C/C++ Debug With Popular Tool Chains

Using Run Control Hardware to Facilitate Software Development: From Coding to Tracing With the Intel® 440BX Scalable Performance Board Development Kit

An Application Note

www.arium.com

Overview

This application note takes the reader through a sample software development environment with the Intel® 440BX Scalable Performance Board Development Kit, illustrating the use of hardware run control in the process. Examples using three different tool chains are given. Sample code and Makefiles are shown, and links to useful utility programs are provided. The sample code and support files are contained in an archive file (440BX Eval Board.zip) that can be found on the American Arium Website at www.arium.com. The files decompress to three folders, one for each of the tool chains that are covered in these exercises.

In order to simplify the process, a floppy disk is available that can be used to boot the Intel 440BX board that is included with the development kit. A copy of the disk can be obtained by e-mailing support@arium.com.

The Advantages of Hardware Run Control

Modern Integrated Development Environments (IDE) such as Wind River® Tornado® II contain many valuable tools that aid in debugging operating systems such as Wind River VxWorks® and in optimizing task control. For application code, the Board Support Package (BSP), the BIOS, OS kernels, and the device drivers, however, something more is needed.

Especially at the lower levels of development (BSP and BIOS), it is important to have a tool that can continue to provide a connection to the processor when the processor is not behaving properly. Most software debuggers lose their usefulness precisely when most firmware engineers need it the most. It is at times like these that you need to do crash-proof debugging, and that is the exclusive domain of hardware run-control products.

In addition, real-time dependent code can be very difficult to debug. When Interrupt Service Routines (ISRs) need to be serviced in determined times, system debug can get very complicated. In a heavily interrupt driven, real-time environment, code visibility and control is invaluable. Hardware-assisted debug tools are offer a powerful solution.

The In-Target Probe (ITP), which connects to the target through the JTAG port, gives you the ability to set breakpoints on Reset, SMM Entry, and SMM exit. These “Emulator” breakpoints supplement the Debug Register (Processor) breakpoints and Software (Trap) breakpoints that are available. An In-Circuit Emulator (ICE) can do anything the ITP can do, and adds trace and Bus Analyzer breakpoint capability. Trace adds immensely to the visibility of the debug environment, while Bus Analyzer breakpoints are a tremendous enhancement to controlling what portion of the code is to be viewed.

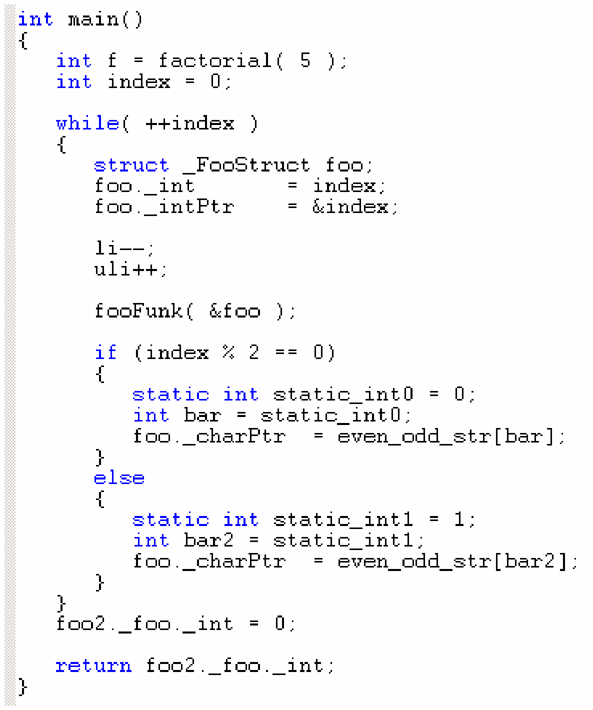
Trace provides a means of seeing how the processor arrived at its current location. It allows you to simply set a breakpoint, run to it, and then analyze the trace buffer, rather than single stepping through large amounts of code. Bus Analyzer breakpoints give you precise control over where the processor stops and what the trace buffer collects. These breakpoints can 1) break on more types of cycles than other types of breakpoints (e.g., Data Read, I/O Read, I/O Write, 2) break on specific data or I/O values (other types of breakpoints cannot), and 3) break on a specific number of occurrences of a breakpoint or on Boolean combinations of multiple breakpoints.

This last capability, to define events, can be used in conjunction with the trace tool to define trace events. This allows you to trace specific functions or routines without stopping, to trace only certain bus-cycles, or to stop tracing after a specified number of iterations of a specified event. The ICE allows the creative programmer to find bugs in a fraction of the time required by other tools.

Wind River Tornado II Tool Chain

Figure 1 below shows a sample of the source code that will be used for all of the examples in this paper. This code simply performs loops that move numbers around in an interesting manner. The files relevant to the Tornado tool-chain will decompress to a folder named VxWorks.

There are several source files (note especially flat.c, csample.c and csample2.c), a header file (see csample.h), numerous library files, support files, and a Tornado project file (flat.wpj). The relationships of these files to one another are most easily understood within Tornado II by opening the Tornado Project File flat.wpj.



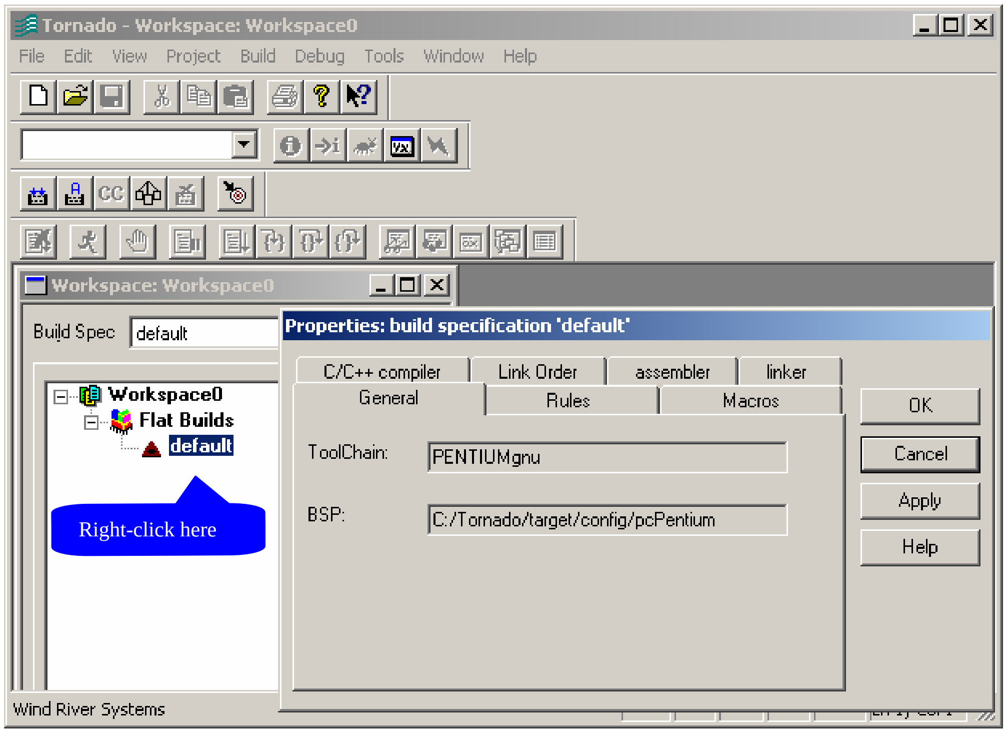
```
int main()
{
   int f = factorial( 5 );
   int index = 0;

   while( ++index )
   {
      struct _FooStruct foo;
      foo._int       = index;
      foo._intPtr    = &index;

      li--;
      uli++;

      fooFunk( &foo );

      if (index % 2 == 0)
      {
         static int static_int0 = 0;
         int bar = static_int0;
         foo._charPtr   = even_odd_str[bar];
      }
      else
      {
         static int static_int1 = 1;
         int bar2 = static_int1;
         foo._charPtr   = even_odd_str[bar2];
      }
   }
   foo2._foo._int = 0;

   return foo2._foo._int;
}
```

Figure 1

Building the Executable

At this early stage of the project, you should stick to “fixed” code. You should build relocatable code only for the portion of the project where you plan to use Tornado tools. When building your aout file, use the Build Specifications properties sheet to examine and change the default or other user-defined build specifications for the project. To open the properties sheet, right-click on the name of the build in the Builds view in the Workspace window (Build tab), and select Properties from the context menu. (See Figure 2 below). Select the labeled tabs at the top of the properties sheet to examine and modify settings in that category.

