A Complete Linux Debugging Solution: Kernel, Driver, and Application Debugging in SourcePoint™

1 Overview

Note: This application note applies to SourcePoint™ version 6.3.1 and higher and Linux kernel version 2.4 or 2.6.

Arium's SourcePoint debugger offers a number of important capabilities to users who are working on Linux-based embedded systems:

- Full symbolic, source-level debugging of Linux kernel code
- Debug of Linux kernel loadable modules from the beginning of its init routine
- Source-level debugging of Linux embedded applications, including the ability to start or stop a Linux process, attach to a running process
- Symbolic shared library debug
- Thread-aware debug
- Linux console devices hosting from within SourcePoint, eliminating the need for a serial port or video device on the target and simplifying the debugging of "headless systems"

SourcePoint allows concurrent debugging of Linux kernel code and Linux application processes. Within SourcePoint, two views provide the user interface to Linux-aware debugging features. The **Operating System Resources** window lists Linux processes and serves as the primary interface for task debugging. The **Target Console** window provides multiple terminals that serve as the Linux system console and as the standard I/O device for processes launched for debugging.

SourcePoint utilizes the ARM Debug Communication Channel (DCC) hardware to communicate over the JTAG port with code on the target, eliminating the need for any serial or network hardware as a prerequisite for debugging. The only hardware dependencies are the processor core and the JTAG debug port.

To gain the OS aware features in SourcePoint, the Linux kernel must be compiled with full DWARF2 symbol information. Also, a Linux driver must be added to the kernel for DCC channel communications. A debug agent application must be compiled and added to the target file system and a patch must be applied to gdbserver to allow for Linux application debug. Finally, SourcePoint must be configured for OS-aware operation. This document will take you through all the steps to accomplish this.

2 Modes of Operation

Note: In SourcePoint, the term task is often used to refer to a Linux process. For the purpose of this documentation, the term task and Linux process should be considered synonymous.

The Linux-aware feature of SourcePoint operates in two modes: Halt mode and Task mode.

Halt mode is familiar to experienced SourcePoint users. The Stop icon causes the target processor to stop and enter debug mode. Breakpoints also cause the target processor to stop and enter debug mode. *Processor* registers and memory are accessible only when the target processor is stopped. This mode allows breakpoints to be set in the Linux kernel as well as in kernel-loadable modules.

Task mode is used for debugging Linux application processes. SourcePoint can start one or more Linux processes for debugging, or attach to existing processes for debugging. In this mode, the Stop icon stops only the process that currently has viewpoint focus in SourcePoint. The target processor is not stopped and does not enter debug mode. Other processes continue to run, as does the Linux kernel. Task-specific breakpoints can be set. Task resources are only accessible when the target processor is running and the specific task is stopped.

2.1 Debugging Context

In SourcePoint, Processors and Tasks are each associated with a debugging context which is used to access pertinent state information such as memory, registers, program symbols, and breakpoints. SourcePoint operates by default on the “focus” or Viewpoint context, which is selectable within the constraints of modes of operation. The Viewpoint window allows the user to change the “focus” or viewpoint of SourcePoint. All views displaying context-related state information show the context name in the title bar.

3 Kernel Debug

Debugging the Linux kernel with SourcePoint simply requires debug symbol information to allow the display of code and data symbols as well as source code while debugging in kernel space. Frame pointer information allows SourcePoint to unwind the call to display the routines at a higher level in the stack. It also allows SourcePoint to step out of a function to the return point in the calling function.

3.1 Configuring the Kernel for Debug

The following steps will ensure that the Linux kernel build process will include appropriate symbol and frame pointer information in the resulting **vmlinux**:

1. The following options should be set in ‘.config’:

```
CONFIG_DEBUG_INFO=y
CONFIG_FRAME_POINTER=y
```

If using ‘make menuconfig’, select the following under ‘Kernel Hacking’:

```
[*] Kernel debugging
      [*] Compile the kernel with  debug info
[*] Compile the kernel with frame pointers
```

