American Arium
Application Note

Linux Kernel Debug: Using Run Control Hardware to Facilitate Linux Kernel Development

Overview

This application note takes the reader through a sample software development environment with American Arium's KIT-XA4: Excalibur™ Quick Start Kit, SC-1000/SC-1000A in-circuit emulator (ICE), and SourcePoint™ debugger, illustrating the use of hardware run control in the process. The KIT-XA4 includes a complete source code tree and downloadable executables for Linux, produced in conjunction with Microtronix Datacom Ltd.

The Advantages of Hardware Run Control

Modern Integrated Development Environments (IDEs) 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 the Board Support Package (BSP), OS kernels, and the device drivers, however, something more is needed.

Especially at the lower levels of development (BSP), 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 the developer needs 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 at determined times, system debug can become very complicated. In a heavily interrupt driven, real time environment, code visibility and control is invaluable. Hardware-assisted debug tools offer a powerful solution.

The SC-1000/SC-1000A, which connects to the target through the JTAG port with the PBD-1KJ personality board, gives the developer access to EmbeddedICE watchpoint and vector catching functions as well as the ability to set software breakpoints. In addition, the ETM (Embedded Trace Macrocell), when used with the PBD-1KE personality board, adds the capability of instruction and data tracing. Trace adds immensely to the visibility of the debug environment and provides 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 the developer to simply set a breakpoint, run to it, and then analyze the trace buffer, rather than single stepping through large amounts of code. The ETM also gives him/her 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 can be used in conjunction with the trace tool to define trace events. This allows the developer 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. This hardware-assisted debug solution allows the creative programmer to find bugs in a fraction of the time required by other tools.

Microtronix Embedded Linux

Included in the KIT-XA4 software installation is the entire source tree to run Linux on the REF-XA4. This Linux is a modified version of Linux 2.5.30-rmk1. Use the procedure described below to load the Linux image and source files through the SourcePoint debugging interface. This procedure is also described in the *Excalibur™ Quick Start Guide* included with the KIT-XA4.

Target Configuration:

- Check jumpers on the REF-XA4 board. They should all be in their default positions per the Arium *User's Guide* except for JP19 (1-2; this is the correct placement), JP21 and JP22 (1-2 as well, routing clocks from phy to EPXA4).
- Attach a null modem cable between serial port J12 on the REF-XA4 and the serial port on the host PC. Start a HyperTerminal session at 38,400, 8N1, flow control off. This cable is NOT included with your kit.

Note: If the Linux kernel includes special console code available from American Arium, Linux console I/O can be accessed through SourcePoint's **Target Console** window.

- Connect a network patch cable from the Ethernet connector J15 on the REF-XA4 to your network, or connect your host PC and the REF-XA4 to a hub.
- Insert a compatible memory DIMM in the REF-XA4 memory slot (J10). This procedure was verified with a 128MB SDRAM (non-parity, PC133), Crucial part # CT16M64S4D7E.

Target Initialization:

Now it is time to open SourcePoint. 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** (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) and observe that the IP is at the reset vector (PC=0).

The macro file "linuxboot.mac" in the default install folder "C:\Program Files\American Arium\Kit-XA4\Linux" contains SourcePoint commands that will initialize the REF-XA4 target as configured as well as load the Linux binary code and symbol information. The manual procedure is described as follows:

1. Reset the target.
2. Load the **REFXA4LINUXINIT.MAC** macro file (**File|Macro|Load Macro**) located in the **Linux** directory defined above. This macro initializes the Excalibur EPXA4 registers.
3. Load the **sdram_init** program (**File|Program|Load Program**) found in the **Linux** directory defined above, with **Offset** of 0, **Verify** enabled, **Symbols Only** disabled, and **Initialize Processor** disabled.
4. Set **PC=20000000** through the **Command** window or current **Register** view. SRAM has been mapped to this location; this is where this code loads.
5. While still in the **Command** window, type: **Go til 200000D4.**

The SDRAM is now initialized and located at 0, ready for kernel loading.

Loading the Linux Binary and Symbols

Now that the target has initialized memory, you can download the binary Linux code to the target, as well as the symbolic information. Support includes C++ source code level debugging, symbolic procedures, variables, and type information. The Linux code included with the KIT-XA4 was compiled into ELF binaries using GNU compilers. One version was produced with Dwarf2 source and symbolic information (**vmlinux-2.5.30-rmk1-mk1.nostrip**), and one was produced with no symbolic information (**vmlinux-2.5.30-rmk1-mk1.strip**).