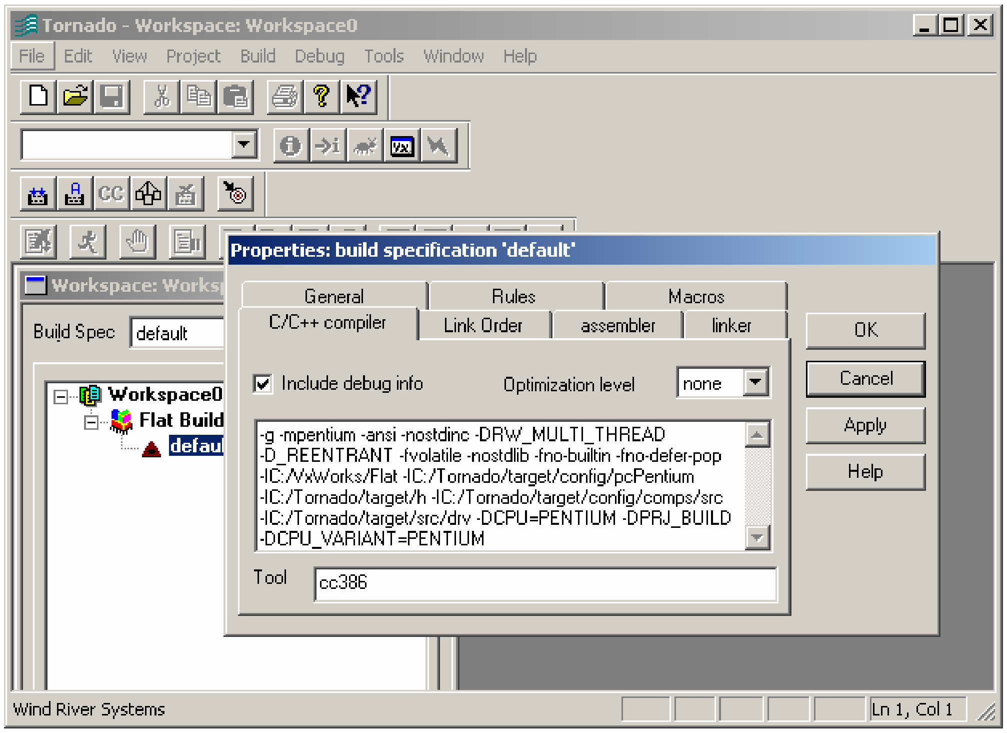
Figure 2 shows the Build Properties window with the General tab selected. Note that the tool chain chosen is PENTIUMgnu and that the BSP used is pcPENTIUM. Figure 3 shows recommended settings for the compiler. Note especially the selections for processor and debug information. The settings for the assembler should be identical to the compiler settings.


Properties: build specification 'default'

ToolChain: PENTIUMgnu

BSP: C:/Tornado/target/config/pcPentium

Figure 2


Properties: build specification 'default'

☑ Include debug info Optimization level: none

```
-g -mpentium -ansi -nostdinc -DRW_MULTI_THREAD
-D_REENTRANT -fvolatile -nostdlib -fno-builtin -fno-defer-pop
-IC:/VxWorks/Flat -IC:/Tornado/target/config/pcPentium
-IC:/Tornado/target/h -IC:/Tornado/target/config/comps/src
-IC:/Tornado/target/src/drv -DCPU=PENTIUM -DPRJ_BUILD
-DCPU_VARIANT=PENTIUM
```

Tool: cc386

Figure 3

The remaining settings are left to your discretion. It should be noted that for the purposes of this exercise, it is not necessary to rebuild the project. You can run the example files with the Intel® 440BX Eval Board without modification.

Booting the Target

To boot the Intel 440 BX Scalable Performance Board with the included VxWorks BSP, you must first set up the evaluation board according to chapter 2, “Getting Started,” of *the Intel® 440 BX Scalable Performance Board Development Kit Manual*.

The easiest way to boot this target with VxWorks is to use the floppy drive. While chapter 2 of the manual states that it is necessary to alter the BIOS to enable the floppy drive, most boards are shipped with the BIOS set up to work properly with the included floppy drive. If this isn’t the case for your board, see Chapter 5, “BIOS Quick Reference,” and go through the process of enabling this drive.

Note: The BIOS boots very quickly. If you are using a cold cathode ray tube (CRT) screen, you may have to press <DEL> before the CRT is warm enough to display output. Otherwise, you won’t be able to enter the SETUP environment.

Loading the Symbols for VxWorks

Now that you have an aout file, you are ready to load the code and begin the debugging process. SourcePoint supports the aout file and stabs symbol formats used in Tornado II, providing full symbolic debugging of your embedded VxWorks application. Support includes source code level debugging, symbolic procedures, variables, and type information. In addition to complete C code support, SourcePoint adds C++ symbol support.
Ensure that the VxWorks boot disk is in the floppy drive, and that the floppy drive is properly connected to the Intel 440BX target. Next, make sure that the ICE is properly connected to the target, and that both are powered up.

Now it is time to open SourcePoint. Once SourcePoint is open, loading symbols is as easy as loading the program object that you created with Tornado II (remember, turn symbols on with the –g compiler option).
Before you endeavor to load the symbols, you should first familiarize yourself with the SourcePoint interface by reading the flyover text for the various buttons. The buttons are the easiest way for a beginner to control the emulator. Find the “reset” button on the SourcePoint interface and click it with the mouse pointer. Now open a Code window (find the button with the flyover text “Code Window” and click it) and observe that the IP is at the reset vector.

Click the “Go” button and wait until you hear a pause in the activity on the floppy drive. Then click the “Stop” button, and with the target halted, load the VxWorks program object created in Tornado II. Depressing the “Program Load” icon button does this most easily. The result is shown in Figure 4. In the directory where you installed the exercises, find the file …\VxWorks\Flat\aout\vxworks.

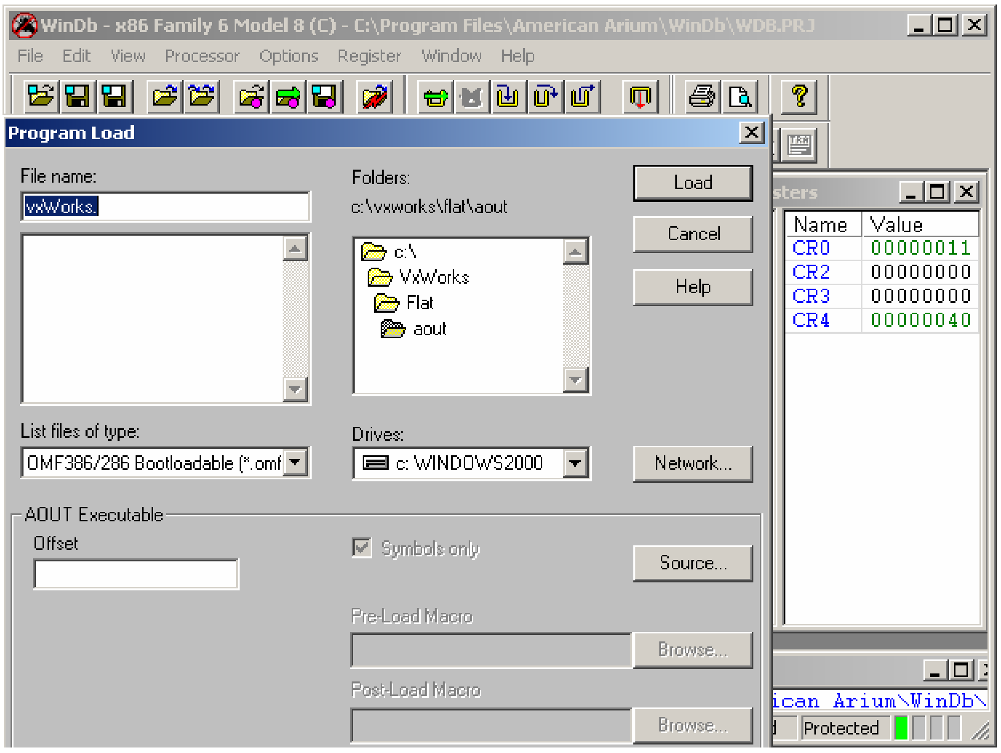
Immediately on loading the program object that contains the debugging symbols, the Symbols window (click the “Symbols Window” icon button) will show your program’s symbols in a tree structure.

> Important: Before loading the symbols, you must be certain that the target is in Protected Mode. Otherwise, the symbol addresses will not resolve correctly. You can ensure you are in Protected Mode when the target is stopped by either the status indicator on the lower right of the SourcePoint application window displaying “Protected,” or by checking the LSB of the CR0 register (‘1’ is Protected Mode “on”).

We do not recommend loading your program onto your target via SourcePoint, as the interface is sometimes slower than that offered by VxWorks. We always suggest that the program object be loaded into SourcePoint with the “Symbols Only” option selected.

Source Code Path

By default, SourcePoint looks for requested source code in the same directory from which your program object was loaded. Most commonly, your source code will be elsewhere. When SourcePoint needs a source code file as it is analyzing debug symbols, it will request it. These requests can become tedious when you have source code spread over several directories. Alternately, you can specify a list of paths to search at load time. This list is reached from the Program Load dialog box by selecting the “Source” button. Enter the list of paths to be searched, separated by semi-colons. This list is saved so that next time you load the same program object, the same directory list can be used.


WinDb - x86 Family 6 Model 8 (C) - C:\Program Files\American Arium\WinDb\WDB.PRJ

Program Load

File name: vxWorks.