2. If SourcePoint is not displaying complete symbol/source information, the kernel makefile(s) may need to be changed to generate DWARF2 style symbols. The standard debug information may be generated in Stabs format. SourcePoint does not support Stabs, but does support DWARF2 symbols

For Linux 2.6 kernels, change the following lines in the kernel root makefile from:

```
ifdef CONFIG_DEBUG_INFO
CFLAGS            += -g
```

To

```
ifdef CONFIG_DEBUG_INFO
CFLAGS            += -gdwarf-2
```

For Linux 2.4 kernels, change the following line in the arch/arm/Makefile from:

```
ifdef CONFIG_DEBUG_INFO
CFLAGS            += -g
```

To:

```
ifdef CONFIG_DEBUG_INFO
CFLAGS            += -gdwarf-2
```

4 Driver Debug (Loadable Kernel Modules)

To debug a Linux loadable kernel module:

1. Load Linux binary and symbols.

2. Run to start_kernel (or beyond) in the Linux kernel. This insures the proper context for the macros below. To debug automatically loaded modules, the target should be halted at the '**start_kernel**' routine.

3. Load the macro '**~/sourcepoint/Macros/Linux/kmod.mac**'

4. When any kernel module is loaded (**insmod**), SourcePoint will halt the processor on the first instruction of its init routine. On the first load of a particular module, SourcePoint will prompt you to provide the location of the symbols and source files. At this point, you can step through the module and set breakpoints based on its data and code symbols.

5. When a module is unloaded (**rmmod**), SourcePoint will remove the module's symbols. Any breakpoints set in the module will automatically be re-established when the module is reloaded.

5 Application Process Debug

5.1 Installing OS-Aware Components in Linux

When debugging application space processes on a Linux embedded target, SourcePoint uses gdbserver to initiate and monitor the application program. SourcePoint communicates to Linux and gdbserver through the Arium DCC driver, which implements multiple virtual TTY ports through the ARM processor's DCC registers. (On XScale targets, the Arium DCC driver communicates through the DBGRX and DBGTX hardware registers.) SourcePoint can use some of these ports for a Linux system console as well as additional command shells. Additional ports are used to communicate to gdbserver, once it is initiated, and to communicate to an application process being debugged.

5.1.1 Step 1: Add ariumdcc driver to kernel build

The file ariumdcc.c can be found in the SourcePoint install tree as follows:

- <SourcePoint install folder>/Samples/OSAware/Linux/Ariumdcc/2.4/ariumdcc.c
- <SourcePoint install folder>/Samples/OSAware/Linux/Ariumdcc/2.6/ariumdcc.c

The module ariumdcc.c is Arium's serial driver that uses the ARM processor's DCC register (or through DBGRX and DBGTX on XScale processors) to communicate to and from SourcePoint. The driver creates virtual serial ports that appear to Linux applications as TTY ports. The Arium DCC driver works like any other Linux serial driver and also includes support for a Linux console.

1. Copy ariumdcc.c to the "drivers/serial" directory of your Linux tree.

2. Make any necessary configuration changes to ariumdcc.c. There are several definitions that may need to be changed in ariumdcc.c. Check the following values:

• **NR_PORTS:** This is the number of ports that the driver is to support. Part of the installation process involves creating device nodes in your target system's /dev directory. By default, the Arium DCC driver supports 8 virtual ports, expecting devices defined starting with ttyDCC0 through ttyDCC7. However, greater or fewer ports can be defined with the NR_PORTS constant value, defined in the driver source. This value should be greater than or equal to the number of ttyDCC device nodes (See Section 4.2). Also, this value should be equal to the number of SourcePoint Linux consoles plus two times the maximum number of tasks plus two. (See Section 5.1, Target Configuration, Channels.)

NR_PORTS >= # of Device Nodes >= Consoles + 2*Max Tasks + 2{dccwrap channels}

• **ARIUMDCC_MINOR**: This should match first minor number of first device node - ttyDCC0 (72 by default).

• **CONFIG_ARIUMDCC_CONSOLE**: By default, the Arium DCC driver includes console support. This is the recommended configuration. However, if it is desirable to disable console support (say, if you are already using a serial port as the console), the value **CONFIG_ARIUMDCC_CONSOLE** can be undefined in the driver source, either by commenting the #define statement out or changing the #define statement to a #undef statement.