1. From the **File** menu, chose **Program|Load Program**. Select the Linux ELF file, **vmlinux-2.5.30-rmk1-mk1.strip,** found in the Linux directory defined above. Disable **Symbols Only** and **Initialize Processor**, enable **Verify**, and set **Offset** to -0x0C0000000. This program load sends the Linux image to the target at the appropriate offset.
2. From the **File** menu, chose **Program|Load Program**. Select the Linux ELF file, **vmlinux-2.5.30-rmk1-mk1.nostrip,** found in the Linux directory defined above. Enable **Symbols Only,** disable **Initialize Processor**, and set **Offset** to 0. This program load sends symbol and source information for the Linux image to SourcePoint at the appropriate offset.

Note: The binary code and symbols are loaded to different offsets do to memory re-mapping that occurs after the code starts executing. See Figure 1 for a sample **Program Load** dialog box.

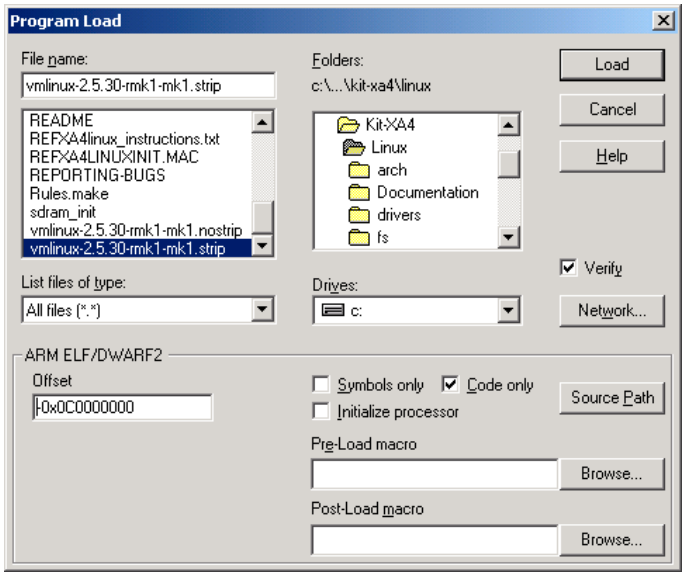

Figure 1

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. If there is path information contained in the symbolic data, SourcePoint will attempt to determine the appropriate mapping. 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. See Figure 2. 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.

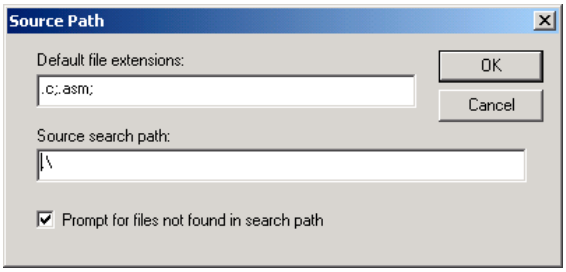

Figure 2

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). Expand the **intrep** module in the **Globals** tab of the **Symbols** window. Note that there is flyover text for each module indicating the path to the associated source file. Right click on **jffs_build_fs**. Figure 3 below shows the context menu that displays when you do this. Choose **Set Breakpoint** from the context menu.

Caution: *By default, SourcePoint creates software breakpoints from most context menus. Software breakpoints may not work when page translation is enabled. It is recommended that you set the default breakpoint type to* ***Processor****. To change this setting, select* ***Preferences*** *in the* ***Options*** *menu, chose the* ***Breakpoints*** *tab, and select* ***Processor*** *in* ***Default code break type****.*

