Ashling Product Brief APB207
v.1.0.6, 10th January 2013

PathFinder-XD for MIPS™ Powered Devices

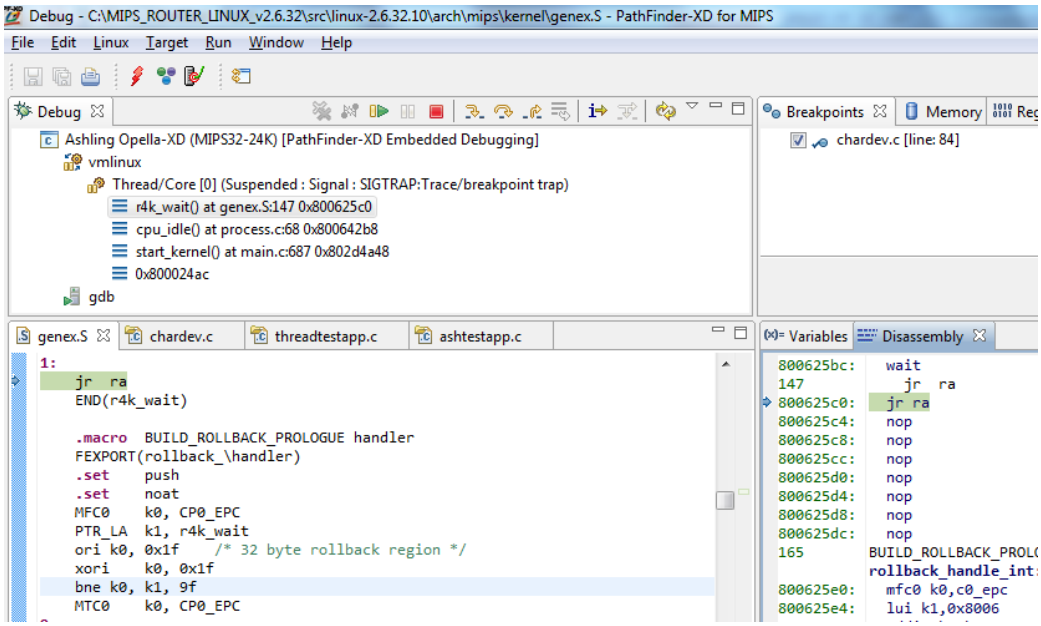


Figure 1. PathFinder-XD for MIPS™

Contents

1. Introduction

This Ashling Product Brief introduces Ashling's PathFinder-XD for MIPS™ Debugger. PathFinder-XD is a C/C++/Assembly debugger based on the Eclipse framework and supports debugging using the QEMU software simulator (www.qemu.org) or the Ashling Opella-XD Debug Probe connected to MIPS™ powered target hardware.

PathFinder-XD supports both "bare-metal" (no target operating system) and Embedded Linux based debugging. This application note introduces PathFinder-XD and covers:

- PathFinder-XD installation
- Using PathFinder-XD with the Opella-XD Debug Probe
- Embedded Linux Debugging with PathFinder-XD and Opella-XD

The last two sections use the Ubiquiti Networks RouterStation Pro (powered by an Atheros AR7161 MIPS24K based SoC) as the target system. This can be purchased from MIPS for approximately $150 dollars. See here for details: http://www.mips.com/products/development-kits/linux-starter-kit/

For details on using PathFinder-XD with the QEMU simulator Please refer to the separate document APB211 which is supplied with PathFinder-XD and also available from the Ashling website at http://www.ashling.com/images/stories/pdfs/technicalarticles/APB211-PF-XD_MIPS_SIM.PDF

2. Installation

PathFinder-XD can be hosted under Windows™ or x86 based Linux and installation requires full administration privileges.

2.1 Windows™ Installation

Run the `SETUP.EXE` program from the Windows directory on the supplied CD (or download) and follow the on-screen instructions.

2.2 Linux Installation

Run the `./SETUP32` (32-bit Linux) or `./SETUP64` (64-bit Linux) program from the supplied CD (or download) and follow the on-screen instructions. PathFinder-XD for MIPS is tested on the following Linux platforms:

- Fedora 13/Ubuntu 10.04 LTS 32-bit/64-bit versions

Please note that the 64-bit Linux version of PathFinder-XD for MIPS requires the 32-bit library package `ia32-libs` library, hence, make sure this is installed in your system. For example, to install on Ubuntu/Debian, issue the following command:

```
> $sudo apt-get install ia32-libs
```

3. Using PathFinder-XD with the Opella-XD Debug Probe

The Ashling Opella-XD as shown below is an entry-level Debug Probe for the MIPS family which connects to the host PC via a USB2.0 interface.

Figure 2. The Ashling Opella-XD Debug Probe

Opella-XD uses the MIPS EJTAG core extension to provide a comprehensive set of debug features. PathFinder-XD is Ashling's software interface for the Ashling Opella-XD.

3.1 Opella-XD USB Driver Installation

3.1.1 Windows™ USB Driver Installation

When you first connect Opella-XD to your PC you will get a **New USB hardware found** message and will be prompted to install the appropriate USB drivers. The Ashling Opella-XD drivers are supplied on your Ashling CD and installed in your installation directory. Direct the Windows **Hardware Installation Wizard** to your installation directory so that it can locate the necessary drivers and complete the installation. Windows only needs to perform this operation the first time you connect your Opella-XD to the PC. The Opella-XD USB driver is called `libusb0.sys (libusb0_x64.sys` for 64-bit operating systems).

3.1.2 Linux x86 USB Driver Installation

Opella-XD uses the `libusb-0.1` driver (http://www.libusb.org/). By default, the driver is stored in `/lib` directory (in 32bit Ubuntu 12.04 it is located at `/lib/i386-linux-gnu` directory)

Check for this as follows:

```
$ ls /lib/libusb*
```

If you see `libusb-0.1.so.4.4.0` or higher then they are installed on your system and you can skip the next section on libusb installation

Please note:

1. If your `/lib` directory does not include a file titled `libusb.so` (exact filename) then create a symlink as follows:

```
$sudo ln -s /lib/libusb-0.1.so.4.4.0 /lib/libusb.so
```

3.1.2.1 Ubuntu/Debian libusb installation

Install `libusb-0.1` using the following command:

```
$ sudo apt-get install libusb-dev
```

If your `/lib` directory does not include a file titled `libusb.so` (exact filename) then create a symlink as follows:

```
$sudo ln -s /lib/libusb-0.1.so.4.4.0 /lib/libusb.so
```

3.1.2.2 Fedora libusb installation

Install libusb-0.1 using the following command running as super user:

```
# yum install libusb
```

If your /lib directory does not include a file titled libusb.so (exact filename) then create a symlink as follows:

```
#ln -s /lib/libusb-0.1.so.4.4.0 /lib/libusb.so
```

3.1.2.3 Using libusb on other installations

Download the latest `libusb-0.1` from http://www.libusb.org and install as follows:

```
$ tar zxf libusb-0.1.12.tar.gz (use appropriate version number)
$ ./configure --prefix=/usr
$ make
$ make install
```

If your `/usr/lib` directory does not include a file titled `libusb.so` (exact filename) then create a symlink as follows:

```
$ln -s /usr/lib/libusb-0.1.so.4.4.0 /usr/lib/libusb.so
```

3.1.2.4 Setting permissions

1. Ensure that Opella-XD is connected to the PC, connected to the target and that the target is powered
2. To ensure the current `$USER` has access to the Opella-XD device we recommend using the Linux utility `udev` (requires kernel 2.6 or later).
3. Ensure `udev` is installed and running on your system by checking for the `udev` daemon process (`udevd`) e.g.:
   ```
   $ ps -aef | grep udev
   ```
4. Create an `udev` rules file to uniquely identify the Opella-XD device and set permissions as required by owner/ groups. An example `udev` file is supplied (`60-ashling.rules`) which identifies Opella-XD device (by Ashling's USB product ID and Vendor ID).
5. The rules `file` must then be copied into the rules directory (requires root permission) e.g.:
   ```
   $ sudo cp ./60-ashling.rules /etc/udev/rules.d
   ```

3.2 Debugging with PathFinder-XD

In this section we will look at using PathFinder-XD and Opella-XD with an Ubiquiti Networks RouterStation Pro board which uses the Atheros AR7161 MIPS24K powered device. Ensure your setup is configured as shown below:

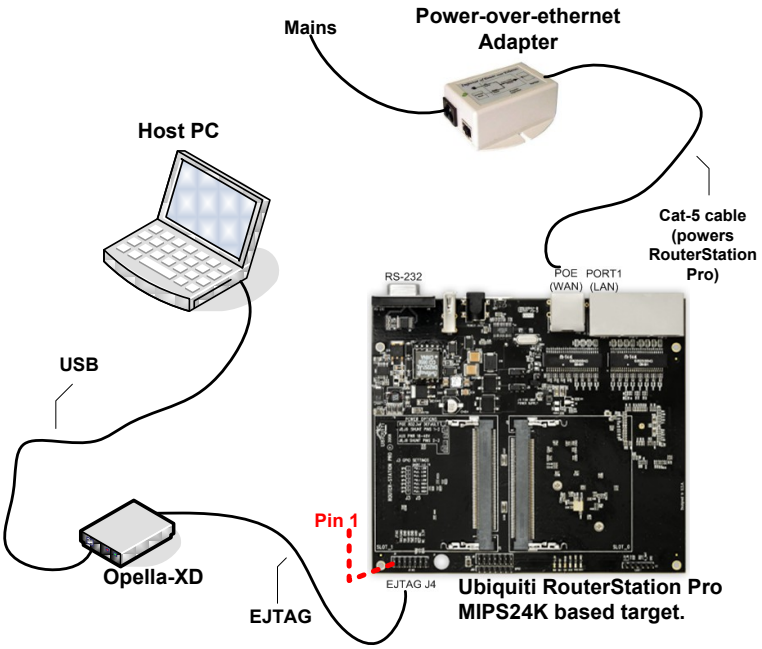


Figure 3. RouterStation Pro Debugging with Opella-XD

3.2.1 Connecting Opella-XD to the Target

Opella-XD is designed to connect to your PC via the USB cable and your target via the supplied EJTAG cable. Pin 1 of the Ashling EJTAG Cable Connector is clearly identified by a ⇧ on the connector; this should mate with pin 1 on your target's EJTAG connector. Please note the following recommended target connection sequence:

1. Ensure your target is powered off.
2. Connect Opella-XD to your PC using the supplied USB cable and ensure Opella-XD's **Power** LED is on.
3. Connect Opella-XD to your target using the supplied EJTAG cable.
4. Power up your target.

3.2.2 Using PathFinder

1. To get started, run PathFinder-XD 

. If this is your first-time running then you will be prompted to specify your Workspace (default directory for projects etc). Accept the default which is located in PathFinder-XD's installation directory.

2. In PathFinder-XD, create a **New Target Configuration** via the **Target** menu

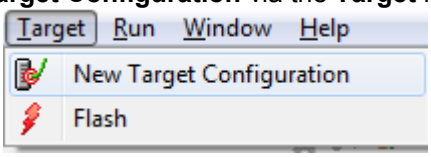


Figure 4. Target Configuration

and select the **Debug using Debug Probe** option as shown below

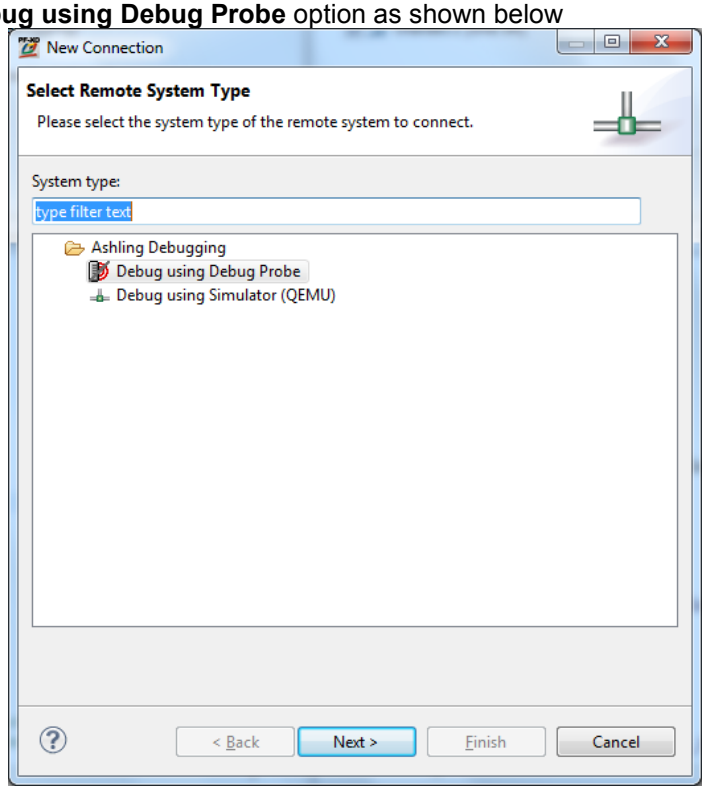


Figure 5. Debug using Debug Probe

3. Click **Next** and we can now configure our Opella-XD settings as shown below:

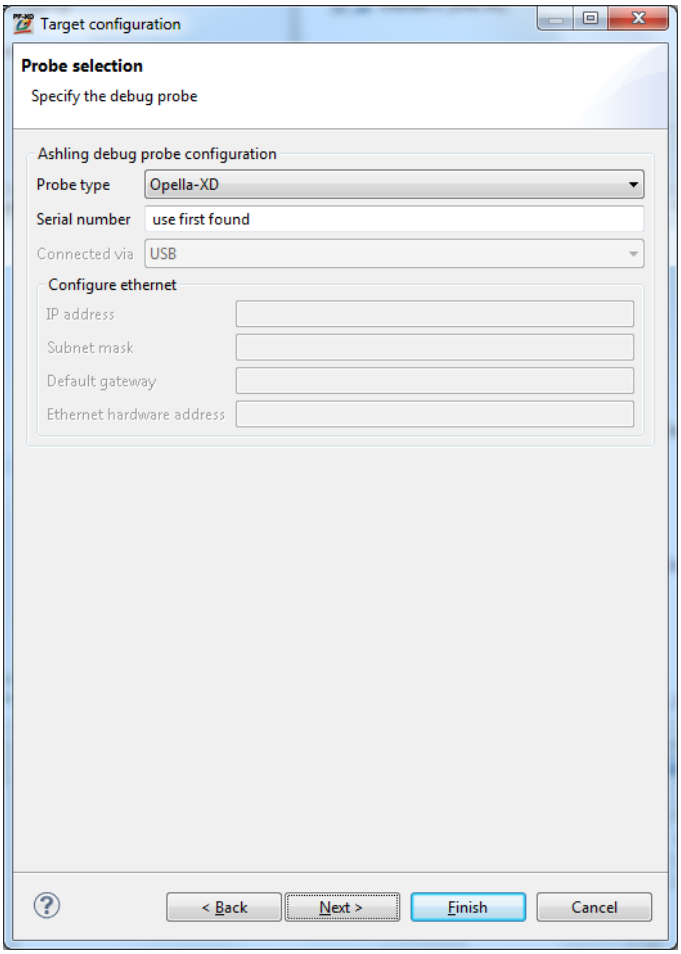


Figure 6. Probe selection

Settings include:

- **Probe type:** The actual Ashling Debug Probe Type to use as the target connection. Select **Opella-XD**
- **Serial number:** The serial number of the Debug Probe to use. Specify the serial number or **use first found** and click on **Next**