Note: If console support is disabled, then the number of consoles declared in SourcePoint should be zero (0). See section 5.2.1.

3. Add the following entry to "drivers/serial/Makefile"

obj-y += ariumdcc.o

4. Rebuild kernel.

5.1.2 Step 2: Define device nodes in /dev

Important! : Skip this step if using 'devfs' for automatic device node creation on your target system.

One of the virtual ports is used to send commands to a Linux shell either to get information about currently running processes (OS Aware Task view), or to start an instance of gdbserver to be able to debug an application process. Although one port is dedicated to shell commands sent automatically by SourcePoint, it might be desirable to create one or more consoles on one or more ports. By default, SourcePoint expects at least three ports to be defined. The first two ports, /dev/ttyDCC0 and /dev/ttyDCC1, can be used as a Console ports. The third port, /dev/ttyDCC2, is used as the port through which SourcePoint sends commands to be executed by the shell (the initiator port). Additional ports should also be defined for use by gdbserver and by any application that may be debugged by gdbserver.

Go to the /dev directory **of the target file system** and execute the following commands for the number of desired ports. The total number of device nodes should be equal to **NR_PORTS** (8 by default). The minor numbers should start with the minor number configured in the driver (**ARIUMDCC_MINOR** is 72, by default):

mknod -m 644 /dev/ttyDCC0 c 4 72
mknod -m 644 /dev/ttyDCC1 c 4 73
mknod -m 644 /dev/ttyDCC2 c 4 74
mknod -m 644 /dev/ttyDCC3 c 4 75
mknod -m 644 /dev/ttyDCC4 c 4 76
mknod -m 644 /dev/ttyDCC5 c 4 77
mknod -m 644 /dev/ttyDCC6 c 4 78
mknod -m 644 /dev/ttyDCC7 c 4 79
…

5.1.3 Step 3: Add entries to inittab

As mentioned earlier, one of the virtual ports is used to send commands to a Linux shell either to get information about processes currently running (OS Aware Task view), or to start an instance of gdbserver to debug an application process. A Linux shell program must be running on that virtual port to execute the commands sent from SourcePoint. To start a shell program running on a virtual port, the inittab file should be modified to start a shell for the virtual port.

The inittab file is used by the init program to start processes, such as getty, during system boot and at the start of normal operation. Generally, the inittab file defines a getty for each TTY device that allows system login. Review your current Linux documentation (such as man pages) on how to update your inittab file.

To allow shell commands to be processed on a port, your target's inittab file must be modified. Add entries to start a shell for each of the virtual ports used as consoles (2 by default, ttyDCC0 and ttyDCC1) and also for the port through which SourcePoint will send commands (the initiator channel, by default, ttyDCC2). Adding more than needed will cause problems when trying to debug Linux tasks.

In short, you should define the Consoles setting of the SourcePoint "Target Configuration/Operating System/Linux" to one less than the number of ttyDCC devices initialized in inittab.
Example {add more description}.

#::respawn:/sbin/getty -L -n -l /bin/sh 38400 /dev/ttyDCC0
#::respawn:/sbin/getty -L -n -l /bin/sh 38400 /dev/ttyDCC1
#::respawn:/sbin/getty -L -n -l /bin/sh 38400 /dev/ttyDCC2

Or…

/dev/ttyDCC0::respawn:/bin/sh
/dev/ttyDCC1::respawn:/bin/sh
/dev/ttyDCC2::respawn:/bin/sh

5.1.4 Step 4: Build SourcePoint OS-Aware agent (dccwrap)

The dccwrap program is used to assist SourcePoint in retrieving Linux OS information as well as launching gdbserver to debug a task. When SourcePoint starts to debug a task, it sends a shell command to start gdbserver via the initiator channel.