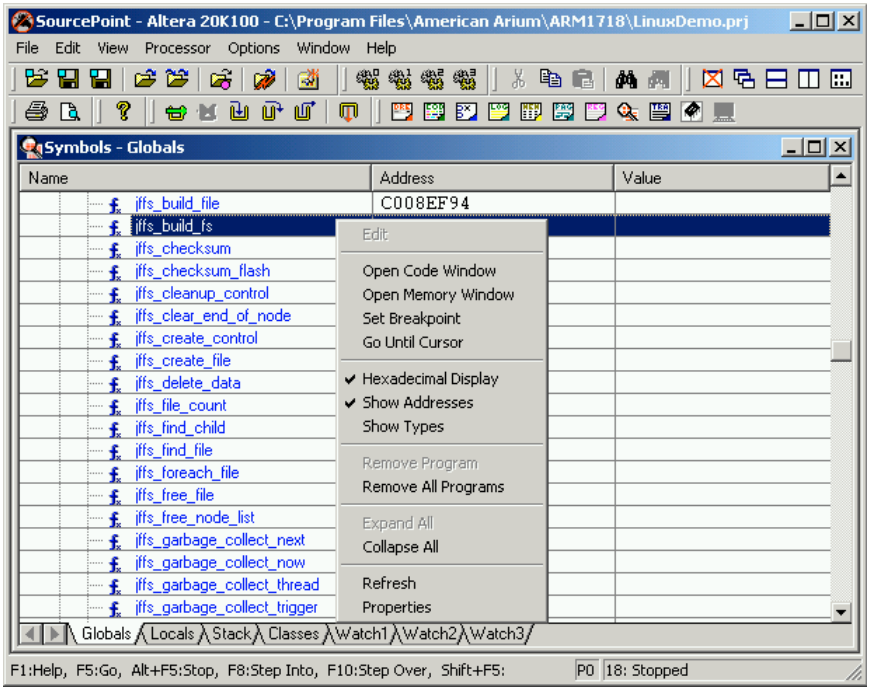

Figure 3

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. 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 4 shows the result of setting and hitting this breakpoint and also shows the contents of the **Breakpoints** window.

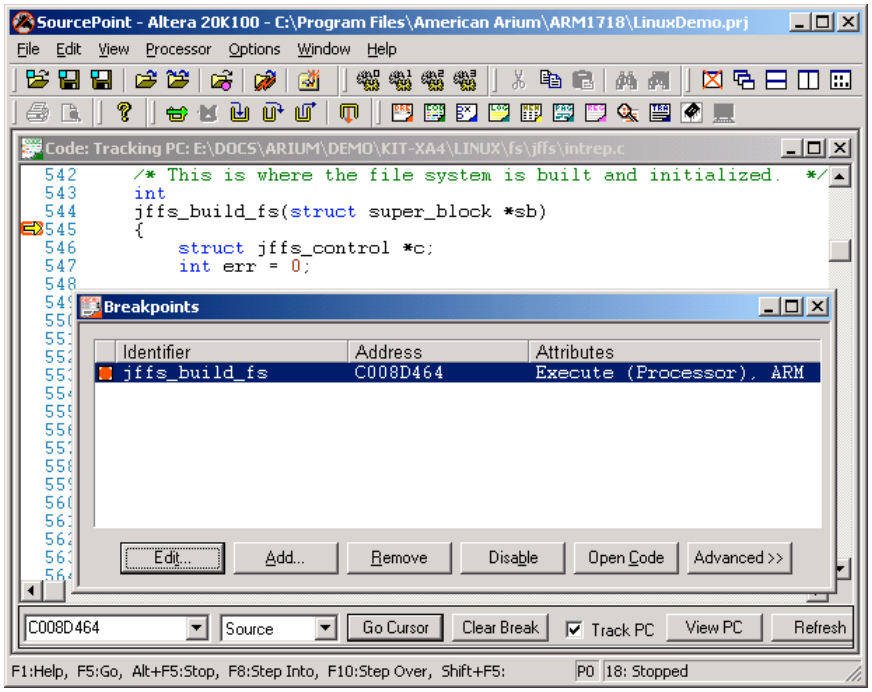

Figure 4

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 pressing 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. Pressing the F9 key will then set a breakpoint.

Alternately, you can open the **Breakpoints** 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 Target Console Window

When debugging Linux kernels that include special console code available from American Arium, SourcePoint allows you to access console input and output through the target console. The window is displayed by selecting **Target Console** from the **View** menu. All console output is displayed in the **Target Console** window. You can also pass input to the target system through this window. Figure 5 shows an example window with shell commands being executed.

