variables to "watch", double click an edit box in the *watch1* tab. We want to watch *LEDval* and *i*. Type in these names OR copy and paste from the variable display. You can Right-click on the value and select ***Hexadecimal*** to view in hex.

5. Now, step through the program : ***Debug*** > ***Step Over (F10)***. Note the values of *LEDval* and *i* change as you step through the program.

6. Now we want to break only after a certain number of events happen. This is useful especially if something goes wrong only after several thousand runs! This is done by setting a *conditional* breakpoint.

You'll need to stop debugging: ***Debug*** > ***Stop Debugging (Shift-F5)***. Note that it is no use to type *Ctl-Break* in this case as the program was never running!





7. ***E dit >Breakpoint (Alt-F9)*** Click on the breakpoint (not the check box). The Condition button will become active. Click on it (or press ***Alt-C***) to bring up the condition dialog box.	8. Type in ***i == 9*** to signify that we want to break only when i equals 9. You will see something similar to: `at '{c:\ecs\lab\lab1\lab1.c,}.26' when 'i==9'` at the breakpoint. OK to continue.

9. Start debugging again and note the value of *i* when it does break.

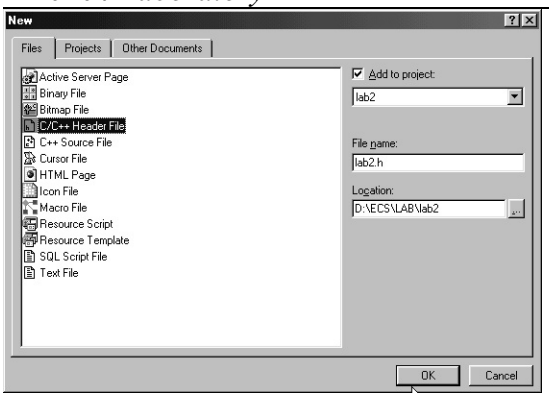
10. You may wish to try out other ways of enabling the conditional break, or the other break options.

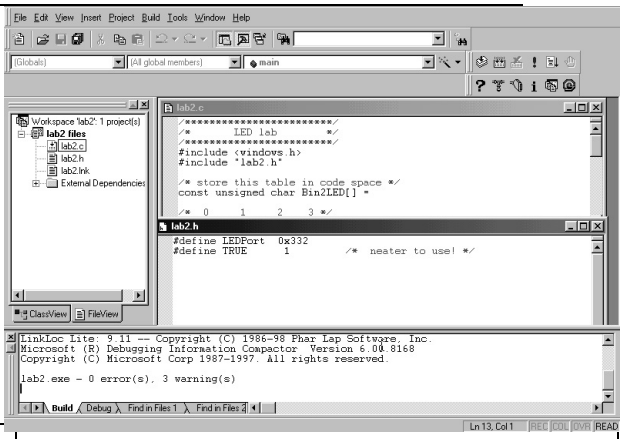
Modular programming features

We have mentioned that the project management features of VC allow it to handle large projects. We will explore this feature now. The degree to which you wish to modularize a program is largely a matter of personal preference, but lab2.C is definitely too small for this! However, the introduction here will help later in the assignment.

Header files

Definitions, constants may distract. If there are a lot of them, it is better to put them into a separate header file. We will create a lab2.H file.





1. ***Project*** >***Add to project*** > ***New***

In the tabbed dialog, select ***Files*** > ***C/C++ Header file***.

Type in ***lab2.h*** in the File Name edit box.

OK the button. You should be editing the document *lab2.h*, which is empty now.

We want to transfer the *#define* statements here.