5.1.4.1 Procedure to build dccwrap

The dccwrap agent source files can be found in the SourcePoint install tree as follows:

<SourcePoint install folder>/Samples/OSAware/Linux/dccwrap/dccwrap.tgz

1. Untar the dccwrap.tar.gz file. This creates a directory with the dccwrap components in it:
   tar -zxvf dccwrap.sp6.1.tar.gz
2. Change the working directory to dccwrap directory: "cd dccwrap"
3. Run make in the dccwrap directory to build dccwrap: "make"
4. Copy the dccwrap program to your target's /home directory.

5.1.5 Step 5: Building gdbserver

SourcePoint uses a patched version of gdbserver that supports breakpoints. Arium supplies the patch for two versions of gdbserver.

The file gdbserver files can be found in the SourcePoint install tree as follows:

- <SourcePoint install folder>/Samples/OSAware/Linux/gdbserver

5.1.5.1 Procedure for building gdbserver (gdb-6.0)

For patch gdbserver\gdb-6.0-patch-aa1.pat:

1. Download GNU GDB 6.0 from the GNU website:
   wget ftp://sources.redhat.com/pub/gdb/old-releases/gdb-6.0.tar.gz
2. Untar gdb-6.0.tar.gz:
   tar -zxvf gdb-6.0.tar.gz
3. Apply patch to GDB directory:
   patch -d gdb-6.0 -p1 < gdb-6.0-aa2.patch
4. Make gdbserver's configuration script executable:
   chmod +x gdb-6.0/gdb/gdbserver/configure
5. Create build directory at the same level as gdb-6.0 directory:
   mkdir gdb-6.0-build
   cd gdb-6.0-build
6. Run the configure script. Set CC variable as you run it:
   CC=arm-linux-gcc ../gdb-6.0/gdb/gdbserver/configure --host=arm-linux
7. Run make in the build directory to make gdbserver
8. Strip symbols out of gdbserver to make the program smaller:
   arm-linux-strip gdbserver

5.1.5.2 Procedure for building gdbserver (gdb-6.4)

For patch gdbserver\gdb-6.4-aa1.patch:

1. Download GNU GDB 6.4 from the GNU website:
   wget ftp://sources.redhat.com/pub/gdb/old-releases/gdb-6.4.tar.gz
2. Untar gdb-6.4.tar.gz:
   tar -zxvf gdb-6.4.tar.gz
3. Apply patch to GDB directory:
   patch -d gdb-6.4 -p1 < gdb-6.4-aa1.patch
4. Make gdbserver's configuration script executable:
   chmod +x gdb-6.4/gdb/gdbserver/configure

5. Create build directory at the same level as gdb-6.4 directory:
   mkdir gdb-6.4-build
   cd gdb-6.4-build
6. Run the configure script. Set CC variable as you run it:
   CC=arm-linux-gcc ../gdb-6.4/gdb/gdbserver/configure --host=arm-linux
7. Run make in the build directory to make gdbserver
8. Strip symbols out of gdbserver to make the program smaller:
   arm-linux-strip gdbserver

5.1.5.3 Procedure for building gdbserver (gdb-6.6)

For patch gdbserver\gdb-6.6-aa1.patch:

1. Download GNU GDB 6.6 from the GNU website:
   wget ftp://sources.redhat.com/pub/gdb/old-releases/gdb-6.6.tar.gz
2. Untar gdb-6.6.tar.gz:
   tar -zxvf gdb-6.6.tar.gz
3. Apply patch to GDB directory:
   patch -d gdb-6.6 -p1 < gdb-6.6-aa1.patch
4. Make gdbserver's configuration script executable:
   chmod +x gdb-6.6/gdb/gdbserver/configure
5. Create build directory at the same level as gdb-6.6 directory:
   mkdir gdb-6.6-build
   cd gdb-6.6-build
6. Run the configure script. Set CC variable as you run it:
   CC=arm-linux-gcc ../gdb-6.6/gdb/gdbserver/configure --host=arm-linux
7. Run make in the build directory to make gdbserver
8. Strip symbols out of gdbserver to make the program smaller:
   arm-linux-strip gdbserver

After the following the procedure above, gdbserver can be moved to your target's /home directory. For additional information about gdbserver, see the GNU GDB web page: http://www.gnu.org/software/gdb.