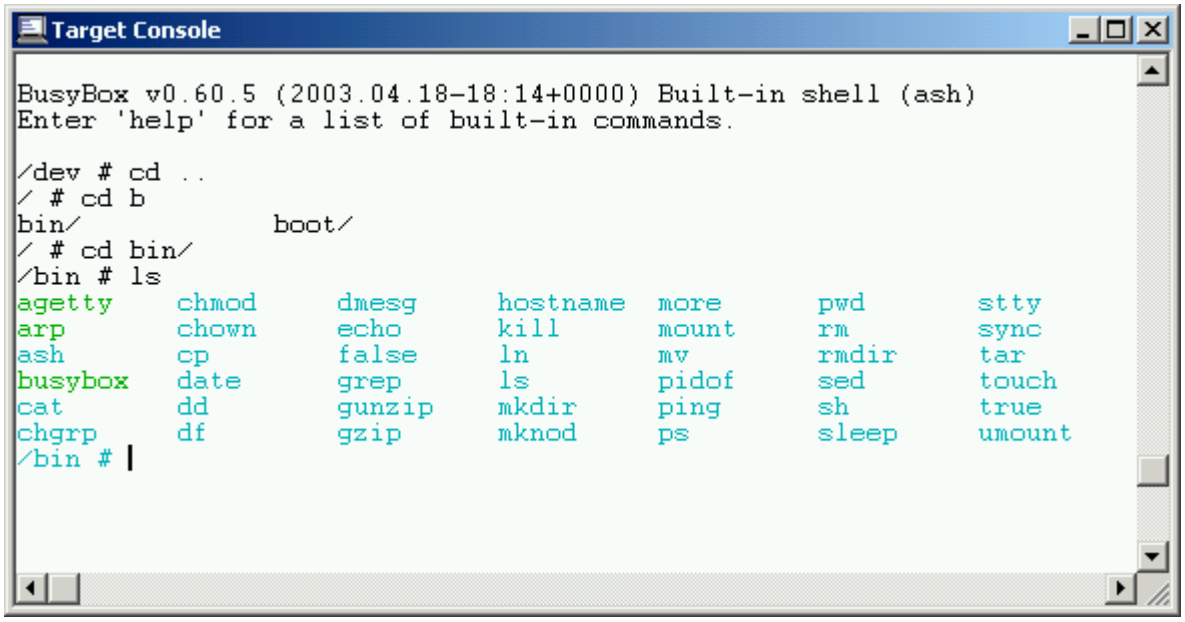

Figure 5

Using the ETM resources

As previously mentioned, the principal advantages of an ETM over a JTAG run control device are: the additional address and data comparators, the ability to capture a trace of the assembly instructions as well as the data transactions, and the ability to break or trace based on complex event sequences.

For embedded development, a potential use of bus analyzer breakpoints is to observe data being written to a serial port. To do this, simply open the **Breakpoints** window and select the **Add** button. In the **Break On** drop down box, select **Data Write**. For **Resource**, choose **ETM**. For **Location**, enter FFFC290 (the UART transmit data register for the REF-XA4), and for **Value**, choose 0A or whatever value you are looking for. Figure 6 shows the **Add Breakpoint** window for the breakpoint. The sequencer of the ETM may be used to limit the trace to only those instructions and data transactions in which you are interested. Figure 6 also shows a single state sequence that records all data written to the UART transmit data register and stopping when a 0x0A is written.