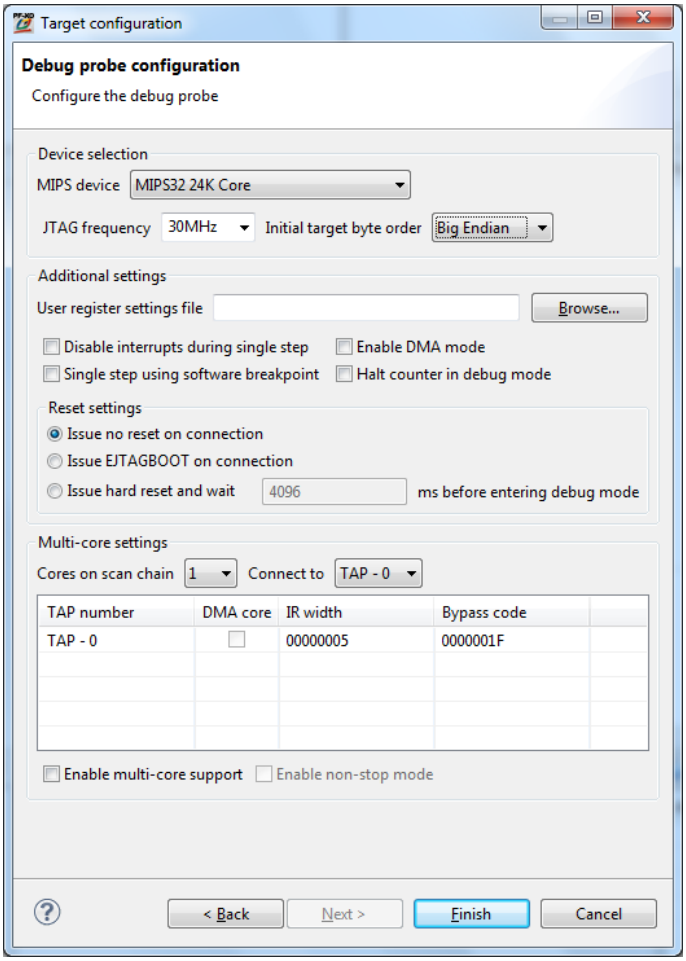


Figure 7. Debug probe configuration

The **Debug probe configuration** settings include:

- **MIPS device:** specifies the MIPS device type you wish to debug. In this example, Broadcom BMIPS5000 is selected.
- **JTAG frequency:** specifies the JTAG TCK frequency to be used for communicating with the EJTAG interface on your MIPS device.
- **Initial target byte order:** allows you to specify the memory Endianess of your target system.
- **User register settings file:** group allows you to initialise other registers or memory locations on PathFinder-XD invocation and after reset. The **Browse…** button allows these register values to be loaded from a simple text file. The text file format is:
  ```
  Name  Size Address Value
  ```
  (all values are in HEX). For example, the following text file initialises the R0, R1, R2 and R3 registers:
  ```
  R0 0x00000004 0xb800380c 0x18000000
  R1 0x00000004 0xb8003808 0x00000006
  R2 0x00000004 0xb8004018 0x00000800
  R3 0x00000004 0xb800401c 0x0000000c
  ```
- **Disable interrupts during single step:** allows you to disable interrupts when single stepping at assembly level (MIPS instruction level). When checked, PathFinder-XD automatically disables interrupts prior to an assembly level single step and re-enables them after the single step is complete.
- **Enable DMA Mode:** enables DMA mode for high-speed transfer between the debug probe and your target. DMA Mode is only available on systems with EJTAG DMA support.
- **Single step using software breakpoint:** allows you to specify that PathFinder-XD should use software breakpoints for single-stepping (i.e. PathFinder-XD should not use the EJTAG hardware based single-step command).
- **Halt counters in debug mode:** allows you tell PathFinder-XD to halt the MIPS Count register(s) (via writing to the Configuration register) whenever your program is halted. There is a slight delay between your program halting and the write to the Configuration register. Note that the **Registers** window always shows your application values for the Configuration register.
- **Issue no reset on connection**: will ensure no hardware reset is issued when you connect to your target (note that this feature requires updated Opella-XD firmware (v1.1.1 or later) which is supplied with PathFinder-XD v1.0.6 or later).

- **Issue EJTAGBOOT on connection**: will issue a hardware reset and halt the target at the reset location.
- **Issue hard reset and wait 'N' ms before entering debug mode**: will issue a hardware reset and wait the specified number of ms before entering debug mode. This mode is also known as NORMALBOOT.
- **Multi-core:** allows you to select the core you wish to debug for multi-core devices.

The settings shown are suitable for an Ubiquiti RouterStation Pro target board. Click Finish when done.

4. PathFinder-XD will now create a new **Target Debugger** setting in its **Remote Systems** Window as shown below:

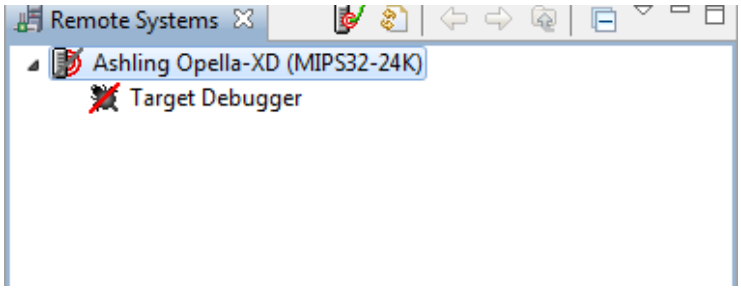


Figure 8. Remote Systems Window

Right-click on **Target Debugger** and click **Connect to** invoke the Opella-XD target connection. Once invoked, the **Remote Systems** window will update as follows:

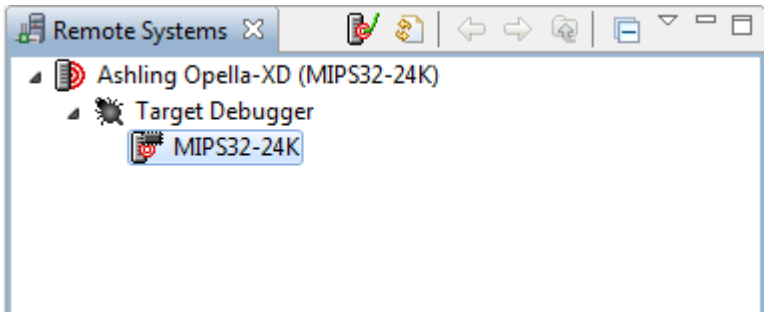


Figure 9. Remote Systems Window showing target connection

5. We can now download a program to the target by right-clicking over **mips32-24k** and selecting **Download and Launch** as follows**:**

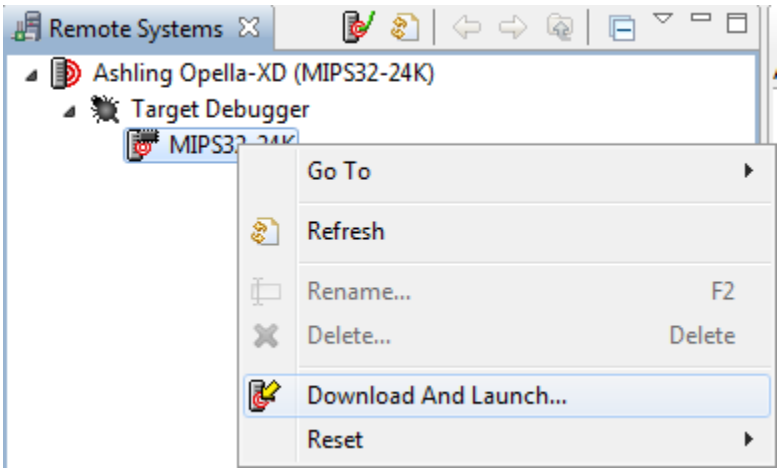


Figure 10. Download and launch

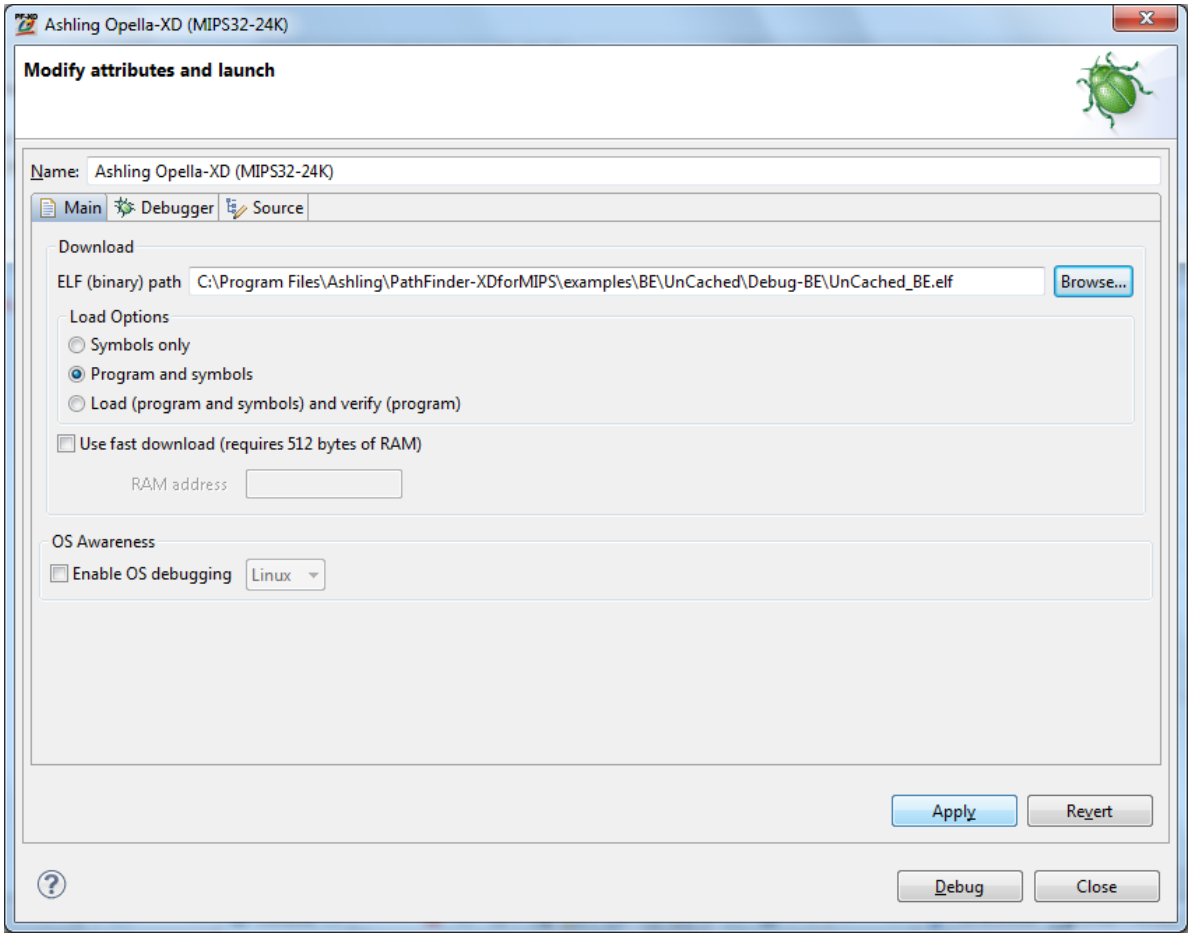


Figure 11. Specifying Target Program to Download

Specify the program to use (**ELF (binary) path**) and press **Debug** to download to the target board. Note:

- PathFinder-XD supports ELF format files which should be compiled/linked with debug information. For example, when using the GNU tool-chain add the compiler gcc switch "`-g`" (generate debug symbols) when compiling all files you wish to be able to debug. Compiler optimisations should not be used as they can cause misalignment between the generated symbolic information and the actual generated machine code thus causing problems when debugging.
- When debugging existing flash based code you should select **Symbols only**. This ensures no code is downloaded to your target system (it is already there in flash) and that PathFinder-XD just extracts the source-file and symbol information from the specified ELF file.
- When downloading (program and symbols) you can verify that target memory matches the original ELF file code contents by choosing the **Load (program and symbols) and verify (program)** option, however, note that this option increases the overall time due to the verification step.
- **Use fast download**… will improve your overall program download time, however, it requires that PathFinder-XD download and use a small 512 byte helper-routine to target RAM at the address specified. Make sure you chose a suitable 512 byte RAM location that is not used by your application as PathFinder-XD does not preserve contents.

PathFinder-XD includes a suitable program (`C:\Program Files\Ashling\PathFinder-XDforMIPS\examples\BE\UnCached\Debug-BE\UnCached_BE.elf`) for running on MIPS32 4Kec or 24K core based targets (including the RouterStation Pro). Select this, **Program and symbols** and press **Debug**

6. PathFinder-XD will now download the program and update its Windows as follows allowing you to start your Debug session:

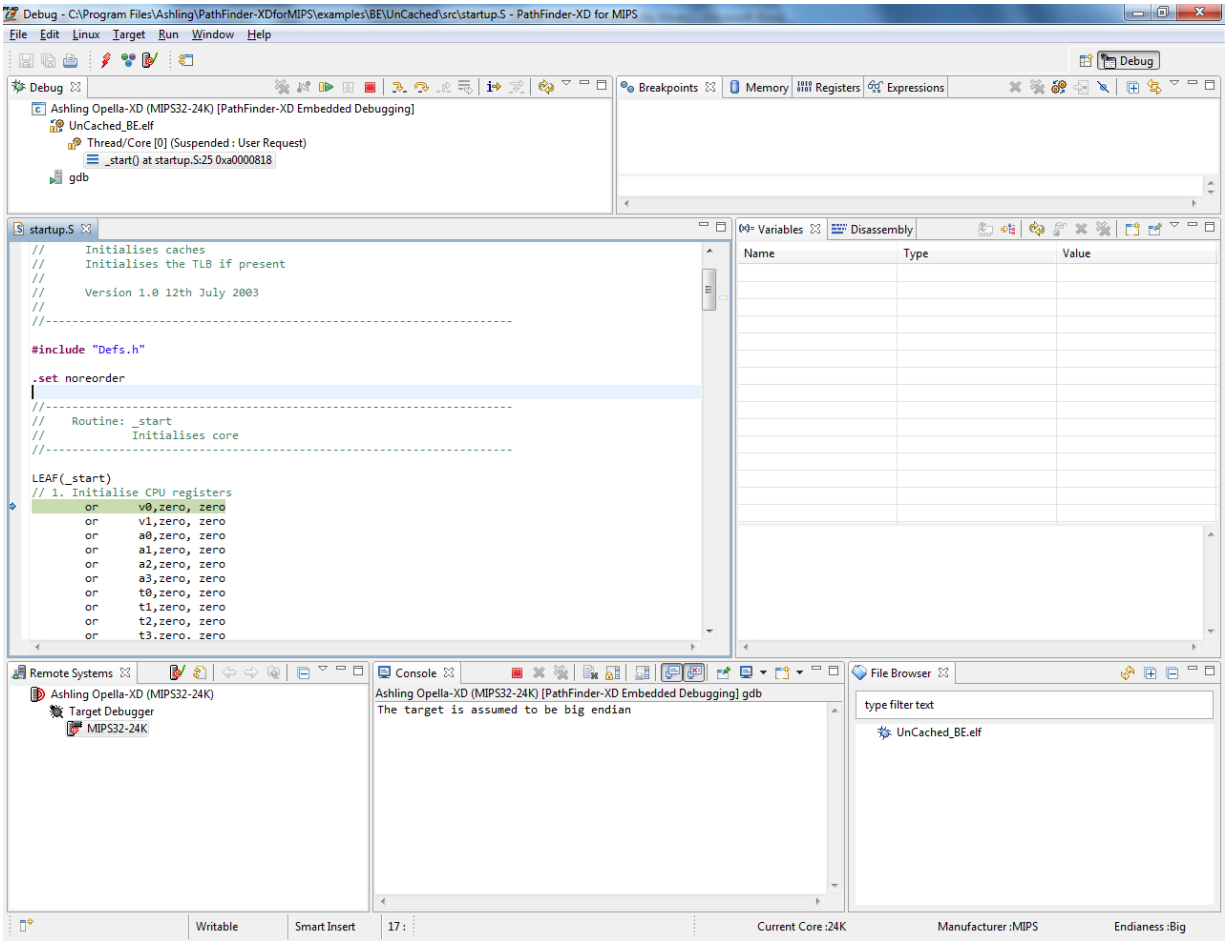
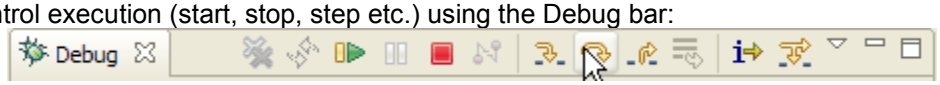


Figure 12. PathFinder-XD after program download

7. You can now control execution (start, stop, step etc.) using the Debug bar:



where the buttons are as follows:

 **Go**

 **Stop/Halt**

 **Step Into, Over and Return (Out)**

 **Terminate** (this button actually terminates the debug session meaning we have to **Download and Launch** again)

Figure 13. Execution Control

When setting/toggling breakpoints in the Source and Disassembly Windows, make sure the mouse pointer is hovering over the left-most column (known as the ruler) of the Window as shown below:

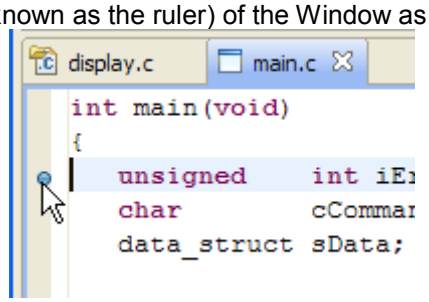


Figure 14. Setting a breakpoint

8. To watch a variable or expression, select it using the mouse and **Add Watch Expression** via the right-mouse button menu as show below:

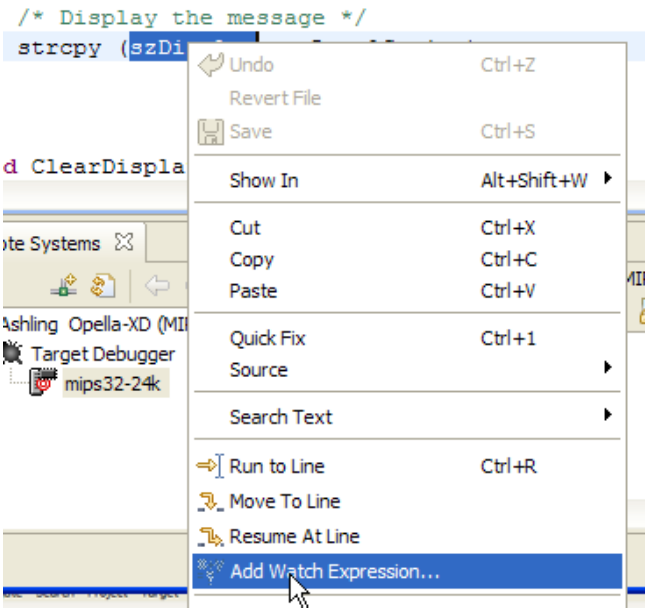


Figure 15. Adding a Watch Expression

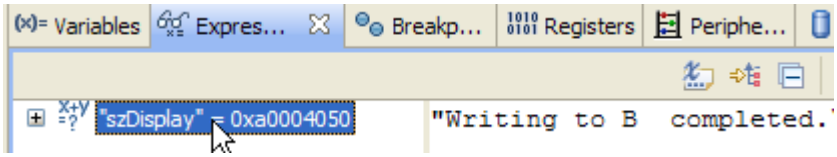


Figure 16. Expression window showing watched expression

You can also quickly watch an expression by hovering the mouse pointer over it as shown below:

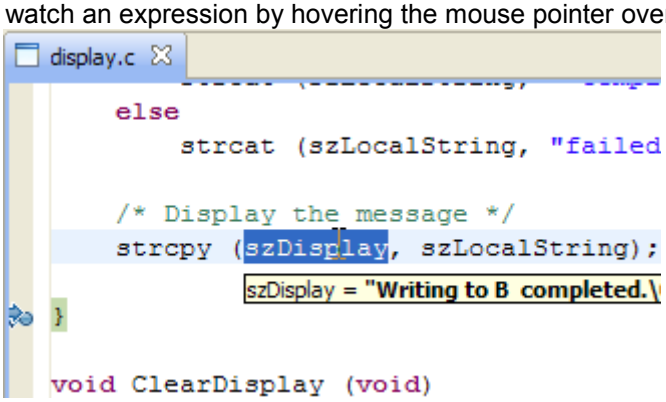


Figure 17. Quick watch via mouse hover

9. PathFinder-XD supports both a Console and Translation Lookaside Buffer Windows (or Views) which can be opened from the Window menu:

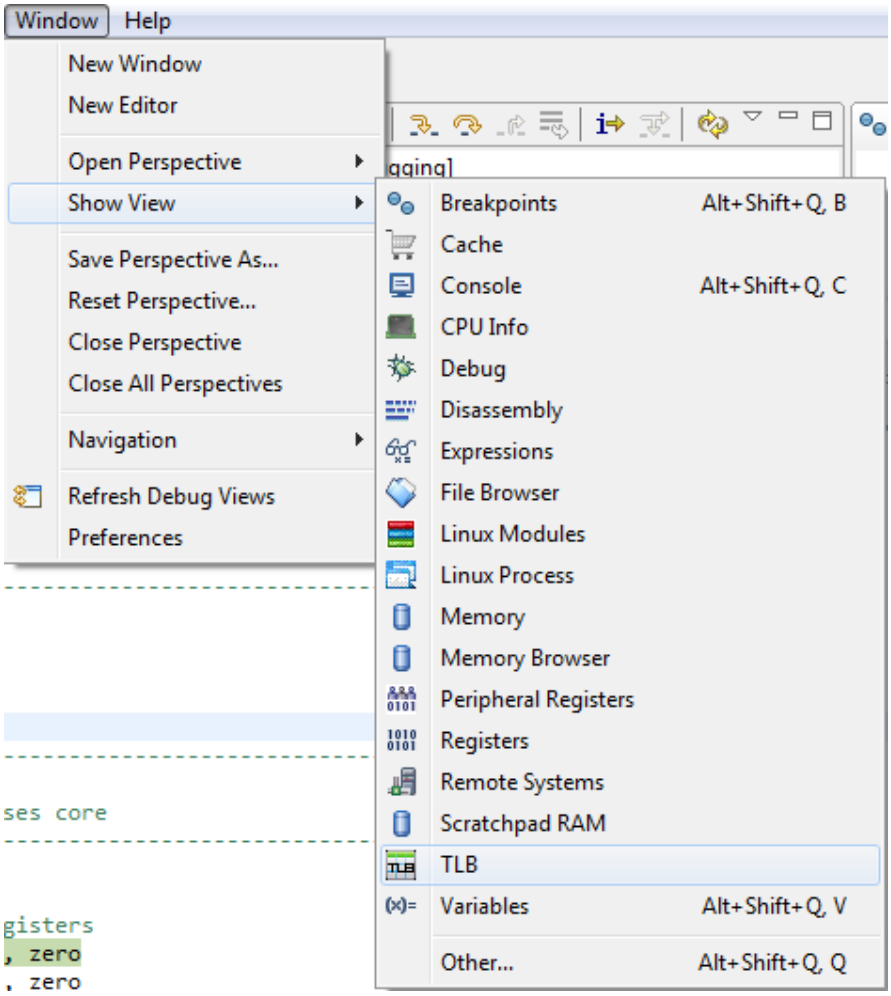


Figure 18. PathFinder-XD Views

Console. Allows you to enter debug commands and view their output. The GNU GDB syntax is fully supported. See here for details: http://sourceware.org/gdb/current/onlinedocs/gdb/index.html or for a handy quick-reference card see here: http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

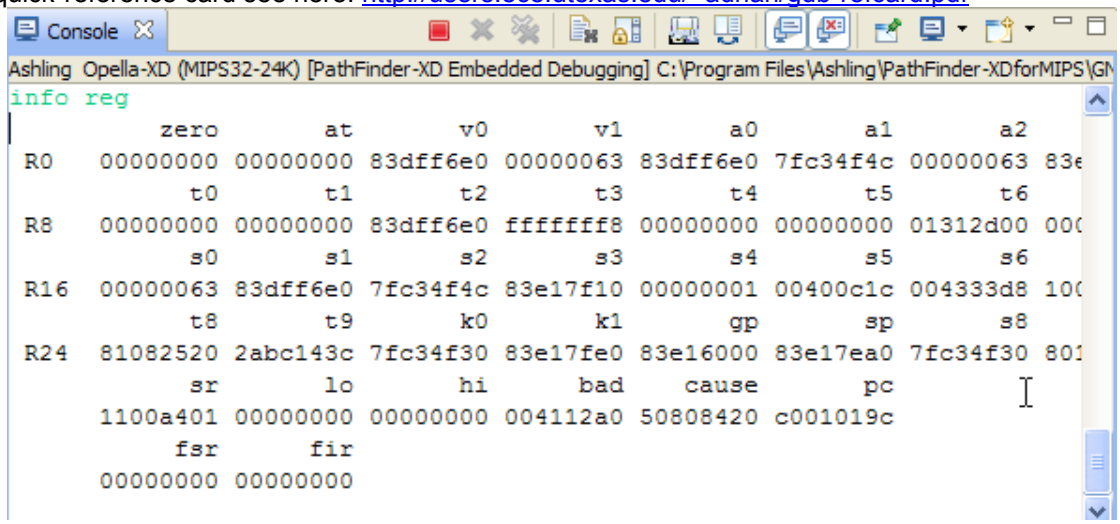


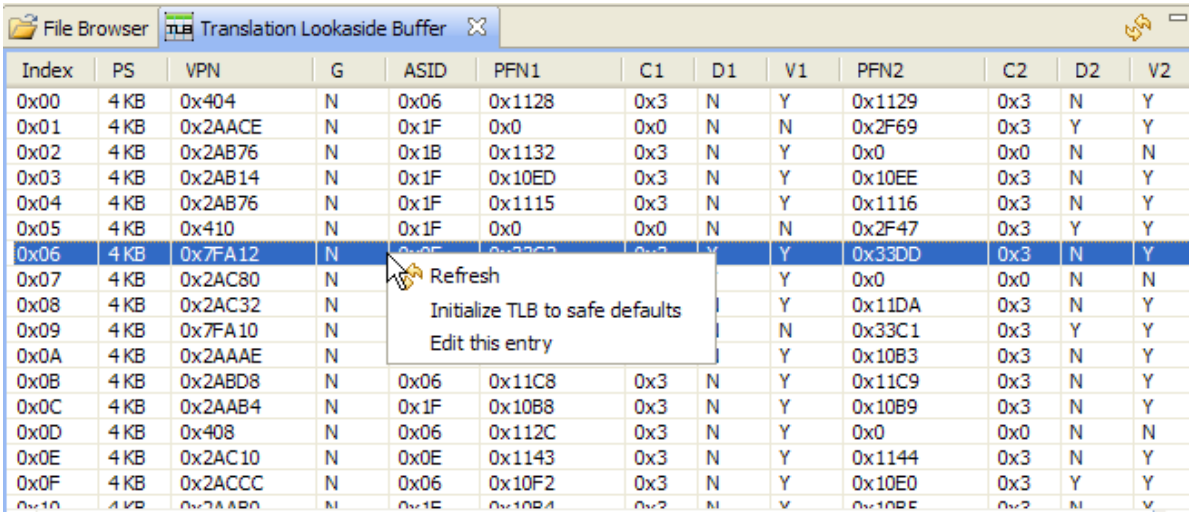
Figure 19. PathFinder-XD Console showing the output of the `info reg` command

For example, to dump **16** **w**ords of memory in he**x** format from 0xA0004200 enter the e**x**amine command as follows:

```
x /16wx 0xA0004200
0xa0004200: 0x00000000  0x00000000  0x74697257  0x20676e69
0xa0004210: 0x41206f74  0x6f632020  0x656c706d  0x2e646574
0xa0004220: 0x00000000  0xa0004228  0x00000004  0xa0004238
0xa0004230: 0x00000000  0x00000000  0x00000000  0x00000000
```

Console commands can also be stored in a text file (GDB script file) and executed from PathFinder-XD's **Run** menu.

- **Translation Lookaside Buffer s**hows the contents of Translation Lookaside Buffer (only available for MIPS devices with EJTAG v2.6 or higher). Right-mouse menu options allow to conveniently setup the TLB to sensible defaults (i.e. perform no mappings)) and edit entries. Note: this Window does not work with the QEMU simulator.



Index	PS	VPN	G	ASID	PFN1	C1	D1	V1	PFN2	C2	D2	V2
0x00	4 KB	0x404	N	0x06	0x1128	0x3	N	Y	0x1129	0x3	N	Y
0x01	4 KB	0x2AACE	N	0x1F	0x0	0x0	N	N	0x2F69	0x3	Y	Y
0x02	4 KB	0x2AB76	N	0x1B	0x1132	0x3	N	Y	0x0	0x0	N	N
0x03	4 KB	0x2AB14	N	0x1F	0x10ED	0x3	N	Y	0x10EE	0x3	N	Y
0x04	4 KB	0x2AB76	N	0x1F	0x1115	0x3	N	Y	0x1116	0x3	N	Y
0x05	4 KB	0x410	N	0x1F	0x0	0x0	N	N	0x2F47	0x3	Y	Y
0x06	4 KB	0x7FA12	N					Y	0x33DD	0x3	N	Y
0x07	4 KB	0x2AC80	N					Y	0x0	0x0	N	N
0x08	4 KB	0x2AC32	N					Y	0x11DA	0x3	N	Y
0x09	4 KB	0x7FA10	N					N	0x33C1	0x3	Y	Y
0x0A	4 KB	0x2AAAE	N					Y	0x10B3	0x3	N	Y
0x0B	4 KB	0x2ABD8	N	0x06	0x11C8	0x3	N	Y	0x11C9	0x3	N	Y
0x0C	4 KB	0x2AAB4	N	0x1F	0x10B8	0x3	N	Y	0x10B9	0x3	N	Y
0x0D	4 KB	0x408	N	0x06	0x112C	0x3	N	Y	0x0	0x0	N	N
0x0E	4 KB	0x2AC10	N	0x0E	0x1143	0x3	N	Y	0x1144	0x3	N	Y
0x0F	4 KB	0x2ACCC	N	0x06	0x10F2	0x3	N	Y	0x10E0	0x3	Y	Y

Figure 20. PathFinder-XD TLB Window

10. Breakpoints can also be set via the **Run|Breakpoint Configuration** dialog. This allows software (RAM) and hardware (RAM/ROM) based breakpoints to be set. Advanced hardware breakpoints (including data access and conditional breakpoints) are also supported

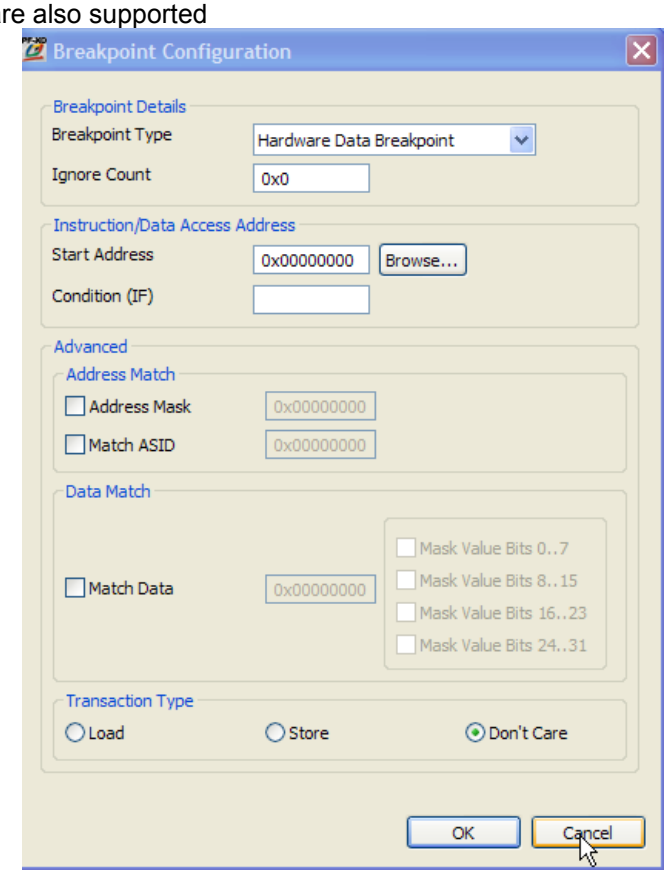


Figure 21. PathFinder-XD Breakpoint Configuration

11. Memory can be viewed via **Window|Show View|Memory**.

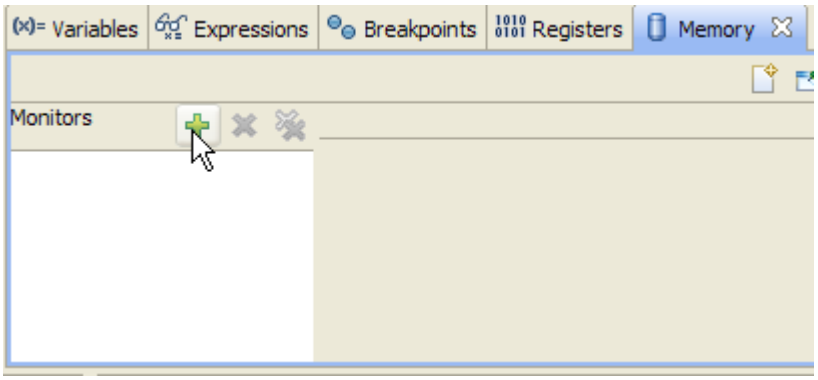


Figure 22. PathFinder-XD Memory View

Add a **Memory Monitor** and specify the address you wish to view (0xA000-0000 in the below example).