Folders: c:\vxworks\flat\aout

c:\
VxWorks
Flat
aout

List files of type: OMF386/286 Bootloadable (*.omf

Drives: c: WINDOWS2000

AOUT Executable

Offset:

Symbols only ✓

Pre-Load Macro

Post-Load Macro

Name	Value
CR0	00000011
CR2	00000000
CR3	00000000
CR4	00000040

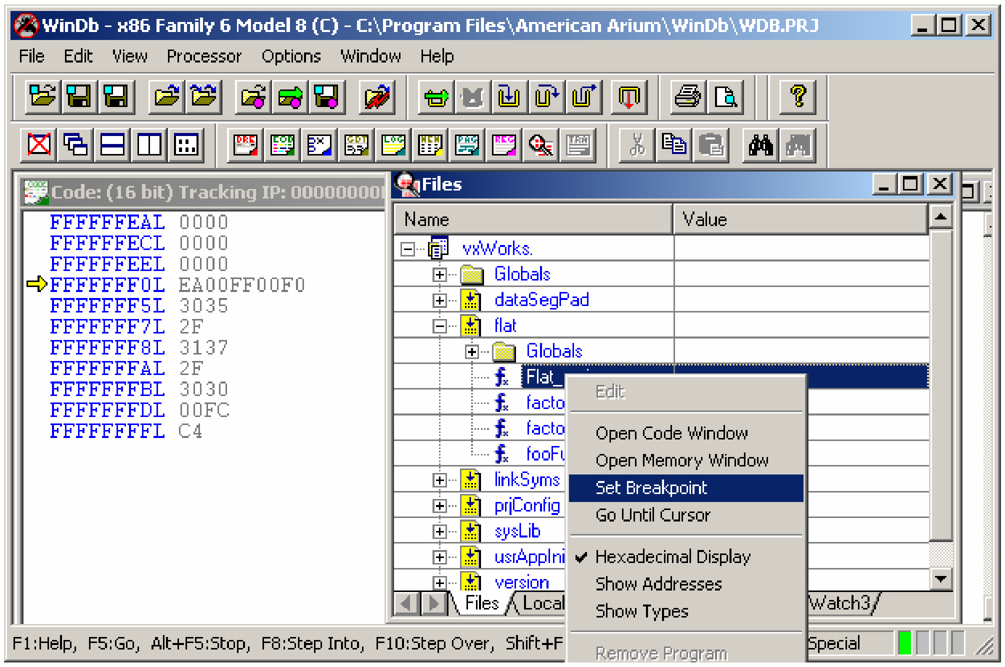
Figure 4

Setting breakpoints

Once your symbols are loaded, you may want to set a breakpoint on a procedure. The procedures are listed in the Symbols window under their respective modules (modules roughly correspond to files). The best place to start is at the beginning of the source code. To find this, click to expand the files under VxWorks, then click to expand the files under Flat. This is where you will find the source code for this exercise.

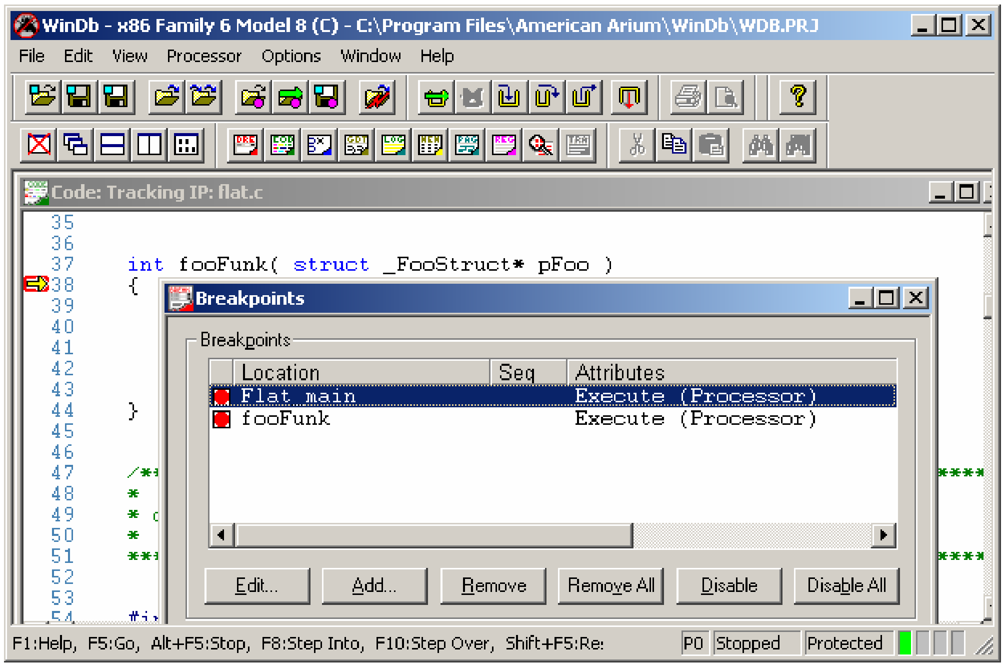
Right click on the Flat_main. Figure 5 below shows the context sensitive menu that pops up when you do this.

With the breakpoint set, continue execution of the code and wait for the breakpoint to be hit. If you have already continued execution before reading this far, you may be past the point where this breakpoint will be encountered. If this is the case, simply set a breakpoint on fooFunk the same way. While you are learning how to use the ICE, you may want to open the breakpoint window to examine the breakpoint that you have just set. Figure 6 shows the result of setting and hitting this breakpoint and also shows the contents of the Breakpoint window.



```
FFFFFFEAL 0000
FFFFFFECL 0000
FFFFFFEEL 0000
FFFFFFF0L EA00FF00F0
FFFFFFF5L 3035
FFFFFFF7L 2F
FFFFFFF8L 3137
FFFFFFFAL 2F
FFFFFFFBL 3030
FFFFFFFDL 00FC
FFFFFFFFL C4
```

Figure 5


WinDb - x86 Family 6 Model 8 (C) - C:\Program Files\American Arium\WinDb\WDB.PRJ
File Edit View Processor Options Window Help
Code: Tracking IP: flat.c
Breakpoints
Edit...
Add...
Remove
Remove All
Disable
Disable All
F1:Help, F5:Go, Alt+F5:Stop, F8:Step Into, F10:Step Over, Shift+F5:Re:
P0
Stopped
Protected

```
int fooFunk( struct _FooStruct* pFoo )
{

}
```

Location	Seq	Attributes
Flat_main		Execute (Processor)
fooFunk		Execute (Processor)

Figure 6

Another method of setting a breakpoint at the beginning of a function is to type the symbol for the function in the Address text box of the Code window (usually found in the lower left corner of the Code window if you haven't customized the window). If SourcePoint recognizes the symbol, the symbol will be replaced by the virtual address immediately on hitting the enter key. This will show the procedure in the Code window and will allow you to place the cursor at the start of the function. The breakpoint can then be set by pressing the F9 key.

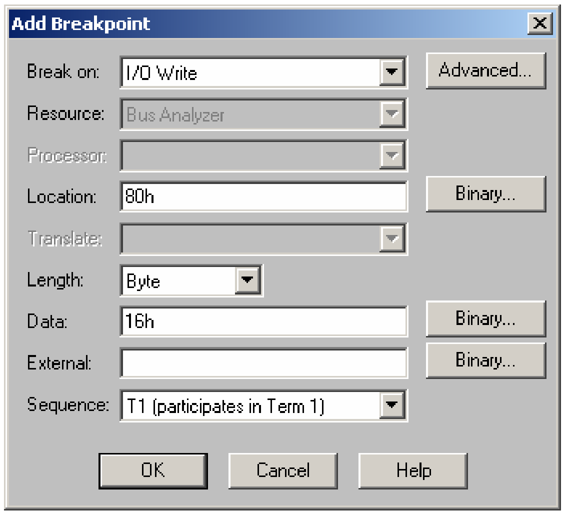
Alternatively, you can open the Breakpoint window and select the Add button. This will open the Add Breakpoint dialog box. In the Location section of this dialog, type the symbol for the function. Unlike the Address text box of the Code window, the symbol will not be replaced by the virtual address.

Using the TRC-20

As previously mentioned, the principal advantages of a full emulator (ICE) over a JTAG run-control device (ITP) are: the ability to set Bus Analyzer breakpoints, the ability to capture a trace of either the bus or the assembly instructions or both, and the ability to break or trace based on complex events.

An excellent application note that covers the use of these features can be found at http://www.arium.com/support/pdf/Measurement.PDF. This application note covers: 1) How to determine the time between and duration of interrupts, and 2) How to trace interrupt routines.

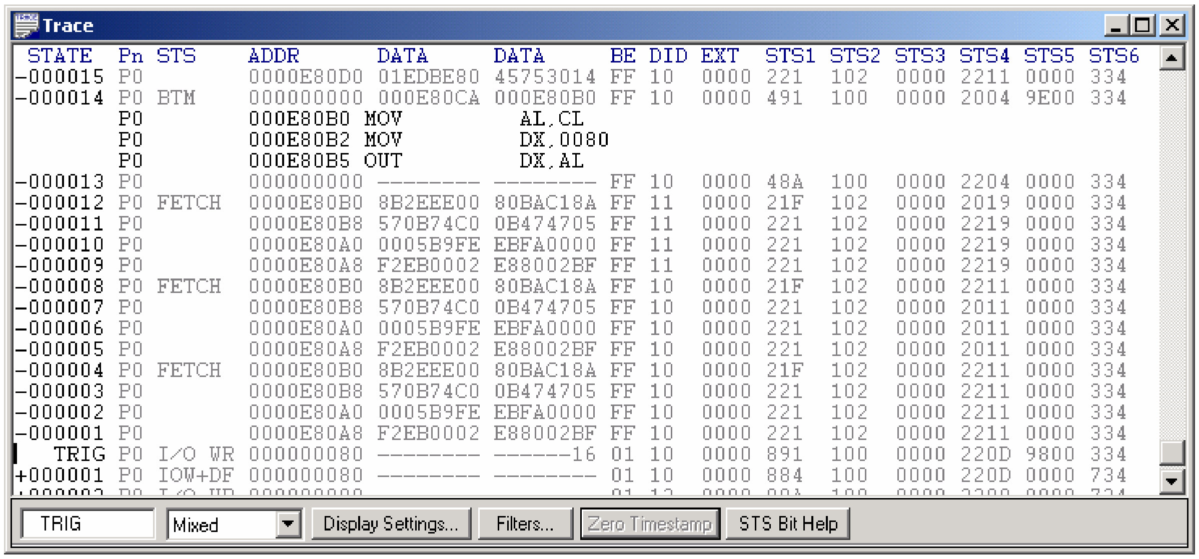
For BIOS development, a potential use of Bus Analyzer breakpoints is to break on a specific POST code. To do this, simply open the Breakpoint Window and select the Add button. In the Break On drop down box, select I/O Write. For Location, select 80h (the POST code address), and for value, choose 16h or whatever value you are looking for. The Figure 7 shows the Add Breakpoint dialog box for the breakpoint.