2. Double click on *lab2.c* in the Workspace view.

Cut the lines:

```
#define LEDPort  0x332
#define TRUE      1
```

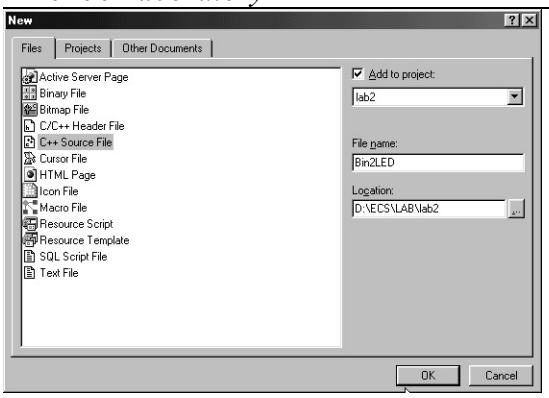
Paste them into lab2.h:

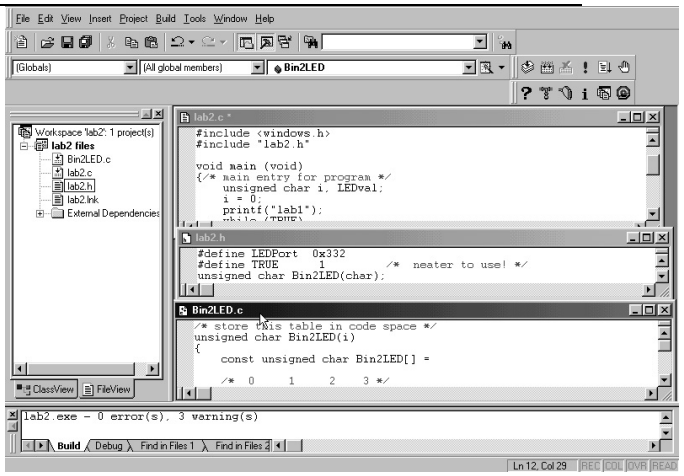
Type in ***#include "lab2.h"*** into the remaining space in lab2.C.

You may wish to compile and test your program.

Functions

In order to make a program easier to read, it is preferable to break it up into smaller, logical parts. This is so we can concentrate on the main task in the program and perhaps subdivide out the other parts for others to do. Let's put the conversion from binary to seven segment code into a routine.





3. ***Project >Add to project > New*** In the tabbed dialog, click ***Files*** > ***C++ Source file***. Type in ***Bin2LED.c*** in the File Name edit box. ***OK*** the button. You should be editing the document *Bin2LED.C*, which is empty now. We want to transfer the *#define* statements here.	4. Double click on *lab2.c* in the Workspace view. ***Cut*** the entire Bin2LED array: `const unsigned char Bin2LED[]= ...` ***Paste*** the lines into Bin2LED.c. Modify the function as in the appendix. In lab2.c, modify the line: `LEDval = Bin2LED[i];` to ***`LEDval = Bin2LED(i);`*** It is also good practice to put into lab2.h: ***`unsigned char Bin2LED(char);`***

Thinking question

In step 4, you changed the *array reference* to a *subroutine* by just changing the square brackets [] to curved brackets (). Explain why this is so.

------ End of Lab --------

Appendix

Template for Bin2LED.C:

```
unsigned char Bin2LED(i)
{
      const unsigned char Bin2LED[] =

      /*  0     1     2    3 */
      {0x__, 0x__, 0x__,0x__,
      /*  4     5     6    7*/
      0x__, 0x__, 0x__,0x__,
      /*  8     9     A    B*/
      0x__, 0x__, 0x__,0x__,
      /*  C     D     E    F*/
      0x__, 0x__, 0x__,0x__
      };
      return(Bin2LED[i]);
}
```

Summary of keystrokes available in a debugging session

Alt+F10	Applies code changes made to C/C++ source files while debugging
Ctrl+F9	Enables or disables a breakpoint
F5	Starts or continues the program
Alt+F11	Switches the memory window to the next display format
Alt+Shf+F11	Switches the memory window to the previous display format
Shift+F9	Performs immediate evaluation of variables and expressions
Ctl+Shf+F9	Removes all breakpoints
Ctl+Shf+F5	Restarts the program
Ctl+F10	Runs the program to the line containing the cursor
Ctl+Shf+F10	Sets the instruction pointer to the line containing the cursor
Alt+Num *	Displays the source line for the instruction pointer
F11	Steps into the next statement
Shift+F11	Steps out of the current function
F10	Steps over the next statement
Shift+F5	Stops debugging the program
F9	Inserts or removes a breakpoint
Ctrl+F11	Switches between source and disassembly view for this instruction