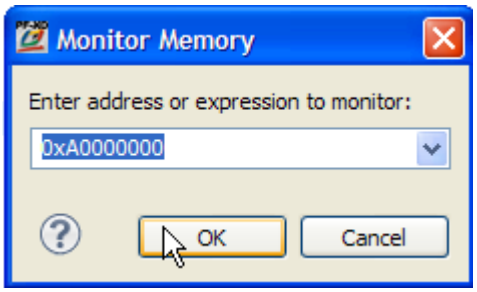


Figure 23. Adding a Memory Monitor

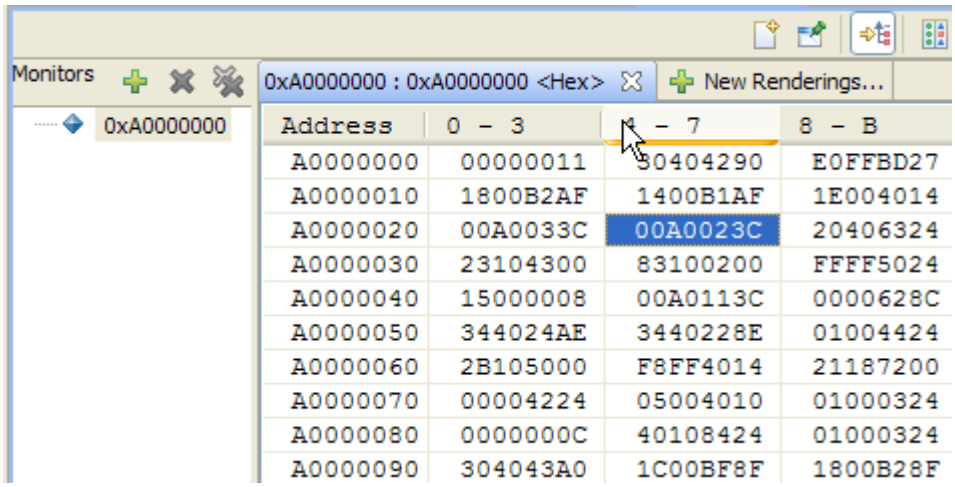


Figure 24. Memory Window showing contents at 0xA000-0000

Full point-and-click in-line editing is supported for writable target memory locations. Select **New Renderings** to show memory as Hex, ASCII etc.

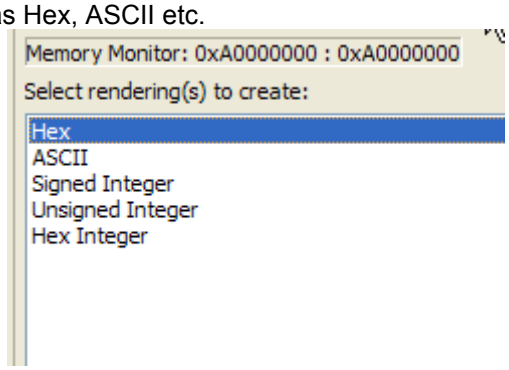


Figure 25. Selecting Memory Renderings

4. Embedded Linux Debugging with PathFinder-XD and Opella-XD

PathFinder-XD supports Embedded Linux Debugging for kernels based on v2.6 or later. Support works in two modes:

- Stop-mode: Debugging is done via the on-chip debug interface (e.g. via Opella-XD) and the whole system is halted (e.g. kernel and applications) whenever a breakpoint is taken.
- Run-mode: Debugging is done purely in software (i.e. no Opella-XD is required) via a target serial/Ethernet interface and requires an application (GDB server) running on the target. In run-mode, the kernel continues to run when an application breakpoint is taken.

Stop-mode debugging is useful for bringing up the kernel as it only requires a functional on-chip debug interface and allows debug from reset. Stop-mode can also be used for process debugging, however, the kernel/interrupts etc. will not continue to run when halted (unlike run-mode). When stop-mode debugging a process, PathFinder-XD automatically scans the kernel MMU mapping for that process and sets up the MIPS core TLB to allow debug access to the process's memory area. Run-mode debugging requires that the kernel is up and running and allows non-intrusive debug of a process (i.e. the kernel will continue to run even when a process is halted). Run-mode also supports thread-aware breakpoints and simultaneous debug of multiple processes.

4.1 Hardware Setup

This section demonstrates Linux Kernel Debugging using PathFinder-XD and Opella-XD connected to an Ubiquiti Networks RouterStation Pro (powered by an Atheros AR7161 MIPS24K based SoC) target running OpenWRT v10.03 (known as Backfire and based on the Linux Kernel v2.6.32). See http://wiki.openwrt.org/ for more details on OpenWRT.

Ashling provide the associated OpenWRT v10.03 Linux Kernel sources files for download at `http://www.ashling.com/support/MIPS/RouterStationPro/MIPS_ROUTER_LINUX_v2.6.32.ZIP` (dated: 30/4/2011 or later) and these should be installed by unzipping to your local hard-disk (ensure you preserve the directory structure as present in the ZIP file). These sources are needed for source-level debug of the kernel and they also include some examples that demonstrate other PathFinder-XD features.

4.1.1 Upgrading RouterStation Pro flash

In order to allow kernel debugging, it is important that your board is programmed with the v10.03 **debug** flash image provided (`openwrt-ar71xx-ubnt-rspro-squashfs-factory.bin` is included in the above .ZIP file in `\MIPS_ROUTER_LINUX_v2.6.32`).

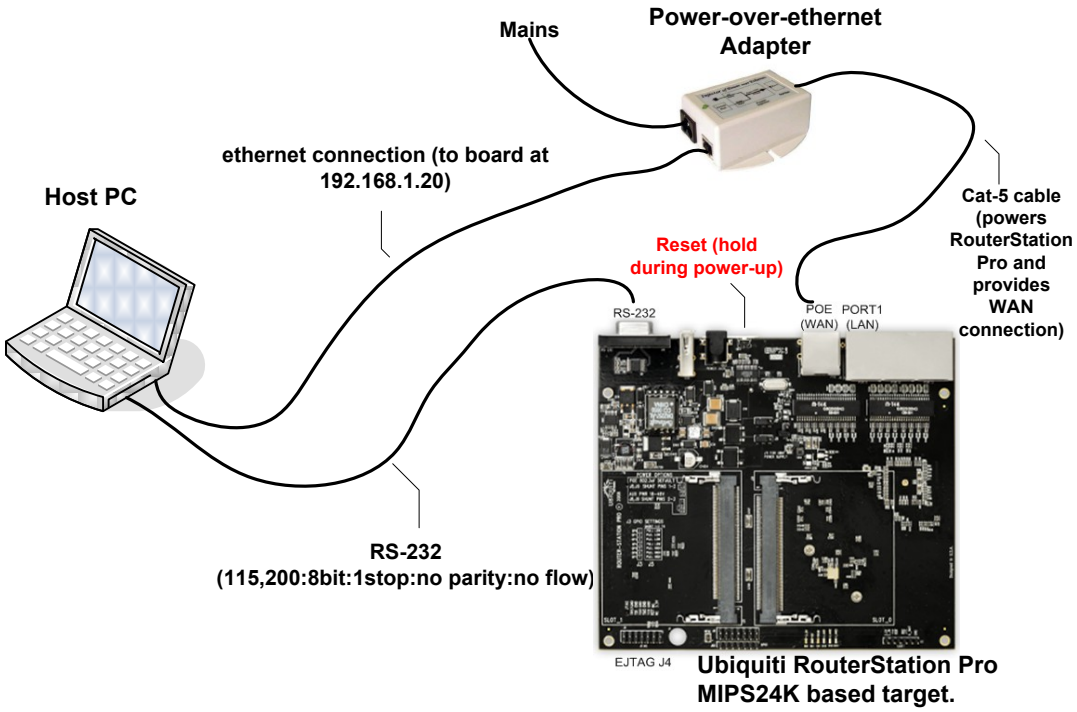


Figure 26. RouterStation Pro flash update

1. Connect as shown in the above figure making sure your Host PC is connected to WAN port on the board (via the P.O.E. adapter which also powers the board). You will need a terminal program (e.g. putty) running on your host PC to show the target's Linux shell (and status messages during boot including the OpenWRT version). The terminal should be configured as per the RS-232 settings shown above.

2. Power-off your board.
3. Hold-down Reset and power-on the board. The board should then boot and after a few seconds show `TFTPD: waiting for connection...` in the terminal indicating that the board is ready to receive an updated flash image at ip address 192.168.1.20.
4. You will need the command-line `tftp` utility to flash the board which may not be installed by default on some Windows hosts (you can install it via the Control Panel as an optional Windows feature). For example, for Windows users issue the following command:
   ```
   >tftp -i 192.168.1.20 put openwrt-ar71xx-ubnt-rspro-squashfs-factory.bin
   ```
   and for Linux:
   ```
   >tftp 192.168.1.20 -m binary -c put openwrt-ar71xx-ubnt-rspro-squashfs-factory.bin
   ```
5. When programming is completed, you will see a display like the following:

```
ar71xx-ohci ar71xx-ohci: new USB bus registered, assigned bus number 2
ar71xx-ohci ar71xx-ohci: irq 14, io mem 0x1c000000
usb usb2: configuration #1 chosen from 1 choice
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 2 ports detected
jffs2_scan_eraseblock(): End of filesystem marker found at 0x0
jffs2_build_filesystem(): unlocking the mtd device... done.
jffs2_build_filesystem(): erasing all blocks after the end marker... done.
mini_fo: using base directory: /
mini_fo: using storage directory: /overlay
```

Figure 27. RouterStation Pro flash update completion message in shell

Press enter to return to the command-prompt and cycle power on the board to reset and run the new flash contents.

6. See http://wiki.openwrt.org/toh/ubiquiti/routerstation.pro for more details on how to upgrade/flash the board if necessary.

4.1.2 RouterStation Pro Hardware Setup for Embedded Linux debugging

For Embedded Linux debugging, setup your hardware as shown below.

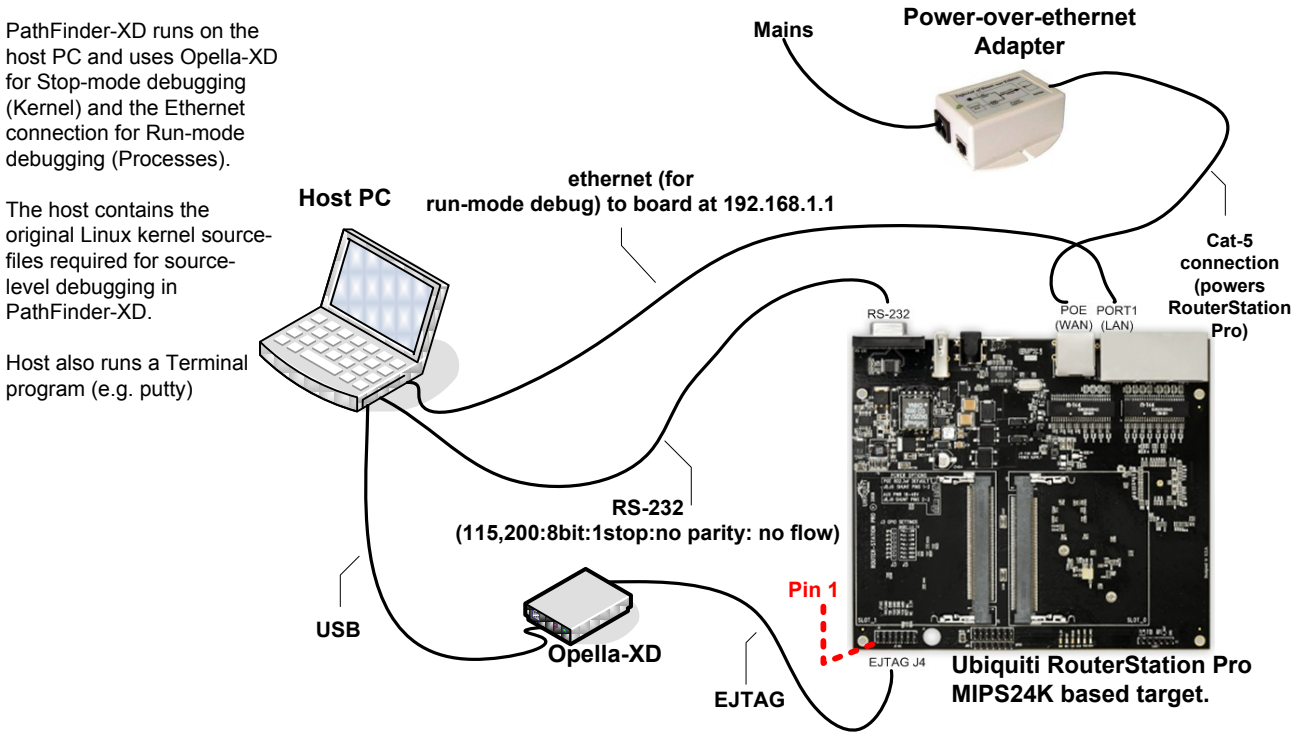


Figure 28. Embedded Linux Demo setup

4.2 Preparing for debugging

This section is only necessary if you are building/using your own Kernel; the version supplied by Ashling includes all of the following requirements.

4.2.1 Building with debug symbols

Your kernel, modules, processes, libraries, drivers etc. must be built with debug symbols as PathFinder-XD needs to access global structures and variables etc. to support Linux debugging. ***Please note*** that debug symbols for Linux kernel (`vmlinux`) are required to debug user-mode applications in stop-mode (to allow PathFinder-XD to handle memory mapping which requires kernel symbols). Kernel symbols are not required for run-mode debugging.

- For the kernel, run make menuconfig, select Kernel hacking, enable Kernel debugging and Compile the kernel with debug and run make to rebuild the kernel with debug symbols.

- For non-kernel items, add the compiler gcc switch **`-g`** (which will generate debug symbols) to your makefile and rebuild.

4.2.2 Compiler optimisations

Compiler optimisations should not be used as they can cause misalignment between the generated symbolic information and the actual generated machine code thus causing problems with debugging. In particular, the flag `--ffunction-sections` should not be used as it will create **.text** sections for every function causing problems for PathFinder-XD. See here for more details: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. To remove these optimizations, change your makefile and rebuild e.g.:

Change **`arch/mips/Makefile`**

from:

```
cflags-y := --ffunction-sections (Line number 51 in Linux kernel 2.6.27)
```

to:

```
#cflags-y := --ffunction-sections
```

4.2.3 On-demand paging (for stop-mode debugging only)

Linux uses “on-demand paging” meaning that a process’s (and its dependant libraries) code, data and stack are not actually paged into memory until they are first used. This can cause problems when you wish to “stop-mode” debug a process from its initialisation as it may not yet be present in memory. For example, you cannot set software breakpoints which require patching of the software breakpoint instruction into the appropriate process’s memory location until the actual associated process code page is in memory. Depending on the size of your target’s memory space and your memory management unit (MMU configuration), you may or may not have this issue. If you do then Ashling provide a kernel patch that will force all of a process’s code, data and stack pages into memory. This file is installed with PathFinder and is called `ash_load_process_pages.c`. Installing the patch requires that you modify some existing kernel files and rebuild; please refer to the file for full details. ***Note that this patch is required only for stop-mode debugging***.