Add Breakpoint

Break on:	I/O Write
Resource:	Bus Analyzer
Processor:	
Location:	80h
Translate:	
Length:	Byte
Data:	16h
External:	
Sequence:	T1 (participates in Term 1)

Figure 7

Open the Trace window after hitting this breakpoint and examine the trigger point. The easiest way to get to the trigger line of the Trace window is to enter a "t" in the Line Number Box of the window. This is the box that is probably in the lower left corner of the window, corresponding roughly to the Address Text Box location of the Code window. Figure 8 shows the trace window as it appears after hitting the breakpoint used in this example.



```
 STATE   Pn STS      ADDR       DATA      DATA      BE DID EXT  STS1 STS2 STS3 STS4 STS5 STS6
-000015 P0          0000E80D0 01EDBE80 45753014 FF 10  0000 221  102  0000 2211 0000 334
-000014 P0 BTM      000000000 000E80CA 000E80B0 FF 10  0000 491  100  0000 2004 9E00 334
        P0          000E80B0 MOV         AL,CL
        P0          000E80B2 MOV         DX,0080
        P0          000E80B5 OUT         DX,AL
-000013 P0          000000000 -------- -------- FF 10  0000 48A  100  0000 2204 0000 334
-000012 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 11  0000 21F  102  0000 2019 0000 334
-000011 P0          0000E80B8 570B74C0 0B474705 FF 11  0000 221  102  0000 2219 0000 334
-000010 P0          0000E80A0 0005B9FE EBFA0000 FF 11  0000 221  102  0000 2219 0000 334
-000009 P0          0000E80A8 F2EB0002 E88002BF FF 11  0000 221  102  0000 2219 0000 334
-000008 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000007 P0          0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2011 0000 334
-000006 P0          0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2011 0000 334
-000005 P0          0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2011 0000 334
-000004 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000003 P0          0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2211 0000 334
-000002 P0          0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2211 0000 334
-000001 P0          0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2211 0000 334
   TRIG P0 I/O WR   000000080 -------- -------16 01 10  0000 891  100  0000 220D 9800 334
+000001 P0 IOW+DF   000000080 -------- -------- 01 10  0000 884  100  0000 220D 0000 734
```

Figure 8

For further information, see the application note at http://www.arium.com/support/pdf/swdebug.pdf for a more complete treatment of this subject.

MSVC // Embedded Power LL386 tool chain

The same source code that can be used for the Tornado II tool chain example can be used for the Microsoft Visual C // Embedded Power Link&Locate386 tool chain. For the sake of simplicity, however, all of the files necessary for this exercise decompress into a directory called MSVC.

Building the Executable

There are as many methods for building the executable for this tool chain, as there are engineers. Many of you will prefer to use either the Microsoft Visual C graphical development environment, while others will choose to work with the Embedded Power Build Wizard. Still others will work in the time-tested makefile paradigm. As with Tornado II, we assume that those who are using Integrated Development Environments are well versed in their use. For those who prefer makefiles, we will show some basic commands to include.

For the purposes of this exercise, we will keep it as simple as possible. We will show the basic commands that are necessary to compile and then link the files that are included with this application note. It is not necessary to re-compile the files; we only include these steps to illustrate a build that is compatible with hardware-assisted debugging.

To properly run these commands, you must first run the batch files to initialize the environment variables for the MSVC and Embedded Power LL386 tools. The MSVC environment variables typically are initialized by the following command: C:\MSDEV\VC98\BIN\VCVARS32.BAT. The Embedded Power LL386 batch file is typically C:\BEACON\LL 386\BIN\ENV.BAT.

After initializing the environment, open a DOS window. Change the working directory to the MSVC sub-directory under the directory where the exercises were loaded on your host. Now it is time to compile the source code.
At the DOS prompt type:
cl csample.c /Zi /c <enter>
cl csample2.c /Zi /c <enter>

These commands compile the source code files for this exercise. The /Zi flag enables the inclusion of debugging information. The /c flag prevents the Microsoft tool from attempting to link the output files. We will use the Embedded Power tool for that function.

Note: There are other debug flags used with this tool (Z7, Zd et. al.), but /Zi is recommended.

To run the linker simply type:
xlink386 @flat <enter>

This links the output of the compiler for the example files. If you are building your own code, replace the word “flat” with the name of your own build file.

Booting the Target

Because the OMF file contains all of the information necessary to properly initialize the registers of the target, it is not necessary to boot the target prior to loading your OMF file.

Loading the Executable

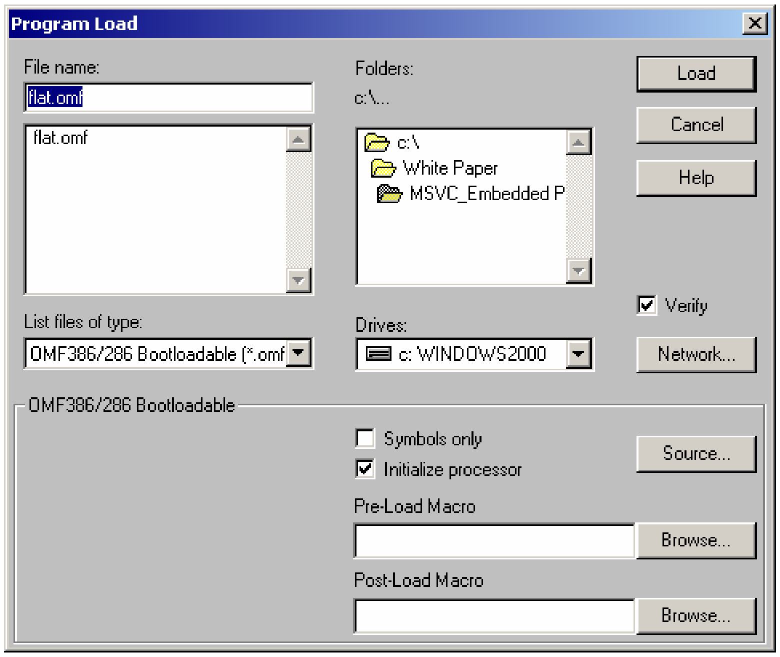
Now that you have an .omf file, you are ready to load the code and begin the debugging process. Ensure that the emulator and target are connected properly and that both are turned on.

Now it is time to open SourcePoint. Once SourcePoint is open, loading symbols is as easy as loading the program object that you created with MSVC // Embedded Power LL386 (remember: turn symbols on with the /Zi compiler option). If you wish to forego the exercise of building the code yourself, you can load the flat.omf file that is included with the distribution.

Before you endeavor to load the symbols, you should familiarize yourself first with the SourcePoint interface by reading the flyover text for the various icon buttons. The buttons are the easiest way for a beginner to control the emulator. Find the “reset” icon button on the SourcePoint interface, and click it with the mouse pointer. Now open a Code window. (Find the icon button with the flyover text “Code Window” and click it.) Observe that the IP is at the reset vector.

Now click the “Go” button and wait until you hear a pause in the activity on the drive that you’re using to boot the target. Then click the “Stop” button, and with the target halted, load either the flat.omf file included with the exercises or the flat.omf file that you created with your own build. The flat.omf can be found under the directory where you installed the exercises at …\MSVC_Embedded Power LL386\flat.omf.

To load the code to the Intel 440BX board from SourcePoint, click the “Program Load” button and choose files of type “OMF” as shown in Figure 9. Make sure that the “Initialize processor” check box is enabled.



File name: flat.omf

flat.omf

Folders: c:\...

c:\
White Paper
MSVC_Embedded P

List files of type: OMF386/286 Bootloadable (*.omf

Drives: c: WINDOWS2000

☑ Verify

☐ Symbols only
☑ Initialize processor

Pre-Load Macro

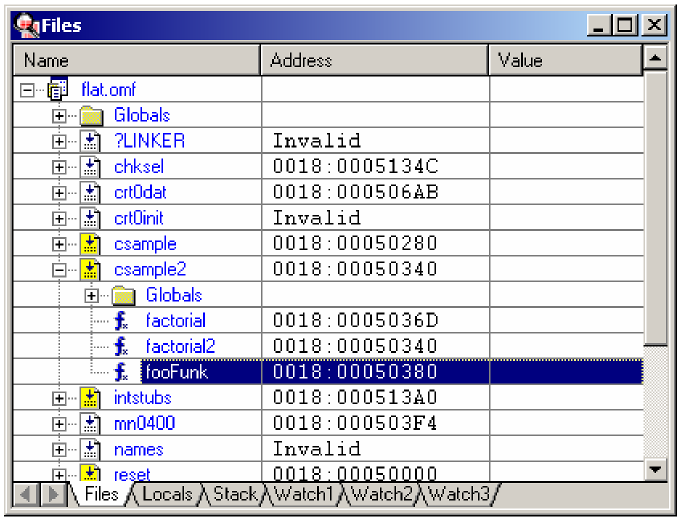
Post-Load Macro

Figure 9

Source Code Path

By default, SourcePoint looks for requested source code in the same directory from which your program object was loaded. Most commonly, your source code will be elsewhere. When SourcePoint needs a source code file as it is analyzing debug symbols, it will request them. These requests can become tedious when you have source code spread over several directories. Alternately, you can specify a list of paths to search at load time. This list is reached from the Program Load dialog box by selecting the “Source” button. Enter the list of paths to be searched, separated by semi-colons. This list is saved so that next time you load the same program object, the same directory list will be used.

Immediately upon loading the program object that contains the debugging symbols, the Symbols window (click the “Symbols Window” button) will show your program’s symbols in a tree structure, as shown below in Figure 10.