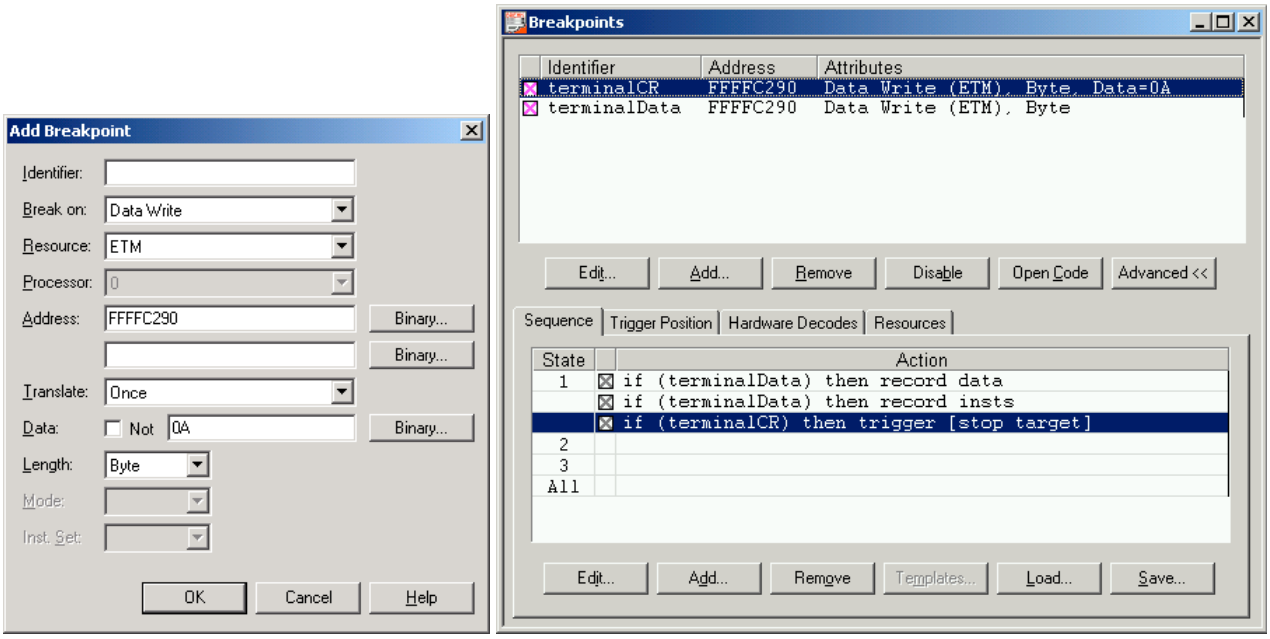

Figure 6

Displaying ETM Trace

Open the **Trace** window after hitting this trigger 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 box is probably in the lower left corner of the window, corresponding roughly to the Address text box of the **Code** window. Figure 7 shows the **Trace** window as it appears after hitting the breakpoint used in this example.

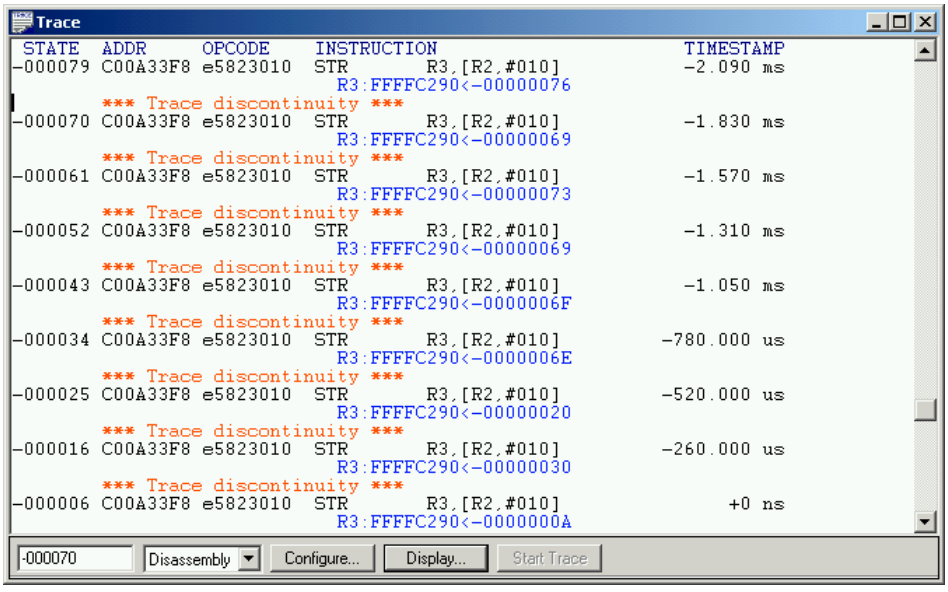

Figure 7

More Information

More information on the ARM® Embedded Trace Macrocell can be found at:

- ETM9 Rev2a Technical Reference Manual
- ETM Performance Characteristics

14811 Myford Road, Tustin, CA 92780 Voice: 877-508-3970 in the US, 949-731-1661 outside the US Fax: 714-731-6344 E-mail: info@arium.com Web www.arium.com **A0151A**