4.3 Stop-mode Debugging

The following features are supported:

- Linux Kernel debugging:
  - Debug modules built as part of the Kernel
- Linux dynamically loadable Modules/Driver debugging:
  - List all inserted modules
  - Debug an already inserted module
  - Debug a module from **`init_module()`**
- Linux process (application) and library debugging:
  - List all running processes and threads
  - Debug a running process
  - Debug a process from **`main()`**
  - Debug shared libraries

4.3.1 Sample Stop-mode Linux Debugging Session

This section demonstrates Linux Kernel Debugging using PathFinder-XD and Opella-XD connected to an Ubiquiti RouterStation Pro target running v2.6.32 of the Linux Kernel. Make sure you hardware is configured as per **4.1.2 RouterStation Pro Hardware Setup for Embedded Linux debugging.**

1. In PathFinder-XD, create a **New Target Configuration** via the **Target** menu

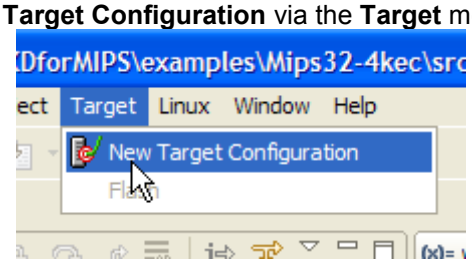


Figure 29. Target Configuration

and select the **Debug using Debug Probe** option as shown below

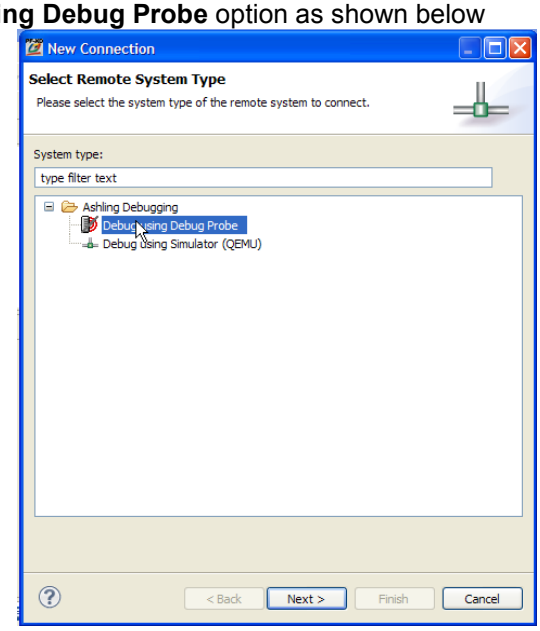


Figure 30. Debug using Debug Probe

2. Click **Next** and we can now configure our Opella-XD settings as shown below:

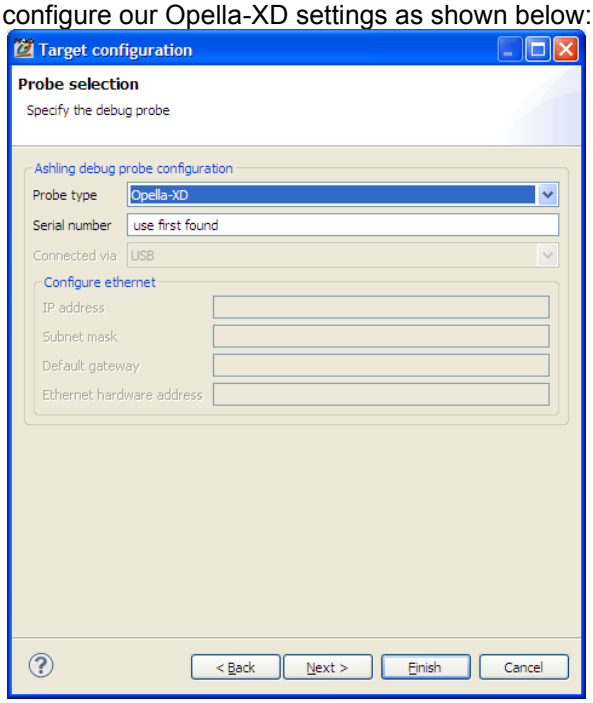


Figure 31. Probe selection

Settings include:

- **Probe type:** The actual Ashling Debug Probe Type to use as the target connection. Select **Opella-XD**
- **Serial number:** The serial number of the Debug Probe to use. Specify the serial number or **use first found** and click on **Next**

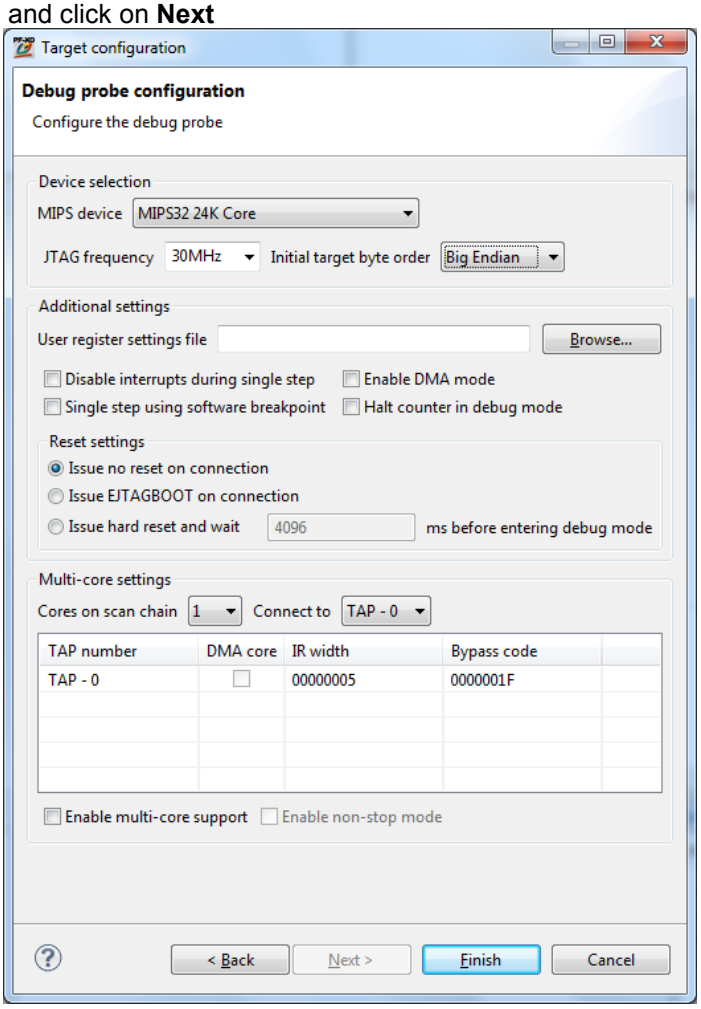


Figure 32. Debug probe configuration

The **Debug probe configuration** settings include:

- **MIPS device:** specifies the MIPS device type you wish to debug. In this example, Broadcom BMIPS5000 is selected.
- **JTAG frequency:** specifies the JTAG TCK frequency to be used for communicating with the EJTAG interface on your MIPS device
- **Initial target byte order:** allows you to specify the memory Endianess of your target system.
- **User register settings file:** group allows you to initialise other registers or memory locations on PathFinder-XD invocation and after reset. The **Browse…** button allows these register values to be loaded from a simple text file. The text file format is:

  ```
  Name  Size Address Value
  ```

  (all values are in HEX). For example, the following text file initialises the R0, R1, R2 and R3 registers:

  ```
  R0 0x00000004 0xb800380c 0x18000000
  R1 0x00000004 0xb8003808 0x00000006
  R2 0x00000004 0xb8004018 0x00000800
  R3 0x00000004 0xb800401c 0x0000000c
  ```

- **Disable interrupts during single step:** allows you to disable interrupts when single stepping at assembly level (MIPS instruction level). When checked, PathFinder-XD automatically disables interrupts prior to assembly level single step and re-enables them after the single step is complete.
- **Enable DMA Mode:** enables DMA mode for high-speed transfer between the debug probe and your target. DMA Mode is only available on systems with EJTAG DMA support.
- **Single step using software breakpoint:** allows you to specify that PathFinder-XD should use software breakpoints for single-stepping (i.e. PathFinder-XD should not use the EJTAG hardware based single-step command).

- **Halt counters in debug mode:** allows you tell PathFinder-XD to halt the MIPS Count register(s) (via writing to the Configuration register) whenever your program is halted. There is a slight delay between your program halting and the write to the Configuration register. Note that the **Registers** window always shows your application values for the Configuration register.
- **Issue no reset on connection**: will ensure that no hardware reset is issued when you connect to your target (note that this feature requires updated Opella-XD firmware (v1.1.1 or later) which is supplied with PathFinder-XD v1.0.6 or later).
- **Issue EJTAGBOOT on connection**: will issue a hardware reset and halt the target at the reset location.
- **Issue hard reset and wait 'N' ms before entering debug mode**: will issue a hardware reset and wait the specified number of ms before entering debug mode. This mode is also known as NORMALBOOT.
- **Multi-core:** allows you to select the core you wish to debug for multi-core devices.

The settings shown are suitable for an Ubiquiti RouterStation Pro target board. Click Finish when done.

3. PathFinder-XD will now create a new **Target Debugger** setting in its **Remote Systems** Window as shown below:

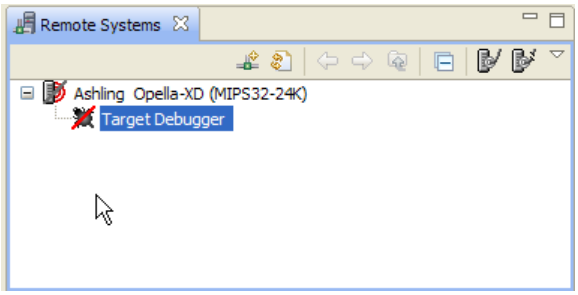


Figure 33. Remote Systems Window

Right-click on **Target Debugger** and click **Connect to** invoke the Opella-XD target connection. Once invoked, the **Remote Systems** window will update as follows:

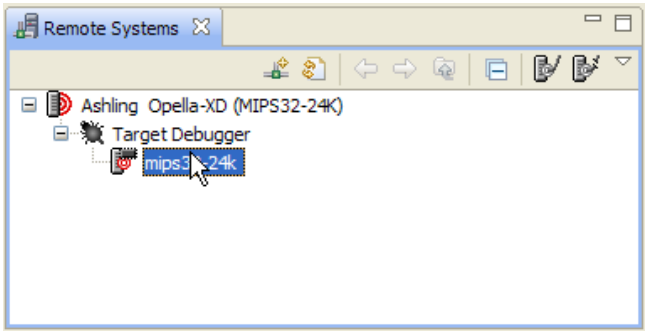


Figure 34. Remote Systems Window showing target connection

4. We can now download a program to the target by right-clicking over **mips32-24k** and selecting **Download and Launch** as follows:

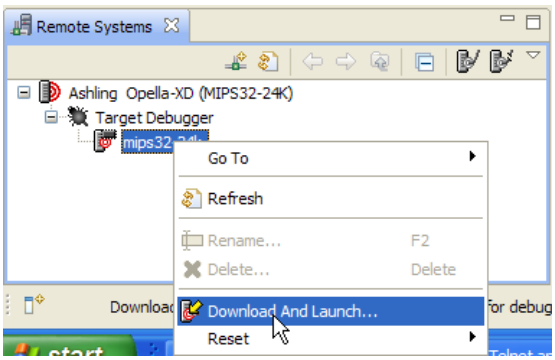


Figure 35. Download and launch

4.3.1.1 Loading kernel symbol information to PathFinder-XD

First, enable Linux debugging via the **Enable OS debugging** check box (this ensures that PathFinder-XD will add the **Linux** specific menu allowing you to perform Module and Process debugging). In this example, our Linux kernel binary image and root file system is flashed on the RouterStation Pro board hence, we only need to select **Symbols only** (for the kernel image) into PathFinder-XD to allow symbolic kernel debug.

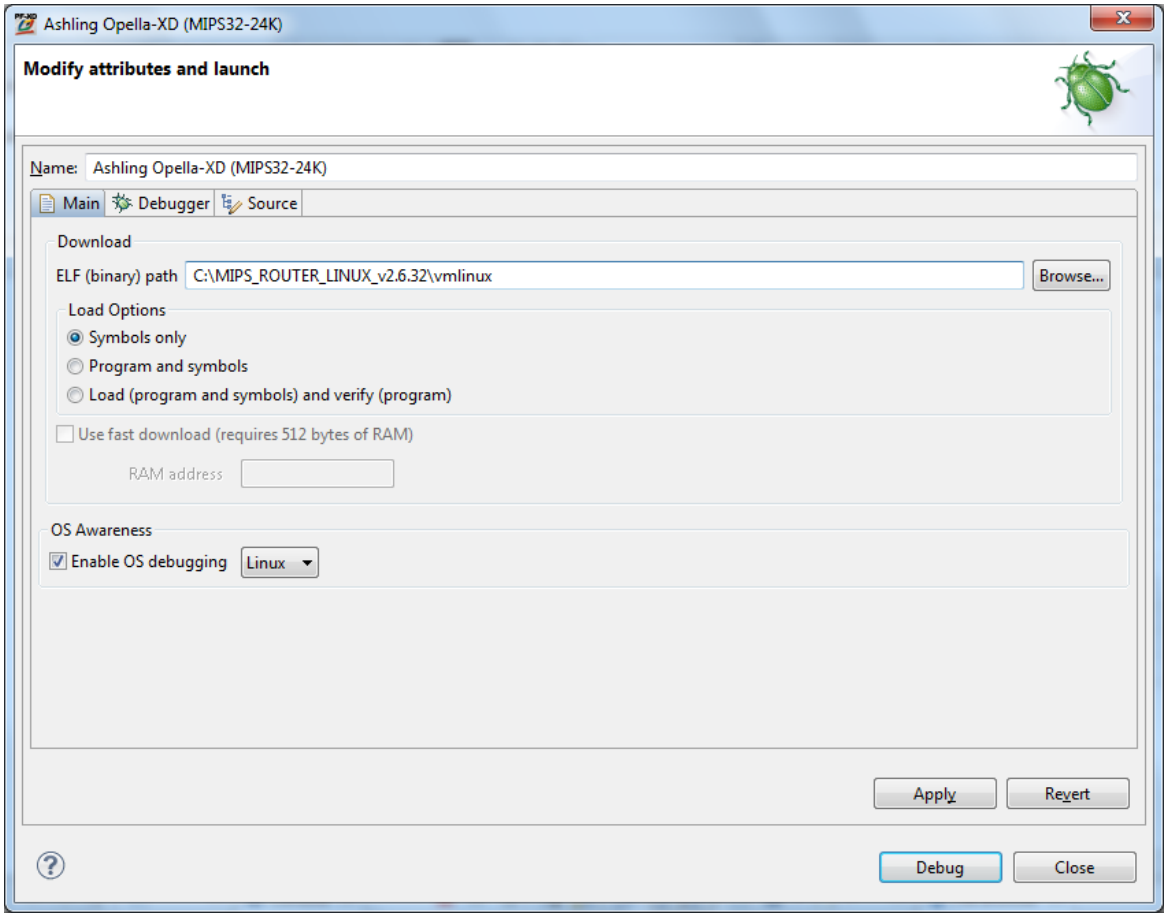


Figure 36. Loading the kernel symbols

Select the **Debugger** tab and make sure that `mips-linux-gnu-gdb.exe` is specified as the **GDB debugger**

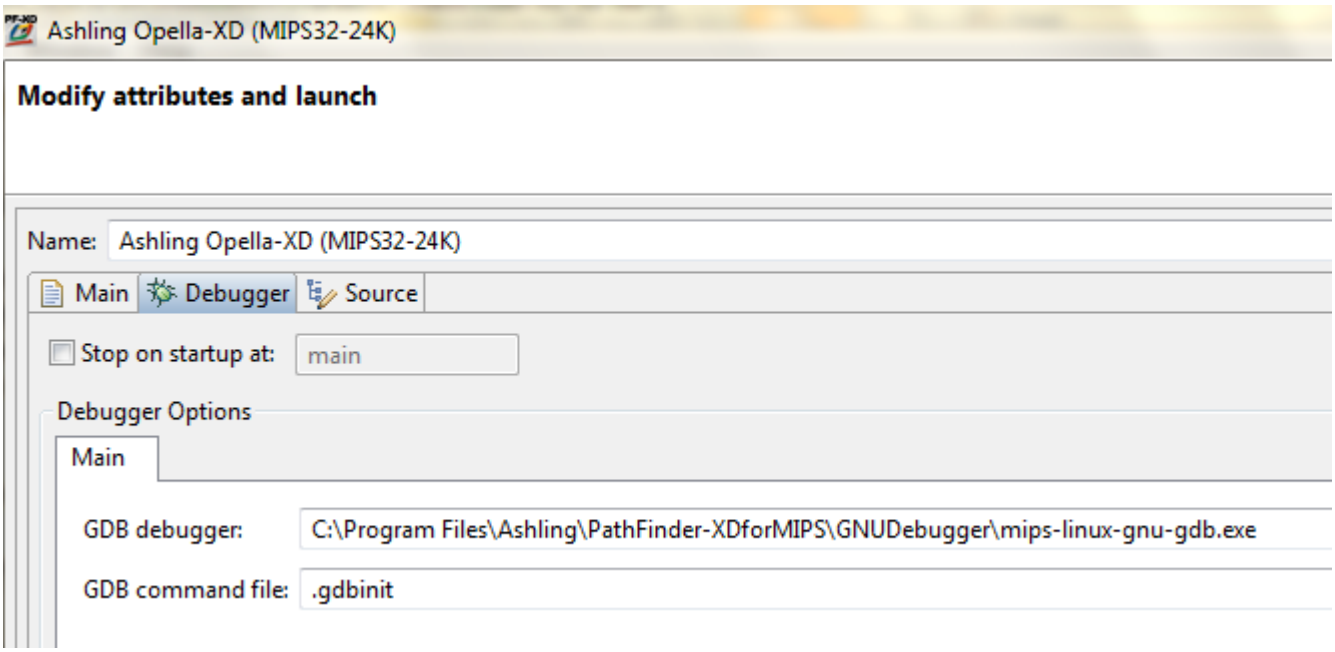


Figure 37. Specifying the correct GDB debugger

Select **Debug** and now execute the target (Run ) and the kernel will boot and show status messages (including OpenWRT version) to your terminal window.