Name	Address	Value
flat.omf		
Globals		
?LINKER	Invalid	
chksel	0018:0005134C	
crt0dat	0018:000506AB	
crt0init	Invalid	
csample	0018:00050280	
csample2	0018:00050340	
Globals		
factorial	0018:0005036D	
factorial2	0018:00050340	
fooFunk	0018:00050380	
intstubs	0018:000513A0	
mn0400	0018:000503F4	
names	Invalid	
reset	0018:00050000	

Figure 10

Setting breakpoints

Once your symbols are loaded, you may want to set a breakpoint on a procedure. The procedures are listed in the

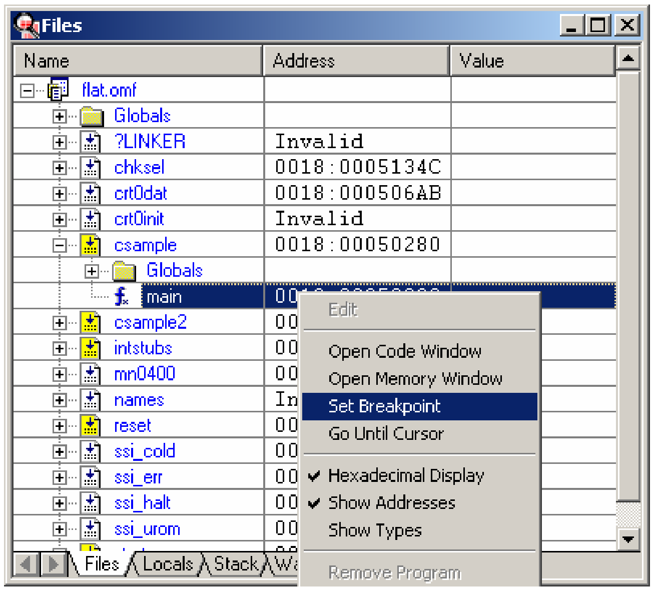
Symbols window under their respective modules (modules roughly correspond to files). The best place to start is at the beginning of the source code. To find this, click to expand the files under flat.omf, then click to expand the files under csample. This is where you will find the source code for this exercise.

Right click on the file named “main”. Figure 11 shows the context sensitive menu that pops up when you do this.

With the breakpoint set, continue execution of the code and wait for the breakpoint to be hit. If you have already continued execution before reading this far, you may be past the point where this breakpoint will be encountered. If this is the case, simply set a breakpoint on fooFunk the same way. The function fooFunk can be found under csample2. While you are learning how to use the emulator, you may want to open the Breakpoint window to examine the breakpoint that you have just set. The Figure 12 shows the result of setting and hitting this breakpoint and also shows the contents of the Breakpoint window.

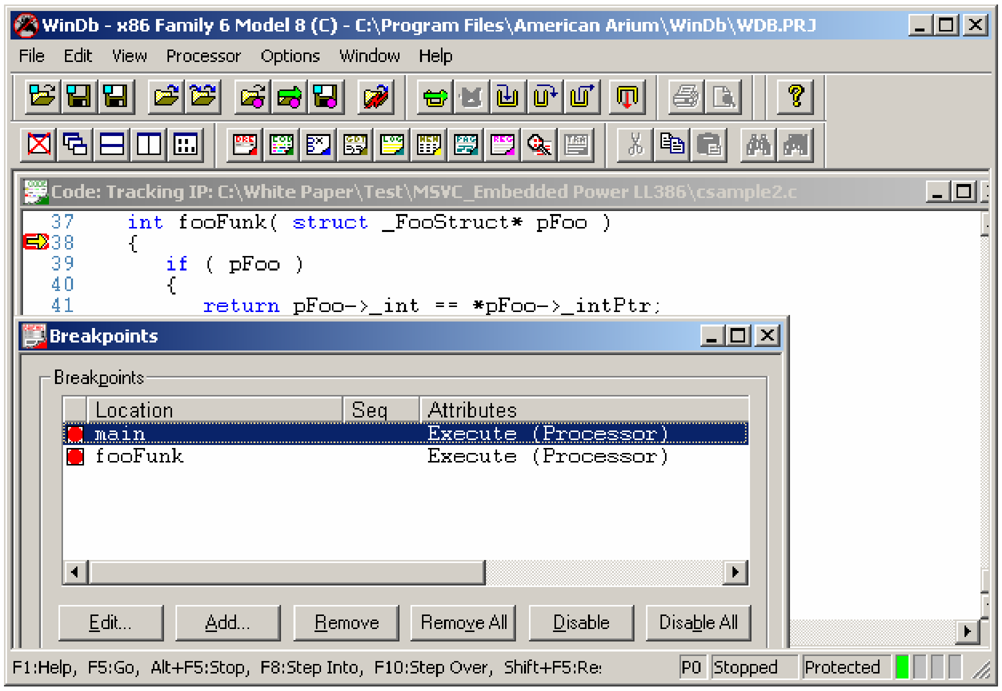
Another method of setting a breakpoint at the beginning of a function is to type the symbol for the function in the Address text box of the Code window (usually found in the lower left corner of the Code window if you haven’t customized the window). If SourcePoint recognizes the symbol, the symbol will be replaced by the virtual address immediately upon hitting the enter key. This will show the procedure in the Code window and will allow you to place the cursor at the start of the function. The breakpoint can then be set by pressing the F9 key.

Alternatively, you can open the Breakpoint window and select the Add button. This will open the Add Breakpoint dialog box. In the Location section of this dialog box, type the symbol for the function. Unlike the Address text box of the Code window, the symbol will not be replaced by the virtual address.



Name	Address	Value
flat.omf		
Globals		
?LINKER	Invalid	
chksel	0018:0005134C	
crt0dat	0018:000506AB	
crt0init	Invalid	
csample	0018:00050280	
Globals		
main	00	
csample2	00	
intstubs	00	
mn0400	00	
names	In	
reset	00	
ssi_cold	00	
ssi_err	00	
ssi_halt	00	
ssi_urom	00	

Figure 11


WinDb - x86 Family 6 Model 8 (C) - C:\Program Files\American Arium\WinDb\WDB.PRJ
File Edit View Processor Options Window Help
Code: Tracking IP: C:\White Paper\Test\MSVC_Embedded Power LL386\csample2.c
Breakpoints
Edit... Add... Remove Remove All Disable Disable All

```
int fooFunk( struct _FooStruct* pFoo )
{
    if ( pFoo )
    {
        return pFoo->_int == *pFoo->_intPtr;
```

Location	Seq	Attributes
main		Execute (Processor)
fooFunk		Execute (Processor)

F1:Help, F5:Go, Alt+F5:Stop, F8:Step Into, F10:Step Over, Shift+F5:Re: P0 Stopped Protected

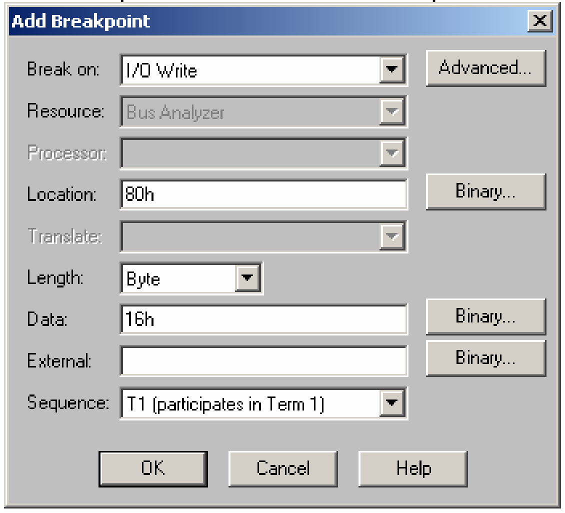
Figure 12

Using the TRC-20

As previously mentioned, the principal advantages of an ICE over a JTAG run control device (ITP) are: the ability to set Bus Analyzer breakpoints, the ability to capture a trace of either the bus or the assembly instructions or both, and the ability to break or trace based on complex events.

An excellent application note that covers the use of these features can be found at http://www.arium.com/support/pdf/Measurement.PDF. This application note covers: 1) How to determine the time between and duration of interrupts, and 2) How to trace interrupt routines.

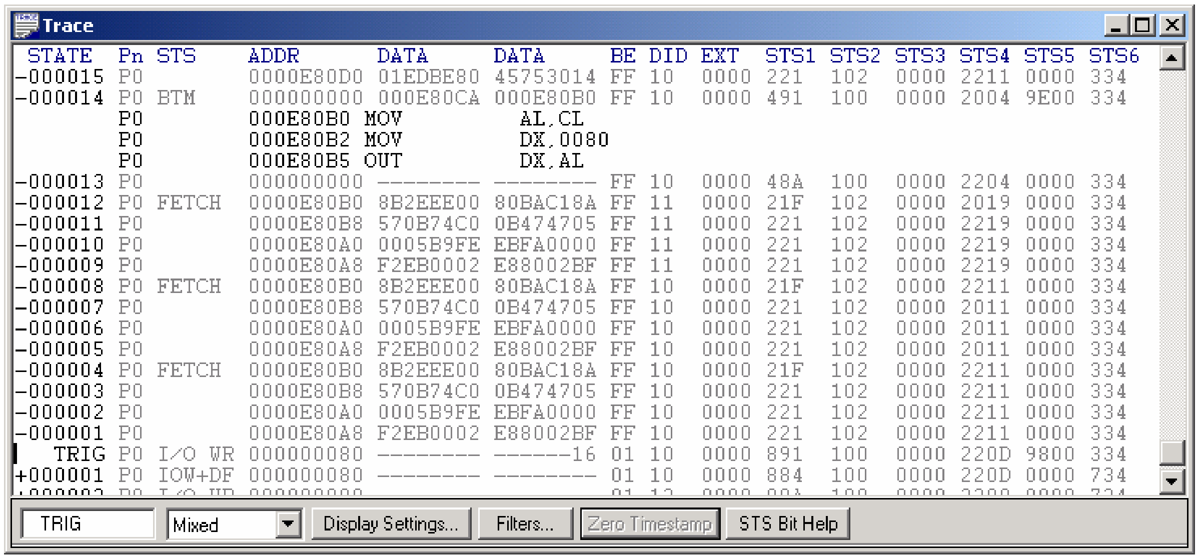
For BIOS development, a potential use of Bus Analyzer breakpoints is to break on a specific POST code. To do this, simply open the Breakpoint window and select the Add button. In the Break On drop down box, select I/O Write. For Location, select 80h (the POST code address), and for Value, choose 16h or whatever value you are looking for. Figure 13 shows the Add Breakpoint window for the breakpoint.