5.1.6 Other Requirements

In order to view source code and stack trace when debugging in shared library code, SourcePoint requires copies of the shared library binaries that include DWARF2 debug symbols. Since rebuilding the toolchain and libraries can be very complex, Arium recommends that the initial build of the toolchain include these symbols. The binaries can be copied and stripped at a later time for inclusion on the target file system.

Note: Threaded application debug requires the use of gdbserver from the gdb-6.4 version or later.

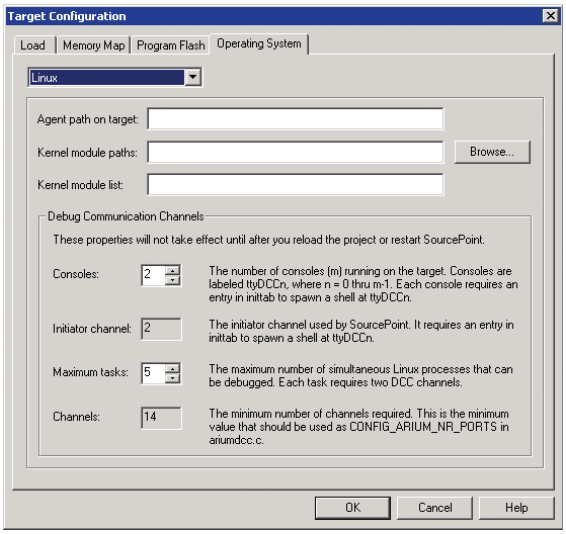
5.2 Linux OS Aware Windows in SourcePoint

There are three windows associated with Linux debug: Target Configuration (Operating System tab), Target Console, and Operating System Resources. They are described in more detail below.

5.2.1 Target Configuration (Operating System Tab)

Select Options|Target Configuration to access the Operating System tab. The options in the tab help to optimize communications between a Linux target and the debugger.

A Complete Linux Debugging Solution: Kernel, Driver, and Application Debugging in SourcePoint™



Operating System *tab under* ***Options\Target Configuration***

Agent path on target: Specifies the full path (in the target file system) to the directory containing the on-target debugging agent (dccwrap and gdbserver). If this path information is included in the normal search path on the target ($path environment variable), then this field can be left blank.

Kernel module paths: Specifies, for kernel modules, the list of directories on the host system to be searched when symbols are loaded. Multiple paths are delimited with commas. This list is appended automatically whenever you are prompted to browse for symbols during loading of kernel modules. The user can also edit the path if he changes his build paths.

Kernel module list: A comma-delimited list showing which kernel modules you wish to debug. If this list is blank, then all kernel modules will trigger debugging (halt and symbol load). See “To Begin Linux Loadable Kernel Module Debug”.

Consoles: Specifies the number of static system consoles to be configured on the target.

Note: As mentioned earlier, by default, SourcePoint expects at least two ports to be defined, ttyDCC0 and ttyDCC1. The first port, ttyDCC0, can be used as a console port. The second port, ttyDCC1, is used as the port through which SourcePoint sends commands to be executed by the shell.

Initiator channel: Specifies which DCC channel will be used to start the SourcePoint Linux debugging agent programs (dccwrap and gdbserver).

Maximum tasks: Specifies the maximum number of Linux processes that you can debug simultaneously.

Channels: Specifies the number of channels required to support the values set in Consoles and Maximum tasks. The value of **NR_PORTS** ariumdcc.c should be equal to or greater than this value. This is also the minimum number of ttyDCC device nodes that must be defined.

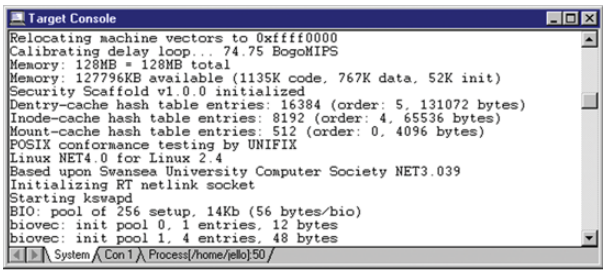
NR_PORTS >= # of Device Nodes >= Consoles + 2*Max Tasks + 2 {2=dccwrap channels}

5.2.2 Target Console Window

Select View|Operating System|Target Console on the menu bar to access the Target Console window or click on the Target Console icon on the toolbar.