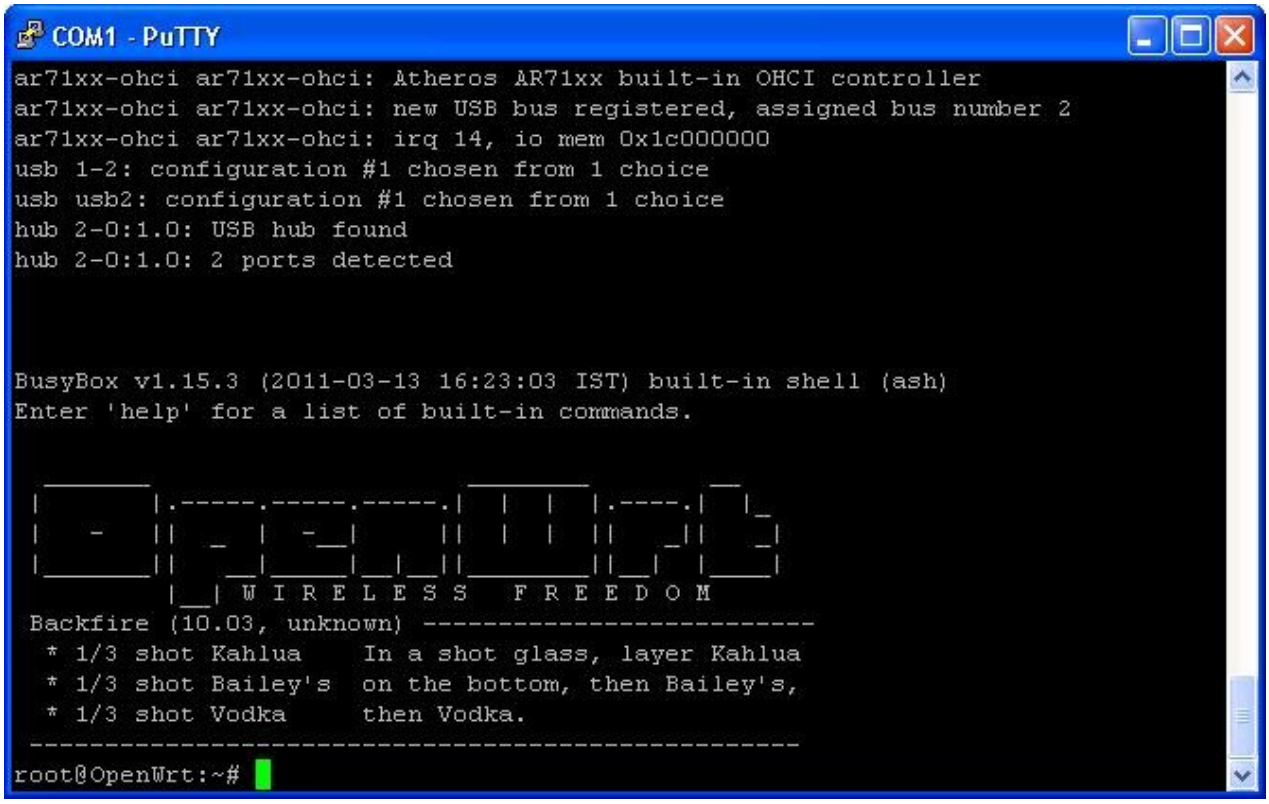


Figure 38. RouterStation Pro Linux shell

Once the kernel is booted, we can halt it within PathFinder-XD (by pressing Stop/Halt). PathFinder-XD then updates as follows:

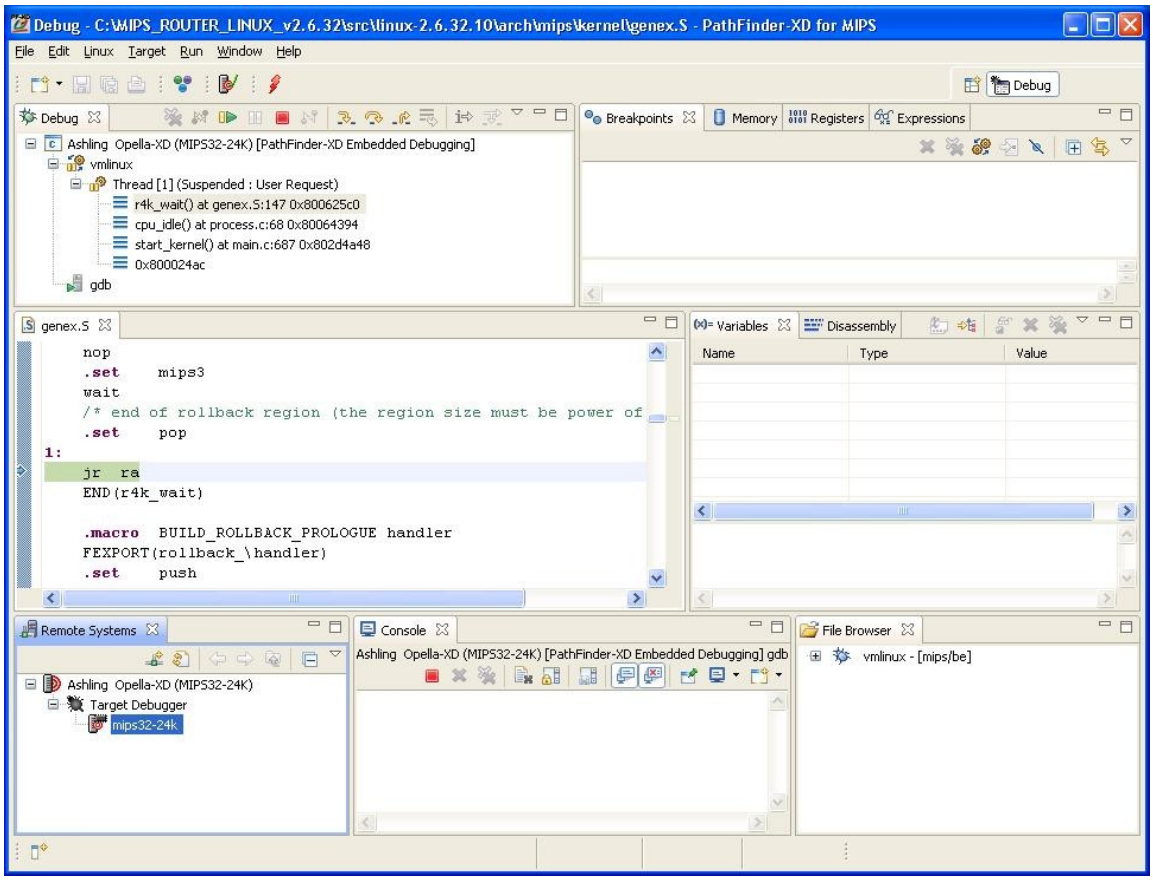


Figure 39. PathFinder-XD after halting the kernel

Notice the following windows:

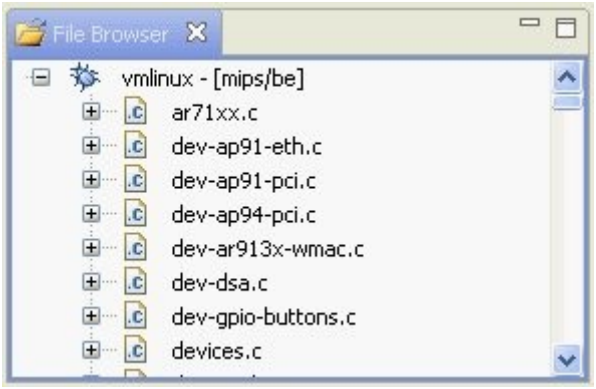


Figure 40. File Browser window showing all kernel source-files

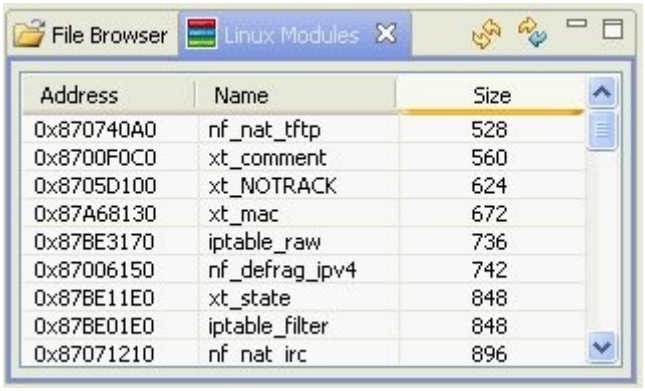


Figure 41. Linux Module window showing all currently loaded kernel modules (enabled via Linux menu)

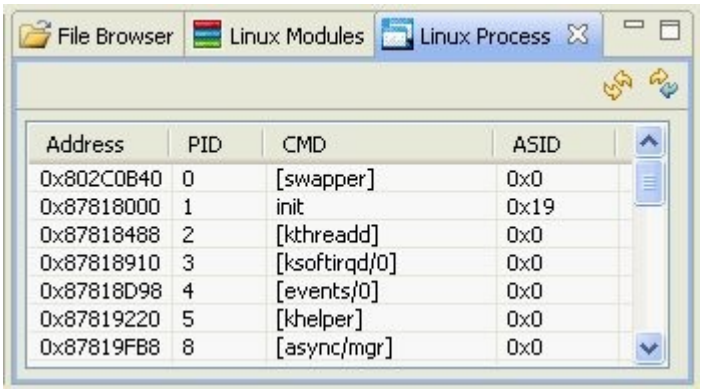


Figure 42. Linux Process window showing all processes (enabled via Linux menu)

Full kernel source-level debug is now possible.

4.3.1.1.1 Debug a module from `init_module()`

Use the **Linux|Modules|Debug A Module From Initialisation** menu to debug a module from it's `init_module()` entry point as follows:

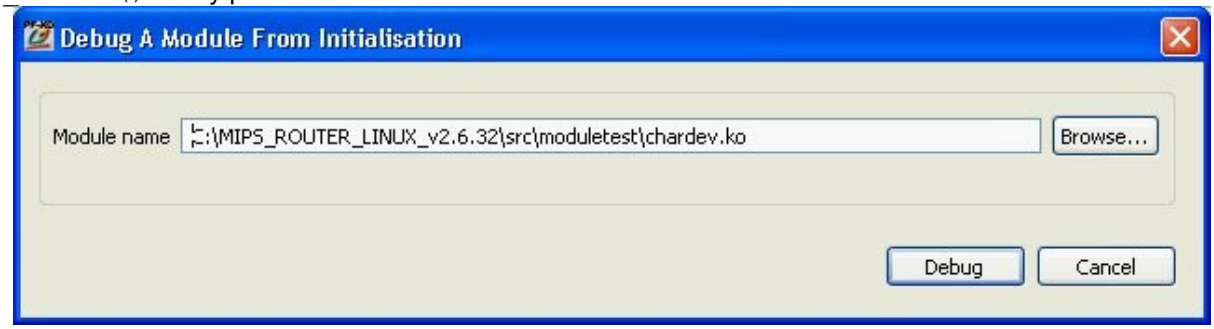
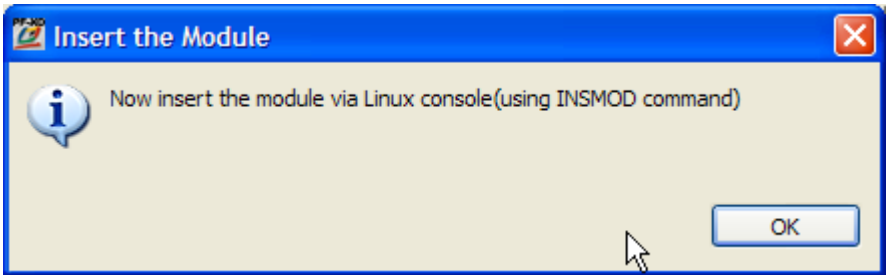


Figure 43. Specifying the module to debug

Once specified, you now need to insert the module via the console as follows:




```
root@OpenWrt:~# insmod chardev.ko
chardev: module license 'unspecified' taints kernel.
```

Figure 44. Inserting (running) the module

PathFinder-XD then halts the module at `init_module()` allowing module debug as shown below:

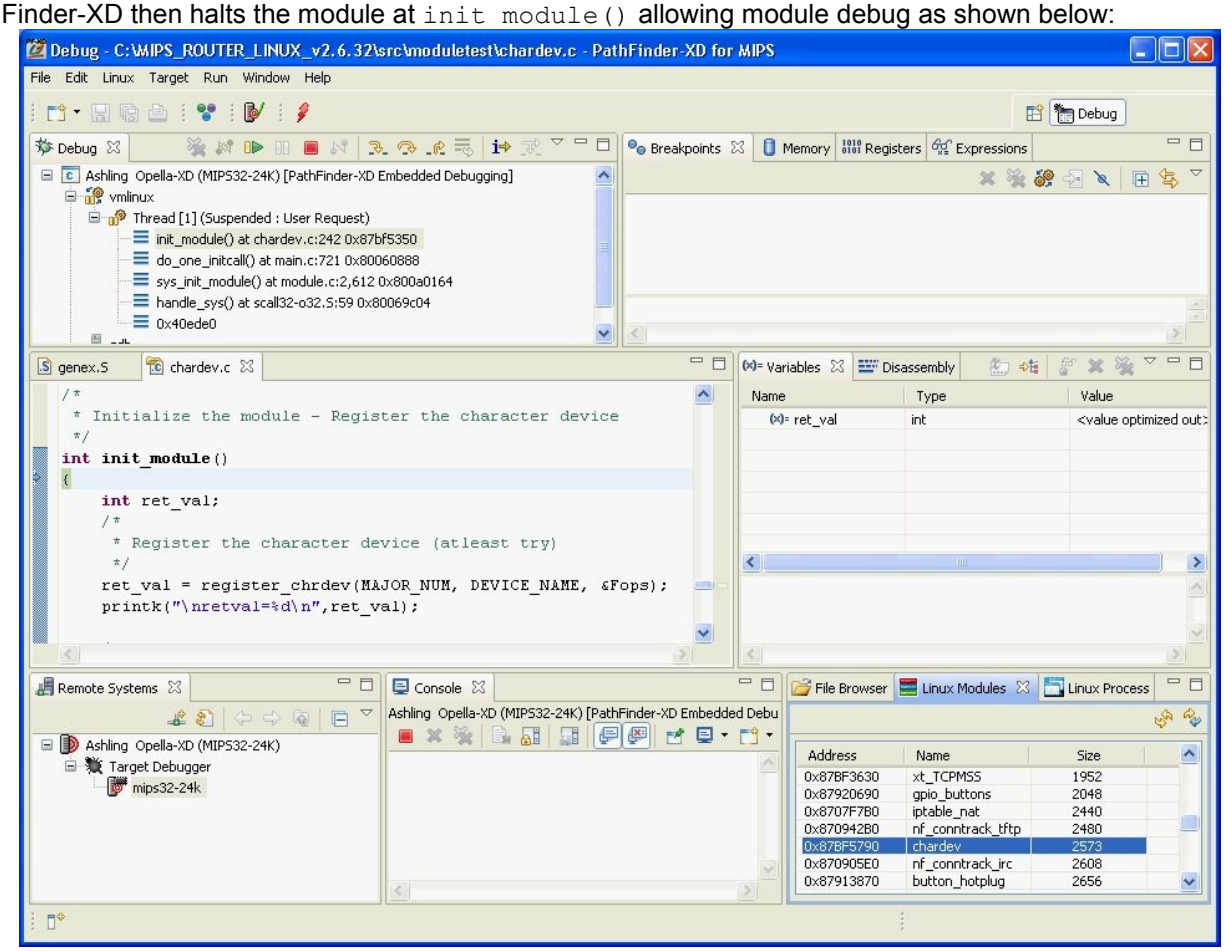


Figure 45. PathFinder-XD halted at `init_module()` allowing module debug

PathFinder-XD's **File Browser** will also update to show the source files associated with the module:

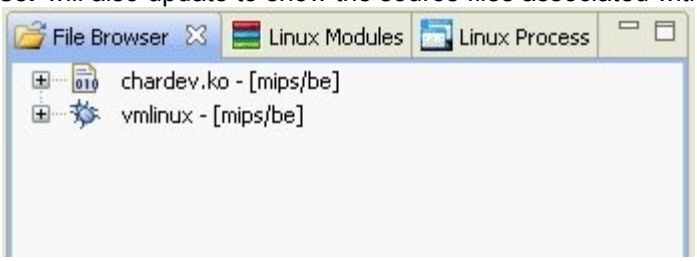


Figure 46. File Browser window showing modules sources

And the **Linux Modules** window will now list the new module:

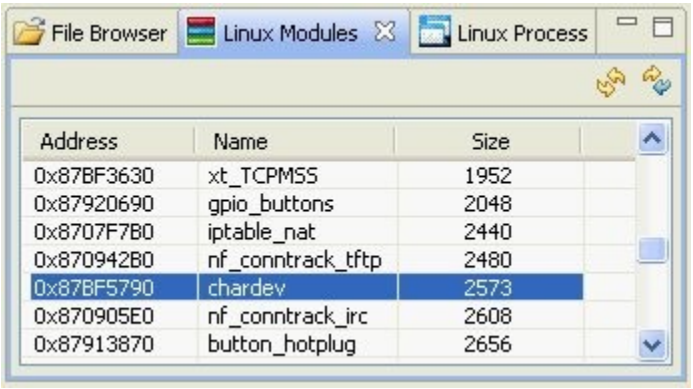
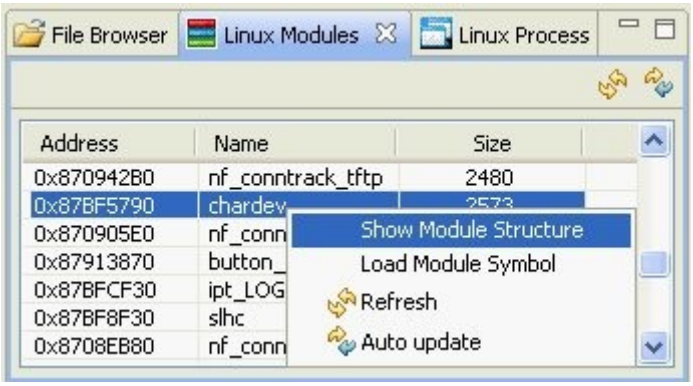


Figure 47. Linux Modules window listing the new module

You can also view the internal module kernel structures via the right-mouse button menu as follows:



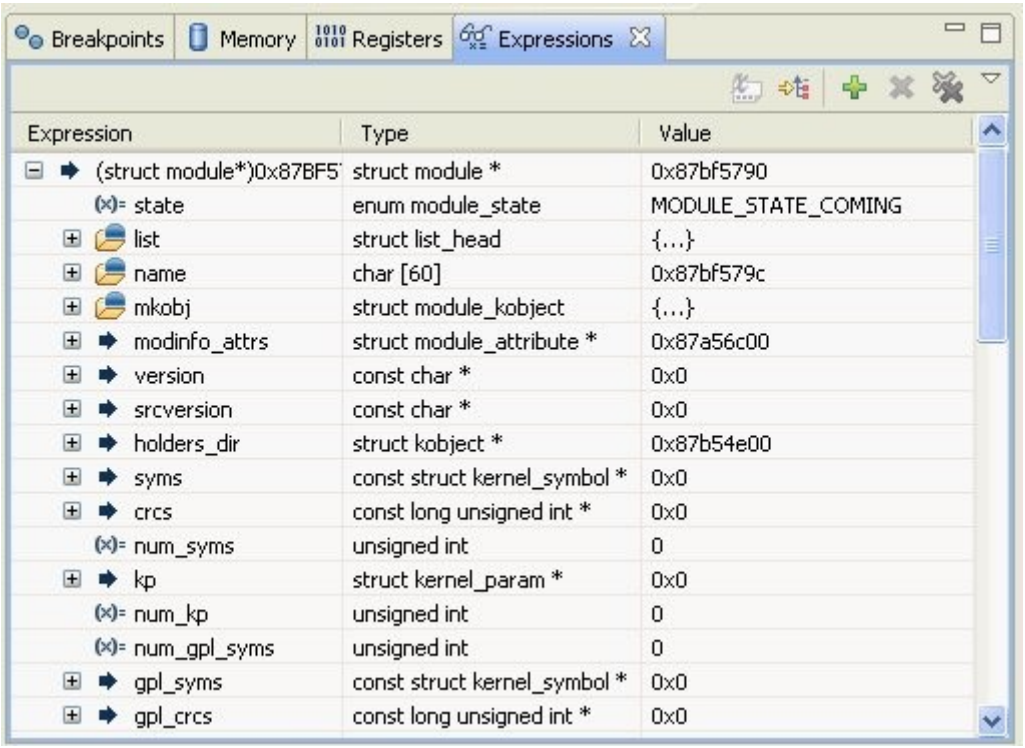


Figure 48. Viewing the internal kernel module structures

In addition, you can load module symbols for a module that is already loaded (Load Module Symbol menu option in the right-mouse button menu)

4.3.1.1.2 Debugging a process from `main()`

Use the **Linux|Processes|Debug A Process From `main()`** to debug a process from it's entry point as follows:

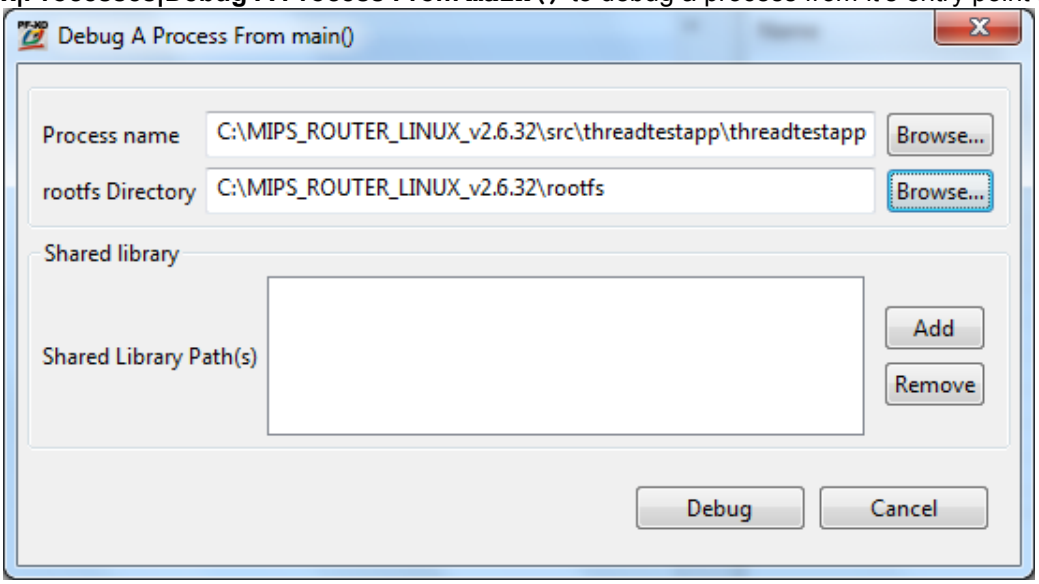
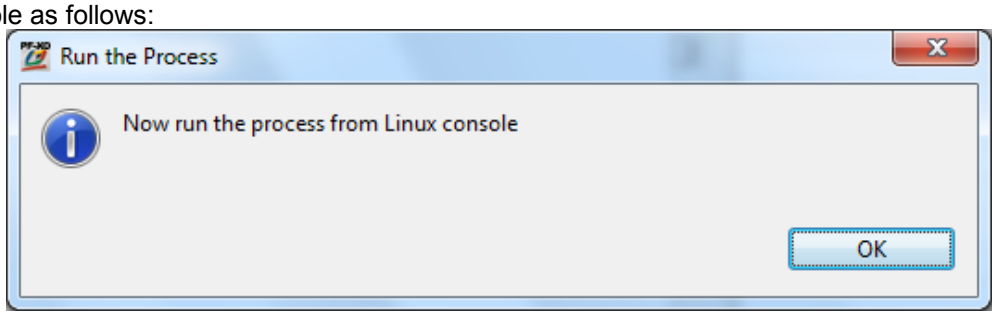


Figure 49. Debugging a process from `main()`

rootfs Directory specifies where the root file-system (`rootfs`) resides in your host machine. This location is needed for loading shared library symbols in PathFinder-XD. Once specified, you now need to run the process from the console as follows:



Press OK and PathFinder will run Linux allowing you to enter a console command as follows:

```
root@OpenWrt:~# ./threadtestapp
```

Figure 50. Running the process

PathFinder-XD then halts the process at `main()` function as shown below:

genex.S | chardev.c | threadtestapp.c

```
int main( void )
{
 pthread_t thr1, thr2;
 //char thr1, thr2;
 thread_data tr1_data, tr2_data;
 unsigned long main_count;

 // initialise thread data and create
 strcpy (tr1_data.msg,"In first thread");
 strcpy (tr2_data.msg,"In second thread");
 tr1_data.cnt=tr2_data.cnt=main_count=1;
 pthread_create( &thr1, NULL, threadfn, (void*)&tr1_data);
 pthread_create( &thr2, NULL, threadfn, (void*)&tr2_data);

 // loop forever, servicing threads
  while (1)
    {
    printf("In main [%08lX]\n",main_count++);
    usleep(5000);
    }


  // never get here
  pthread_join( thr1, NULL );
  pthread_join( thr2, NULL );
```

Figure 51. PathFinder-XD halted at the process's `main()` function

The **File Browser** window will update to show the process's source-code.
Note: To exit the application, press Ctrl+C from the Linux console.

4.3.1.1.3 Debugging a running process

You can load the symbols for a running process via the Linux Process window. Right-click on the process and select Load Process Symbol:

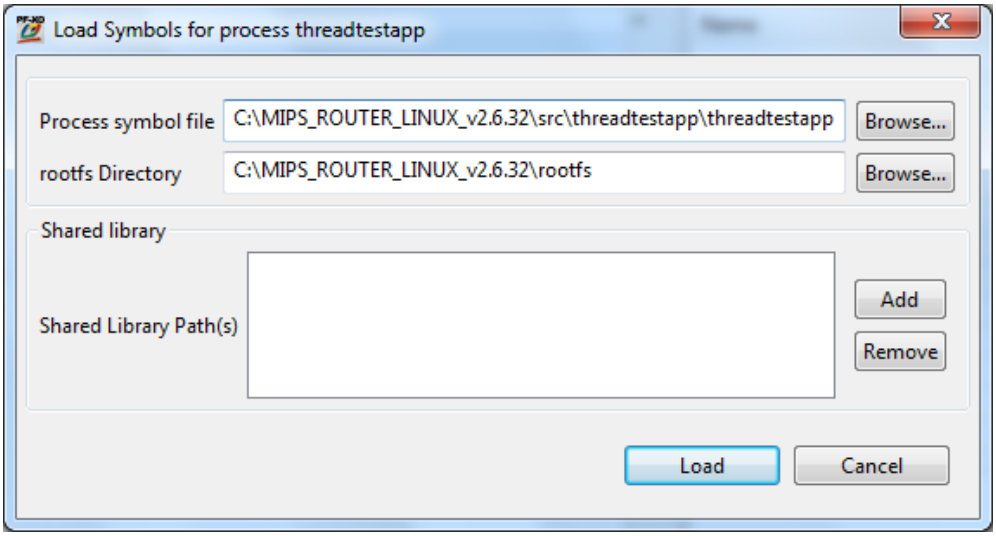


Figure 52. Loading a process's symbols

It is recommended that you use hardware breakpoints when debugging a running process (i.e. do not use software breakpoints as the process may not be paged in at this point). Once the hardware breakpoint has been taken the process is in memory, hence, you can use software breakpoints.

4.3.1.1.4 Library debugging

Debugging of libraries is handled seamlessly without any extra requirements/setup.

4.3.2 Run-mode Debugging

Run-mode debugging is done via a target Serial/Ethernet interface and requires an application (GDB server) running on the target. In run-mode, the kernel continues to run when a process (application) breakpoint is taken. Run-mode debugging requires that the kernel is up and running and allows non-intrusive debug of process (i.e. the kernel will continue to run even when a process is halted).

4.3.2.1 Sample Run-mode Linux Debugging Session

This section demonstrates Linux Process Debugging using PathFinder-XD and Opella-XD connected to an Ubiquiti RouterStation Pro target running v2.6.32 of the Linux Kernel. The example will demonstrate debugging of a Process and a Module (that contains functions called from the Process). Kernel/Module level debugging is done via the Opella-XD; Process debugging is done via an Ethernet connection to the target.

As before we have to prepare our kernel for debug, download it to the target, execute it and load the kernel symbols into PathFinder. See previous sections. Once these steps are complete we are ready to begin debugging our Module and Process as follows:

4.3.2.1.1 Copying the necessary files to the target

Ashling provide a precompiled version of the GNU `gdbserver` (v7.2 or later) to support run-mode debugging and by default this is included in the root file-system provided by Ashling for debugging the RouterStation Pro board, hence, no copying is necessary.

When debugging your own target, note that the `gdbserver` application is installed with PathFinder-XD in `PathFinder-XDforMIPS\target\linux\gdbserver` and versions are supplied for big/little endian and `libc/uclibc` target libraries.