Break on:	I/O Write
Resource:	Bus Analyzer
Processor:	
Location:	80h
Translate:	
Length:	Byte
Data:	16h
External:	
Sequence:	T1 (participates in Term 1)

Figure 13

Open the Trace window after hitting this breakpoint and examine the trigger point. The easiest way to get to the trigger line of the Trace window is to enter a “t” in the Line Number Box of the window. This is the box that is probably in the lower left corner of the window, corresponding roughly to the Address text box of the Code window. Figure 14 shows the trace window as it appears after hitting the breakpoint used in this example.



```
 STATE   Pn STS      ADDR       DATA     DATA     BE DID EXT  STS1 STS2 STS3 STS4 STS5 STS6
-000015 P0           0000E80D0 01EDBE80 45753014 FF 10  0000 221  102  0000 2211 0000 334
-000014 P0 BTM       000000000 000E80CA 000E80B0 FF 10  0000 491  100  0000 2004 9E00 334
        P0           000E80B0 MOV          AL,CL
        P0           000E80B2 MOV          DX,0080
        P0           000E80B5 OUT          DX,AL
-000013 P0           000000000 -------- -------- FF 10  0000 48A  100  0000 2204 0000 334
-000012 P0 FETCH     0000E80B0 8B2EEE00 80BAC18A FF 11  0000 21F  102  0000 2019 0000 334
-000011 P0           0000E80B8 570B74C0 0B474705 FF 11  0000 221  102  0000 2219 0000 334
-000010 P0           0000E80A0 0005B9FE EBFA0000 FF 11  0000 221  102  0000 2219 0000 334
-000009 P0           0000E80A8 F2EB0002 E88002BF FF 11  0000 221  102  0000 2219 0000 334
-000008 P0 FETCH     0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000007 P0           0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2011 0000 334
-000006 P0           0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2011 0000 334
-000005 P0           0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2011 0000 334
-000004 P0 FETCH     0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000003 P0           0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2211 0000 334
-000002 P0           0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2211 0000 334
-000001 P0           0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2211 0000 334
   TRIG P0 I/O WR    000000080 -------- ------16 01 10  0000 891  100  0000 220D 9800 334
+000001 P0 IOW+DF    000000080 -------- -------- 01 10  0000 884  100  0000 220D 0000 734
```

Figure 14

For more information, see the application note at http://www.arium.com/support/pdf/swdebug.pdf for a more complete treatment of this subject.

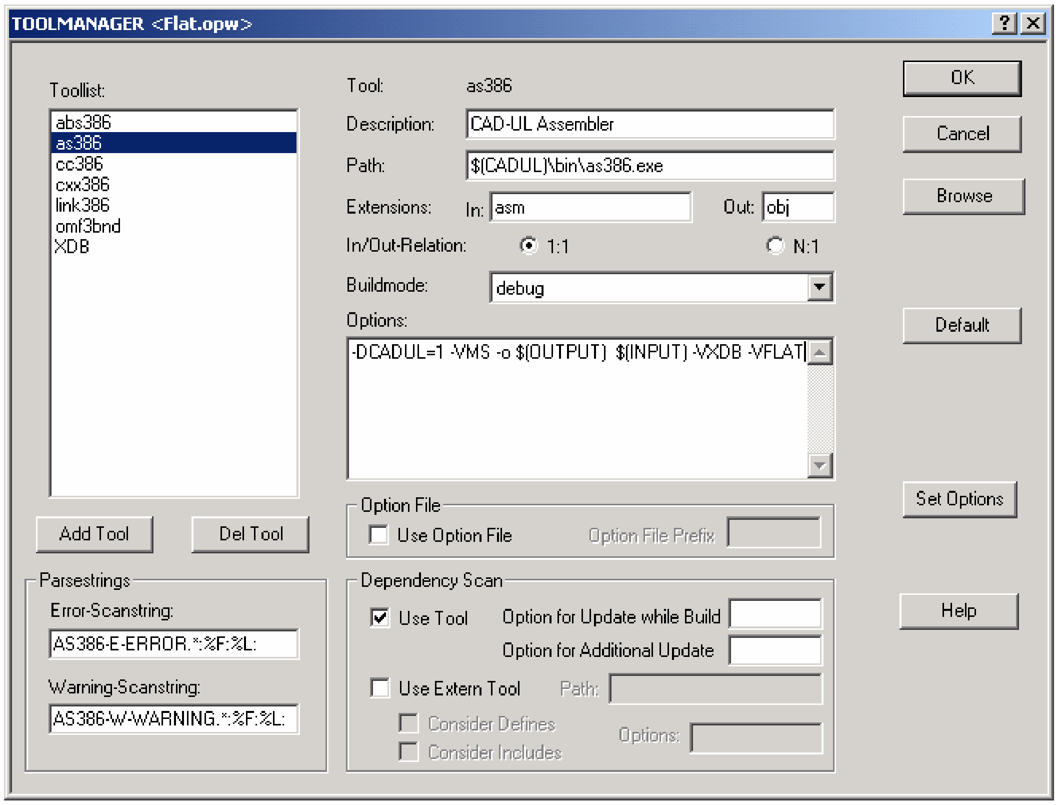
CAD-UL Tool Chain

The same source code that can be used for the other tool chains can be used for the CAD-UL tool chain. For the sake of simplicity, however, all of the files necessary for this exercise decompress into a directory called CAD-UL.

Building the Executable

The CAD-UL Integrated Development Environment (IDE) is in many ways more similar to the Wind River Tornado II tool chain than to MSVC. For that reason, it is assumed that you will be able to use CAD-UL to produce the needed executable. The following screen captures show the settings that were used to build the sample files. The object code need not be rebuilt. The .omf file that comes with the exercises will suffice for the examples here.

Figure 15 shows the settings that were used for the assembler for the build of this example. Figure 16 shows the settings for the compiler for the build of this example. Note especially the use of the –VXDB option. This option is instrumental in producing the symbols used by SourcePoint.


TOOLMANAGER <Flat.opw>

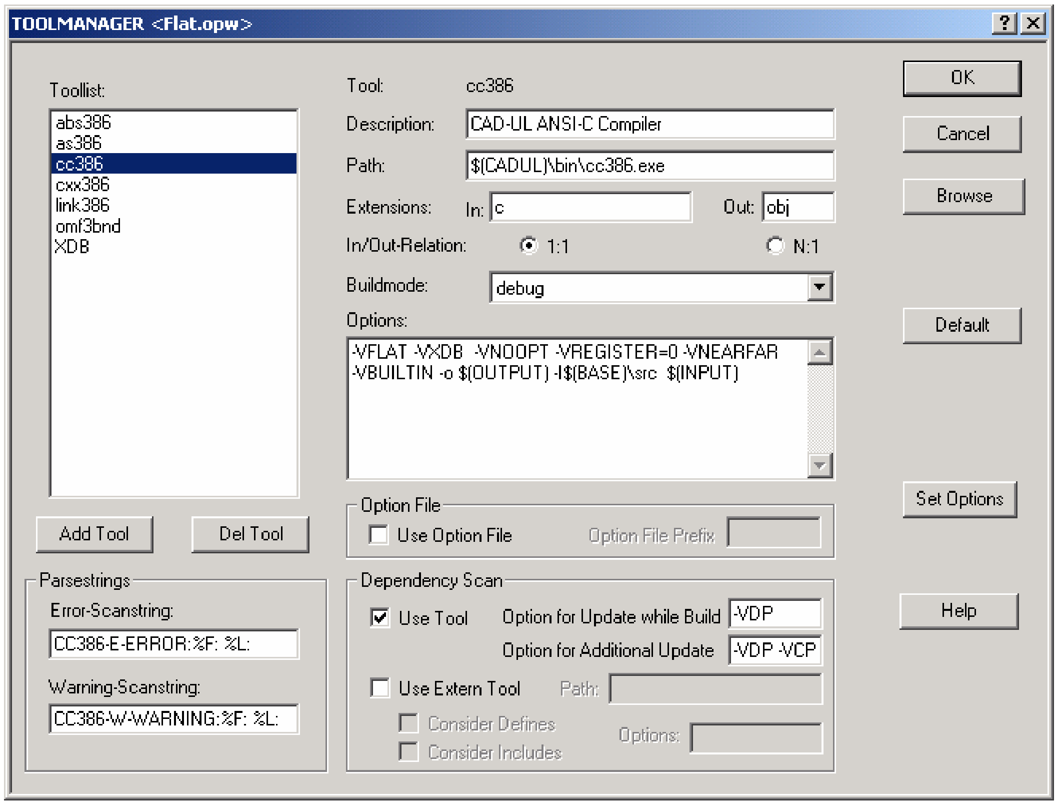
Toollist:
abs386
as386
cc386
cxx386
link386
omf3bnd
XDB