Note: To open a Target Console window, you must first enable the OS-aware features described above.

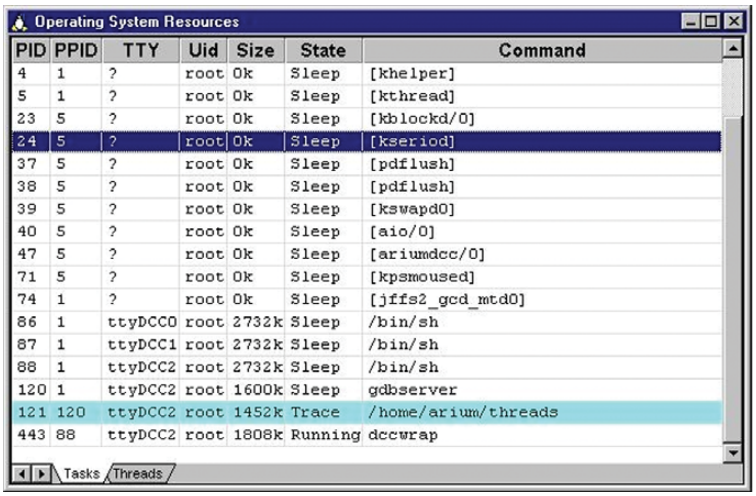
The Target Console window contains tabbed views for the Linux console along with each process under debug on the target. Each view implements an ANSI VT100 serial terminal display/keyboard device.



Target Console *window*

5.2.3 Operating System Resources Window

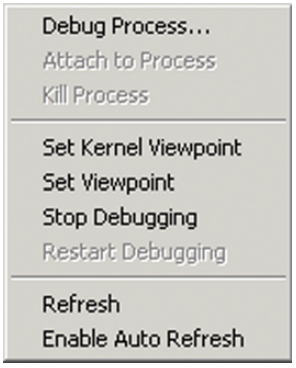
To open an Operating System Resources window, select View|Operating System|Resources. The Operating System Resources window displays the tasks running under Linux. Tasks being debugged by SourcePoint are denoted by a light blue background.



After a task is launched, it is added to the ***Operating System Resources*** *window* ***Task*** *List*

5.3 Task Context Menu

To display the context menu for a task, select a process in the Operating System Resources window and right-click. The menu allows you to select a task from a list and attach to it or launch a new process under debug.



Linux task *context menu*

Debug Process command. Use this command to initiate a program in a new process for debugging.

Attach to Process command. Use this command to intercept the selected process for debugging.

Kill Process command. This command sends a **SIG_TERM** signal to the selected process. (Not functional at this time.)

Set Kernel Viewpoint command. This command causes SourcePoint to switch focus to the *Processor* context and enter **Halt** mode.

Set Viewpoint command. This command causes SourcePoint to switch focus to the *Task* context for the selected task.

Stop Debugging command. This command abandons debugging on the selected task. If debugging was initiated by the **Debug Process** command, the process ends. If debugging was initiated by the Attach to Process command, the process continues to run.

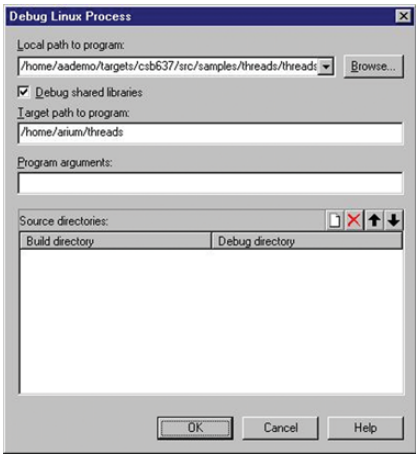
Restart Debugging command. This command causes the program under debug to be restarted. (Not functional at this time.)

Refresh. This command refreshes contents of the Operating System Resources window.

Enable/Disable Auto Refresh. This command toggles the timed refresh of the Operating System Resources window.