Tool: as386
Description: CAD-UL Assembler
Path: $(CADUL)\bin\as386.exe
Extensions: In: asm Out: obj
In/Out-Relation: [✓ 1:1] N:1
Buildmode: debug
Options:
```
-DCADUL=1 -VMS -o $(OUTPUT) $(INPUT) -VXDB -VFLAT
```
Option File
☐ Use Option File   Option File Prefix:

Parsestrings
Error-Scanstring: AS386-E-ERROR.*:%F:%L:
Warning-Scanstring: AS386-W-WARNING.*:%F:%L:

Dependency Scan
☑ Use Tool   Option for Update while Build: Option for Additional Update:
☐ Use Extern Tool   Path:
☐ Consider Defines   Options:
☐ Consider Includes

Figure 15



TOOLMANAGER <Flat.opw>

Toollist:
- abs386
- as386
- cc386
- cxx386
- link386
- omf3bnd
- XDB

Tool: cc386
Description: CAD-UL ANSI-C Compiler
Path: $(CADUL)\bin\cc386.exe
Extensions: In: c Out: obj
In/Out-Relation: [✓ 1:1] N:1
Buildmode: debug
Options:

```
-VFLAT -VXDB  -VNOOPT -VREGISTER=0 -VNEARFAR
-VBUILTIN -o $(OUTPUT) -I$(BASE)\src  $(INPUT)
```

Option File
☐ Use Option File   Option File Prefix

Parsestrings
Error-Scanstring: CC386-E-ERROR:%F: %L:
Warning-Scanstring: CC386-W-WARNING:%F: %L:

Dependency Scan
☑ Use Tool   Option for Update while Build: -VDP
Option for Additional Update: -VDP -VCP
☐ Use Extern Tool   Path:
☐ Consider Defines   Options:
☐ Consider Includes

Figure 16

Loading the Executable

Now that you have an .omf file, you are ready to load the code and begin the debugging process. Ensure that the emulator and target are properly connected and that both are turned on.

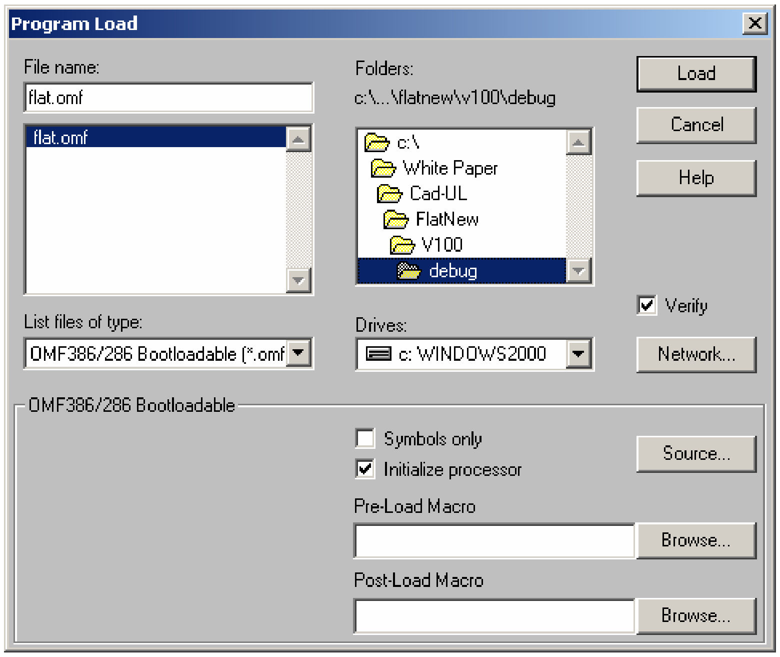
Now it is time to open SourcePoint. Once SourcePoint is open, loading symbols is as easy as loading the program object that you created with CAD-UL (remember: turn symbols on with the -VXDB compiler option).
Before you endeavor to load the symbols you should first familiarize yourself with the SourcePoint interface by reading the flyover text for the various icon buttons. The icon buttons are the easiest way for a beginner to control the emulator. Find the "reset" icon button on the SourcePoint interface and click it with the mouse pointer. Now open a Code window. (Find the button with the flyover text "Code Window" and click it.) Observe that the IP is at the reset vector.

Now click the "Go" button and wait until you hear a pause in the activity on the drive that you're using to boot the target. Then click the "Stop" button, and with the target halted, load either the flat.omf file included with the exercises or the flat.omf file that you created with your own build. The flat.omf file can be found under the directory where you installed the exercises at …\Cad-UL\Flat\V100\debug\flat.omf.

To load the code to the Intel 440BX board from SourcePoint, click the "Program Load" button and choose files of type "OMF" as shown in Figure 17. Make sure that the "Initialize processor" check box is enabled.

Booting the Target

Because the OMF file contains all of the information necessary to properly initialize the registers of the target, it is not necessary to boot the target prior to loading your OMF file.



File name: flat.omf

flat.omf

Folders: c:\...\flatnew\v100\debug

c:\
White Paper
Cad-UL
FlatNew
V100
debug

List files of type: OMF386/286 Bootloadable (*.omf

Drives: c: WINDOWS2000

☑ Verify

☐ Symbols only

☑ Initialize processor

Pre-Load Macro:

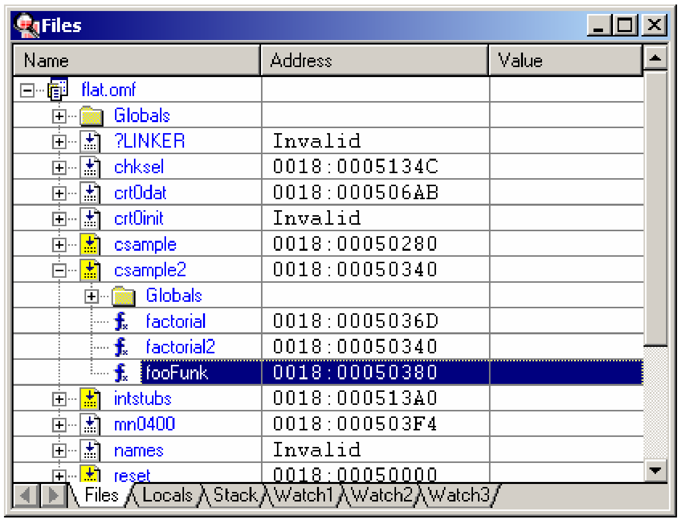
Post-Load Macro:

Figure 17

Source Code Path

By default, SourcePoint looks for requested source code in the same directory from which your program object was loaded. Most commonly, your source code will be elsewhere. When SourcePoint needs a source code file as it is analyzing debug symbols, it will request them. These requests can become tedious when you have source code spread over several directories. Alternately, you can specify a list of paths to search at load time. This list is reached from the Program Load dialog box by selecting the “Source” button. Enter the list of paths to be searched, separated by semi-colons. This list is saved so that next time you load the same program object, the same directory list will be used.

Immediately on loading the program object that contains the debugging symbols, the Symbols window (click the “Symbols Window” button) will show your program’s symbols in a tree structure, as shown below in Figure 18.



Name	Address	Value
flat.omf		
Globals		
?LINKER	Invalid	
chksel	0018:0005134C	
crt0dat	0018:000506AB	
crt0init	Invalid	
csample	0018:00050280	
csample2	0018:00050340	
Globals		
factorial	0018:0005036D	
factorial2	0018:00050340	
fooFunk	0018:00050380	
intstubs	0018:000513A0	
mn0400	0018:000503F4	
names	Invalid	
reset	0018:00050000	

Figure 18

Setting breakpoints

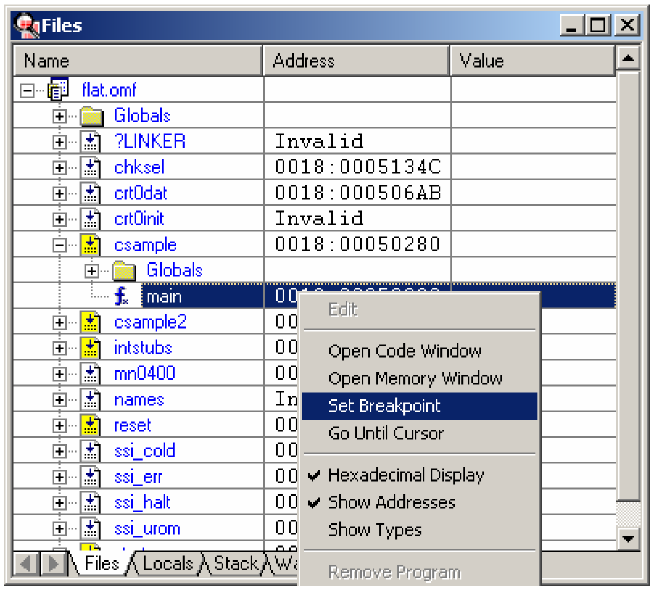
Once your symbols are loaded, you may want to set a breakpoint on a procedure. The procedures are listed in the Symbols window under their respective modules (modules roughly correspond to files). The best place to start is at the beginning of the source code. To find this, click to expand the files under flat.omf, then click to expand the files under csample. This is where you will find the source code for this exercise.

Right click on the file named “main”. Figure 19 shows the context sensitive menu that pops up when you do this.

With the breakpoint set, continue execution of the code and wait for the breakpoint to be hit. If you have already continued execution before reading this far, you may be past the point where this breakpoint will be encountered. If this is the case, simply set a breakpoint on fooFunk the same way. The function fooFunk can be found under csample2. While you are learning how to use the emulator, you may want to open the Breakpoint window to examine the breakpoint that you have just set. Figure 20 on the following page shows the result of setting and hitting this breakpoint and also shows the contents of the Breakpoint window.

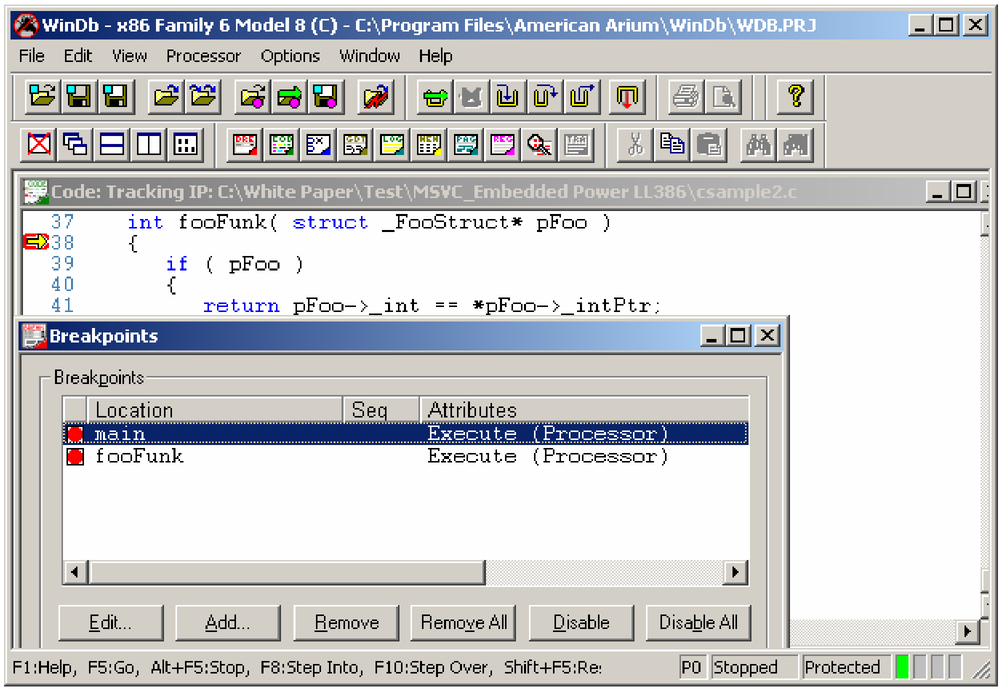
Another method of setting a breakpoint at the beginning of a function is to type the symbol for the function in the Address text box of the Code window (usually found in the lower left corner of the Code window if you haven’t customized the window). If SourcePoint recognizes the symbol, the symbol will be replaced by the virtual address immediately upon hitting the enter key. This will show the procedure in the Code window and will allow you to place the cursor at the start of the function. The breakpoint can then be set by pressing the F9 key.

Alternatively, you can open the Breakpoint Window and select the Add button. This will open the Add Breakpoint dialog box. In the Location section of this dialog, type the symbol for the function. Unlike the Address text box of the Code window, the symbol will not be replaced by the virtual address.



Name	Address	Value
flat.omf		
Globals		
?LINKER	Invalid	
chksel	0018:0005134C	
crt0dat	0018:000506AB	
crt0init	Invalid	
csample	0018:00050280	
Globals		
main	00	
csample2	00	
intstubs	00	
mn0400	00	
names	In	
reset	00	
ssi_cold	00	
ssi_err	00	
ssi_halt	00	
ssi_urom	00	

Figure 19


WinDb - x86 Family 6 Model 8 (C) - C:\Program Files\American Arium\WinDb\WDB.PRJ

Code: Tracking IP: C:\White Paper\Test\MSVC_Embedded Power LL386\csample2.c

```
int fooFunk( struct _FooStruct* pFoo )
{
    if ( pFoo )
    {
        return pFoo->_int == *pFoo->_intPtr;
```

Breakpoints

	Location	Seq	Attributes
🟥	main		Execute (Processor)
🟥	fooFunk		Execute (Processor)

F1:Help, F5:Go, Alt+F5:Stop, F8:Step Into, F10:Step Over, Shift+F5:Re P0 Stopped Protected

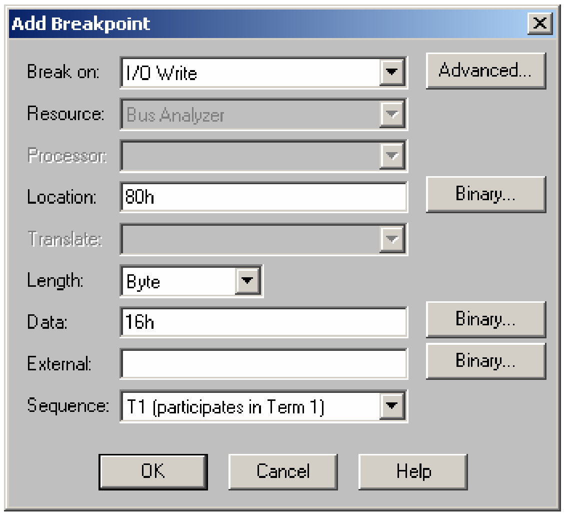
Figure 20

Using the TRC-20

As previously mentioned, the principal advantages of an ICE over a JTAG run control device (ITP) are: the ability to set Bus Analyzer breakpoints, the ability to capture a trace of either the bus or the assembly instructions or both, and the ability to break or trace based on complex events.

An excellent application note that covers the use of these features can be found at http://www.arium.com/support/pdf/Measurement.PDF. This application note covers: 1) How to determine the time between and duration of interrupts, and 2) How to trace interrupt routines.

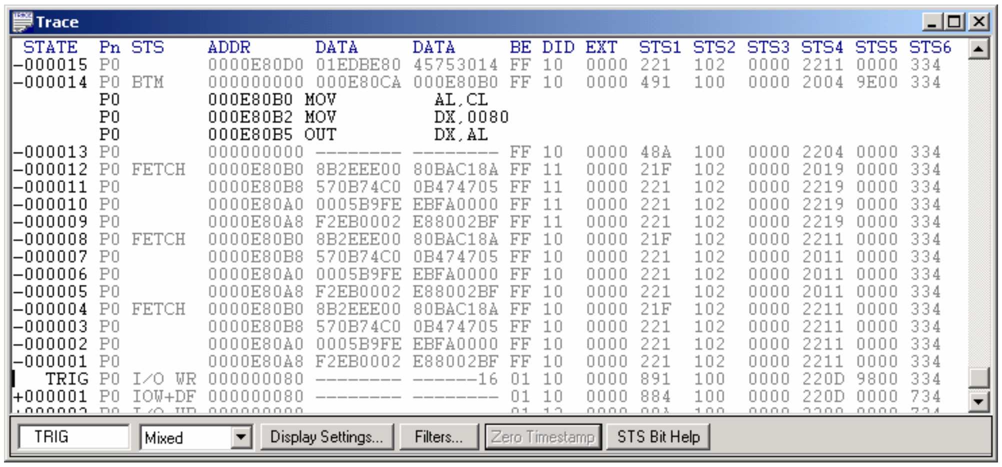
For BIOS development, a potential use of Bus Analyzer breakpoints is to break on a specific POST code. To do this, simply open the Breakpoint window and select the Add button. In the Break On drop down box, select I/O Write. For Location, select 80h (the POST code address), and for Value, choose 16h or whatever value you are looking for. Figure 21 shows the Add Breakpoint window for the breakpoint.



Break on: I/O Write
Resource: Bus Analyzer
Processor:
Location: 80h
Translate:
Length: Byte
Data: 16h
External:
Sequence: T1 (participates in Term 1)

Figure 21

Open the Trace Window after hitting this breakpoint and examine the trigger point. The easiest way to get to the trigger line of the Trace window is to enter a "t" in the Line Number Box of the window. This is the box that is probably in the lower left corner of the window, corresponding roughly to the Address text box of the Code window. Figure 22 shows the trace window as it appears after hitting the breakpoint used in this example.



```
 STATE   Pn STS      ADDR      DATA     DATA     BE DID EXT  STS1 STS2 STS3 STS4 STS5 STS6
-000015 P0          0000E80D0 01EDBE80 45753014 FF 10  0000 221  102  0000 2211 0000 334
-000014 P0 BTM      000000000 000E80CA 000E80B0 FF 10  0000 491  100  0000 2004 9E00 334
        P0          000E80B0 MOV          AL,CL
        P0          000E80B2 MOV          DX,0080
        P0          000E80B5 OUT          DX,AL
-000013 P0          000000000 -------- -------- FF 10  0000 48A  100  0000 2204 0000 334
-000012 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 11  0000 21F  102  0000 2019 0000 334
-000011 P0          0000E80B8 570B74C0 0B474705 FF 11  0000 221  102  0000 2219 0000 334
-000010 P0          0000E80A0 0005B9FE EBFA0000 FF 11  0000 221  102  0000 2219 0000 334
-000009 P0          0000E80A8 F2EB0002 E88002BF FF 11  0000 221  102  0000 2219 0000 334
-000008 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000007 P0          0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2011 0000 334
-000006 P0          0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2011 0000 334
-000005 P0          0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2011 0000 334
-000004 P0 FETCH    0000E80B0 8B2EEE00 80BAC18A FF 10  0000 21F  102  0000 2211 0000 334
-000003 P0          0000E80B8 570B74C0 0B474705 FF 10  0000 221  102  0000 2211 0000 334
-000002 P0          0000E80A0 0005B9FE EBFA0000 FF 10  0000 221  102  0000 2211 0000 334
-000001 P0          0000E80A8 F2EB0002 E88002BF FF 10  0000 221  102  0000 2211 0000 334
   TRIG P0 I/O WR   000000080 -------- ------16 01 10  0000 891  100  0000 220D 9800 334
+000001 P0 IOW+DF   000000080 -------- -------- 01 10  0000 884  100  0000 220D 0000 734
```

Figure 22

For more information, see the application note at http://www.arium.com/support/pdf/swdebug.pdf for a more complete treatment of this subject.