5.4 To Begin Linux Task Debug

To debug a Linux program, right-click in the in the Operating System Resources window to summon the context menu. From the menu select Debug Process. This displays the Debug Linux Process dialog box.



Debug Linux Process *dialog box*

Local path to program: This field specifies the full path on the host system (running SourcePoint) to the executable with debugging symbols. Either select an entry from the combo dropdown list or use the Browse button to navigate to the desired program.

Debug shared libraries: This check box will tell SourcePoint to load symbols for all dynamic libraries that are used by this processes. If you want source level debug, then a copy of the libraries with full DWARF2 debug symbols must be available. This option must be checked to debug multi-threaded applications.

Target path to program: This field specifies the full path on the target system to the executable to be debugged. The value in this field is used to launch the process executable on the target. It is disabled for Attach operations.

Program arguments: This field is used to specify any command line arguments to be supplied to the new process. It is disabled for Attach operations.

Source directories: This list is a map relating the directories in the symbols from the Build system top to the corresponding directories on the local (Debug) system. It is used for the purpose of locating source files. If the Build and Debug paths are identical, leave the list blank. Otherwise, specify only the leftmost part of the path that varies.

Example: /home/fred/linux-42 -> c:\linux-42

This map is also populated automatically by SourcePoint whenever you are asked to locate source.

Press the OK button to start the process.

5.5 Debugging Under Linux

The primary goal of the SourcePoint Linux-aware feature is to provide concurrent source-level debugging support for the Linux kernel and application processes, using the same user interfaces with the capability to transition seamlessly between the two modes.

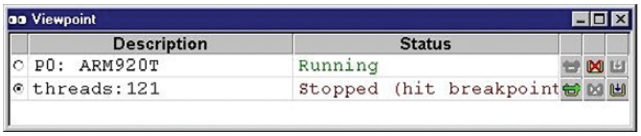
When Task mode debugging is initiated by Debug Process, SourcePoint switches the focus context to the new task, which is then stopped at its entry point. The title bars for all views tracking the focus context display the program name and Linux process ID (PID). When Task mode debugging is initiated by Attach to Process, the selected task is stopped at the point of intercept, which is often at the return from a system call.

Once Task mode is entered, SourcePoint behavior for most views is essentially the same as Halt Mode. The most notable difference is in the lifetimes of Task and Processor contexts. Processors are perpetual while Tasks are transient. When a Task ends, its context is destroyed, and SourcePoint switches its focus to another Task. When the last Task context exits, SourcePoint switches to the kernel (Processor) context and enters Halt mode running. When the kernel hits a breakpoint in, say, a device driver or system call, SourcePoint switches to the kernel context and enters Halt mode. The usual run controls then can be used to debug the kernel. To resume Task debugging, use the Go command, then select the desired task in the System Resources Task list and use its context menu Set Viewpoint command to switch to the task.

Open a Code view. From there you can go and step and set task breakpoints. You can also set breakpoints from the standard SourcePoint Breakpoints window.

5.5.1 Switching between Halt and Task mode debugging

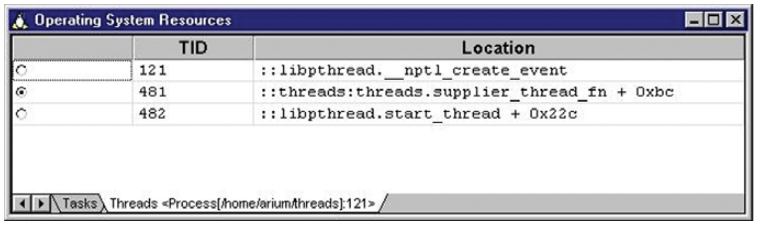
SourcePoint allows the user to switch between Halt mode and Task mode debugging. The Viewpoint window can be used to switch the debugging mode from the processor (Halt mode) to one of the running processes (Task Mode). As previously described, when debugging in Task mode, the processor must be running, so when switching from Halt mode to Task mode, you must start/run the processor.



Viewpoint *window*

5.6 Debugging Multi-Threaded Applications

When debugging a multi-threaded application, SourcePoint will detect when threads are created and destroyed. SourcePoint will display all current threads in the Threads tab of the Operating System Resources window. The currently selected thread controls the context of the Registers and Code windows. When the process is stopped by a breakpoint, the thread that triggered the breakpoint will be selected in the Threads tab as the current thread.



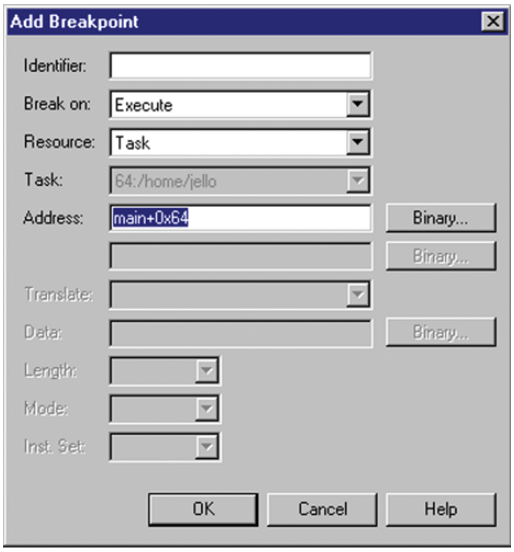
Operating System Resources *window –* ***Threads*** *tab*

5.2 Linux OS Aware Windows in SourcePoint

SourcePoint includes a new class of breakpoints known as task breakpoints. As the name indicates, this break type applies to a specific task and is available only when you are in Task mode.

To set a task breakpoint from the Breakpoints window:

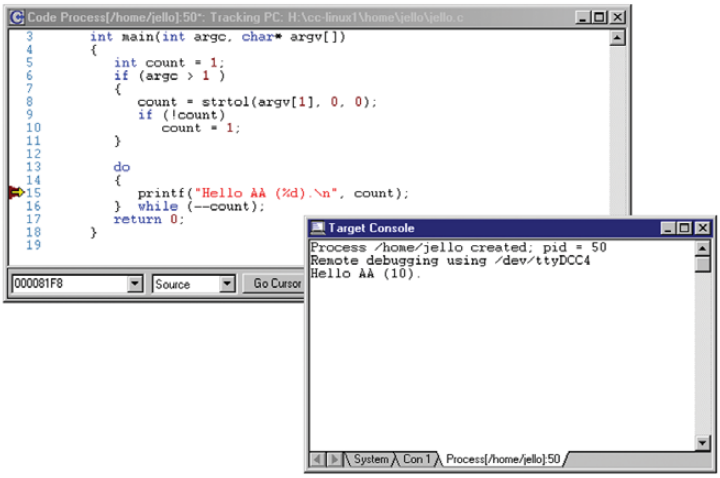
1. Select View|Breakpoints. The Breakpoints window displays.
2. Click on the Add button. The Add Breakpoint dialog box displays.
3. From the Identifier dialog box, specify an identifier for a breakpoint. If no value is entered, a default identifier (event# where # is some number) is used.
4. From the Break On drop down box, select Execute.
5. From the Resource drop down box, select Task.
6. In the Address text box, key in the location of the task breakpoint you want to add.



Add Breakpoint *dialog box*

The Breakpoints window re-displays with the breakpoint listed in the Breakpoint list box.

Note: A target reset causes SourcePoint to switch from Task mode to Halt mode. However, all breakpoints previously set still display in the Breakpoint list box, including task breakpoints. These task breakpoints become active again if and when the applicable Linux process becomes the current process again.



Code Process *window showing breakpoints executed under Linux as indicated in the* ***Target Console*** *window*

6 Summary

SourcePoint allows concurrent debugging of Linux kernel code and Linux application processes. Within SourcePoint, two new views provide the user interface to Linux-aware debugging features. The Operating System Resources window lists Linux processes and serves as the primary interface for task debugging. The Target Console window emulates multiple terminals which serve as the Linux system console and as the standard input and output device for processes launched for debugging. Together, they allow the user to do full symbolic, source-level debugging of the kernel and a process seamlessly, moving back and forth between the two with ease.

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