4.3.2.1.2 Debugging the Module and Process

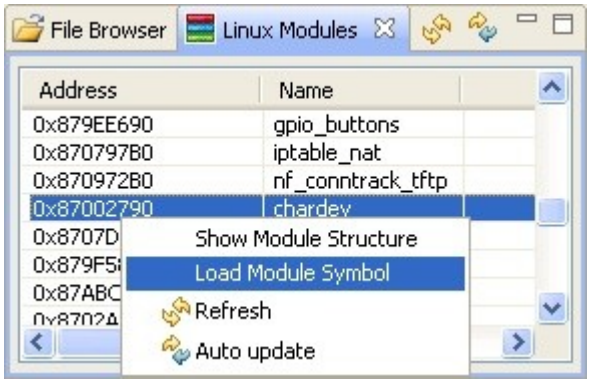
1. First we load the Module (using `insmod`) from our Linux shell as follows:

```
root@OpenWrt:~# insmod chardev.ko
chardev: module license 'unspecified' taints kernel.
```

Figure 53. Loading the Module to be debugged

Note: do not attempt to load a module twice or debugging will not work correctly (use `rmmod chardev.ko` if you need to remove or unload the module)

2. Now, we halt the kernel in PathFinder-XD and load the Module symbols from within the PathFinder-XD **Linux Modules** window:



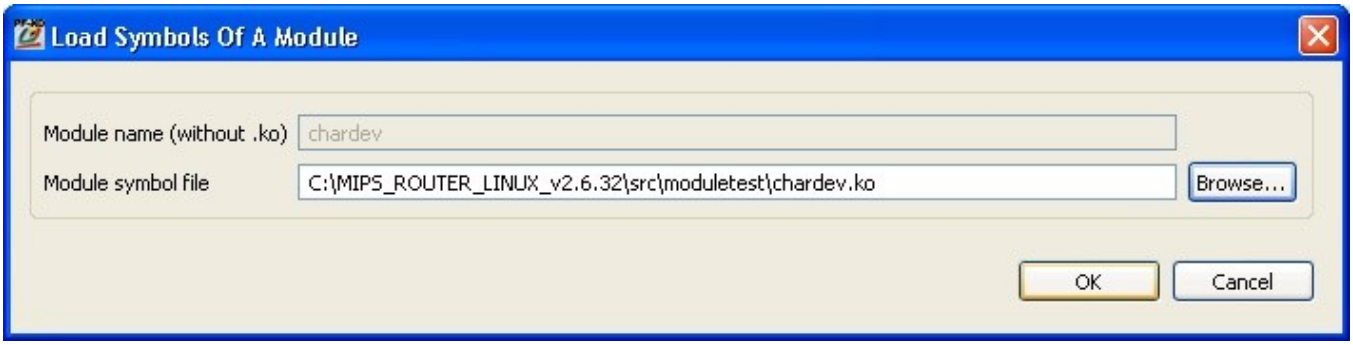


Figure 54. Loading the Module symbols

3. Notice how the File Browser now shows the Module and Kernel symbols:

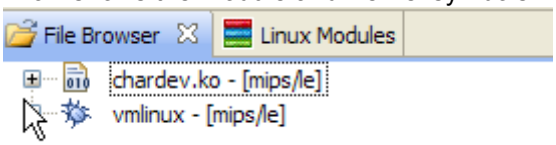


Figure 55. File Browser showing Kernel and Module symbols

We can double-click on the Module to list the files and double-click on a source-file to show it in the Source Window. In the below example we have opened the Module source-file (`chardev.c`) and set a breakpoint at the function (`device_read`) that we wish to debug (i.e. when this function is called from the Process)

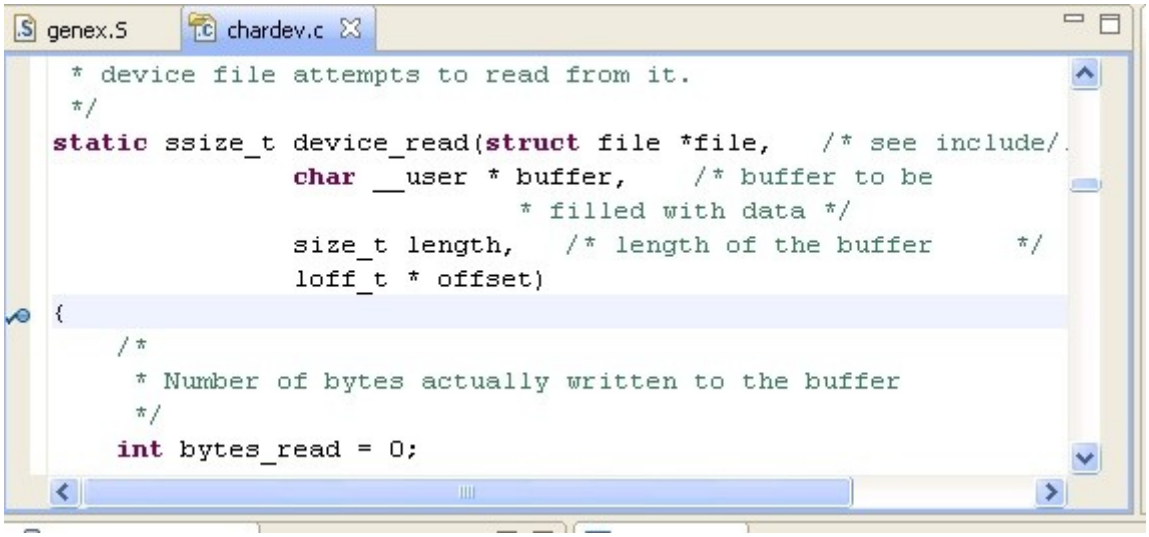


Figure 56. Setting a Breakpoint in the Module

4. Next, we run our Kernel in PathFinder-XD

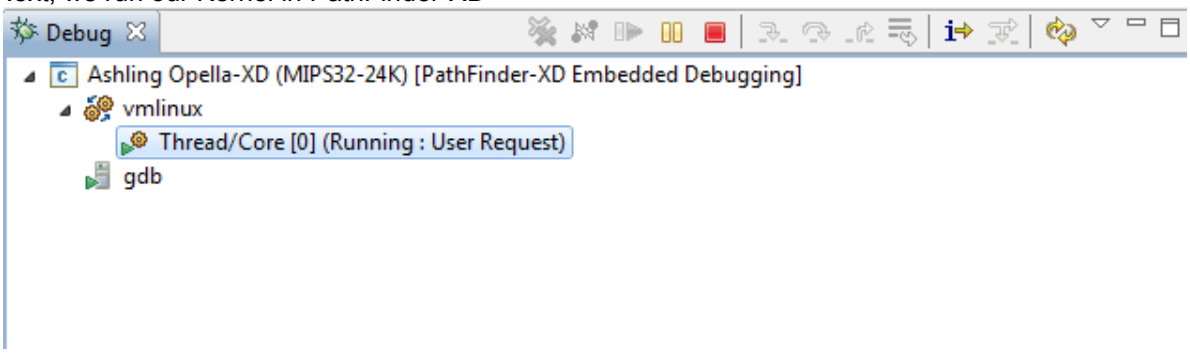


Figure 57. Running the Kernel

and launch gdbserver on the target (i.e. in the Linux shell) specifying the Process we wish to debug (`testapp`). Notice how we tell `./gdbserver` which port to listen on (1234)

```
root@OpenWrt:~# gdbserver :1234 ./ashtestapp
Process ./ashtestapp created; pid = 922
Listening on port 1234
```

Figure 58. Launching the Process

5. Now we need to **Debug A Process in Run-mode** using PathFinder-XD (the Kernel is now running) as follows:

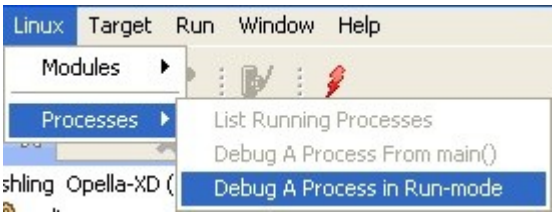


Figure 59. Debugging a Process in Run-mode

We need to specify the Process:

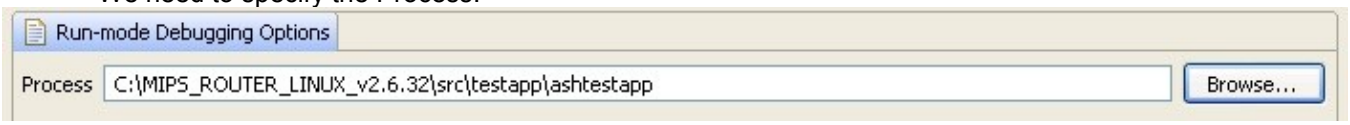


Figure 60. Specifying the Process

The location of the shared libraries:

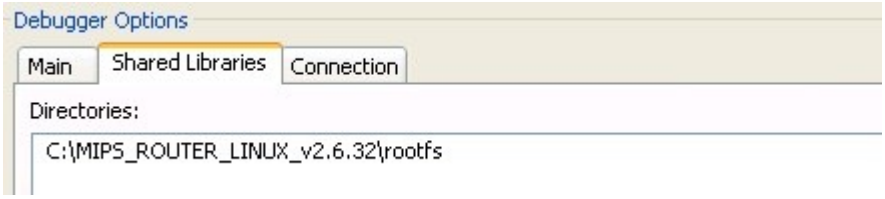


Figure 61. Specifying the Share Library location

And finally, the connection mechanism (TCP in our example) and IP address of the target system (i.e. the RouterStation Pro at 192.168.1.1 which is running gdbserver on port 1234):

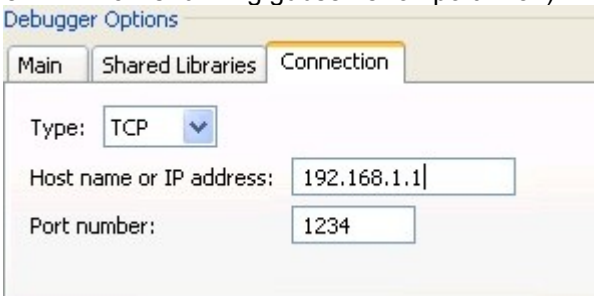


Figure 62. Specifying the Connection mechanism

Make sure that `mips-linux-gnu-gdb.exe` is specified as the **GDB debugger** (default) and press **Debug** to start debugging the Process

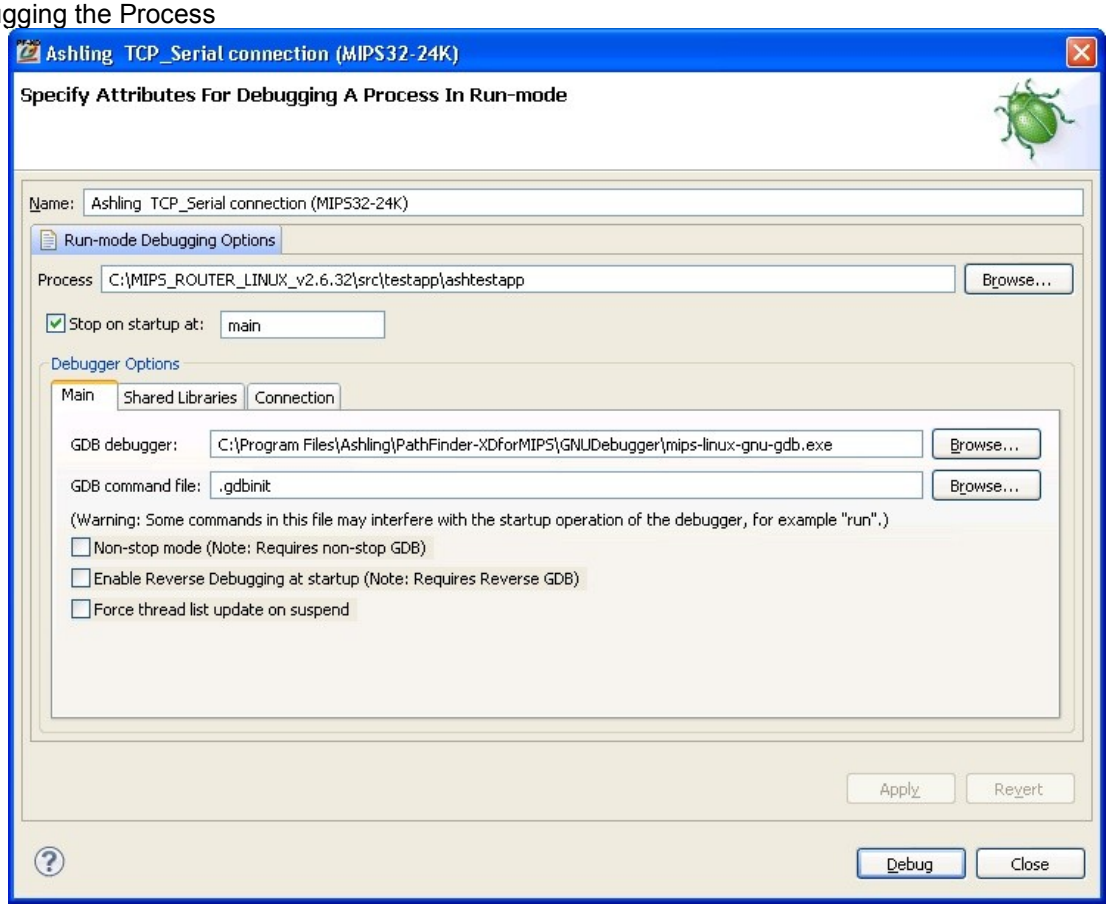


Figure 63. Debugging a Process in Run-mode dialog

6. PathFinder-XD will now update as follows:

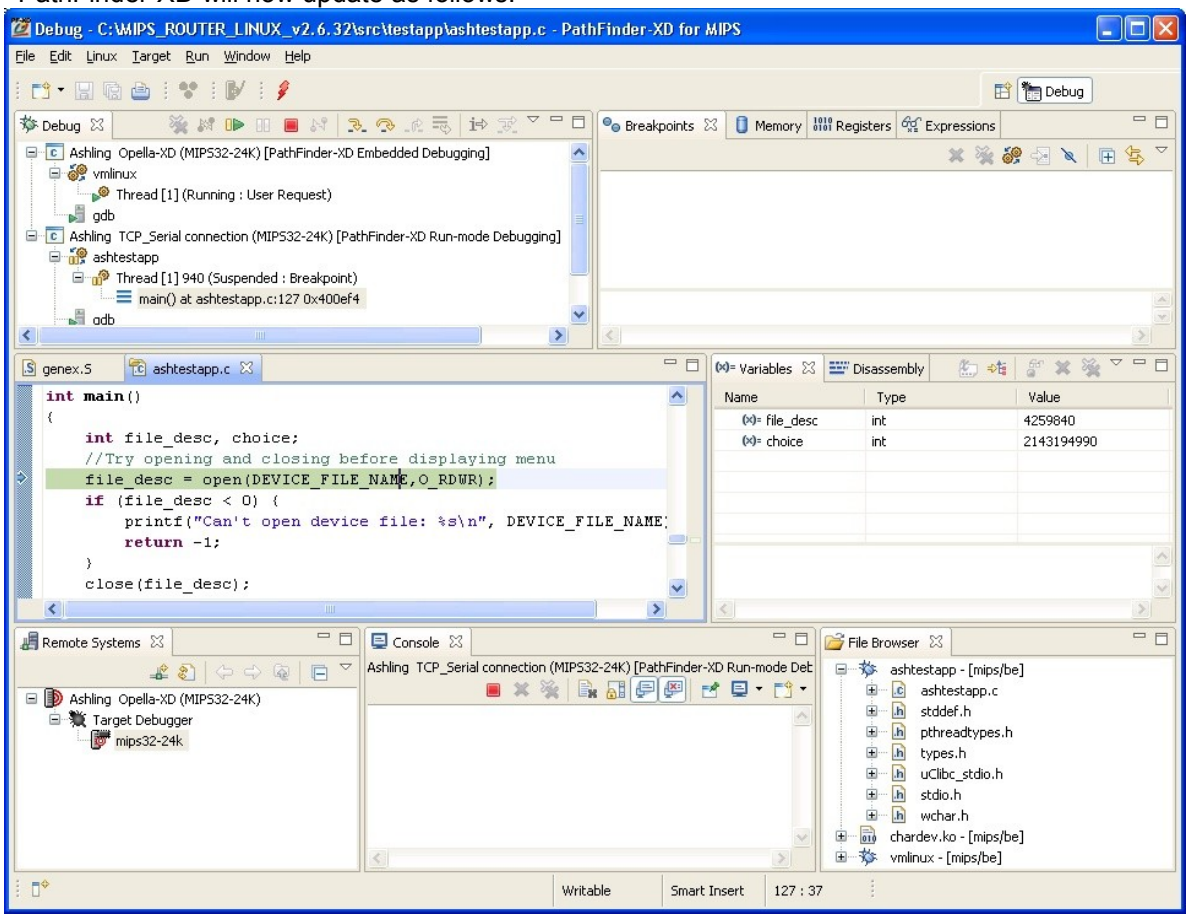


Figure 64. PathFinder-XD in Run-mode

Notice how:

- The **Debug** window show both the Kernel (Embedded Debugging) and Process (Run-mode Debugging) status:

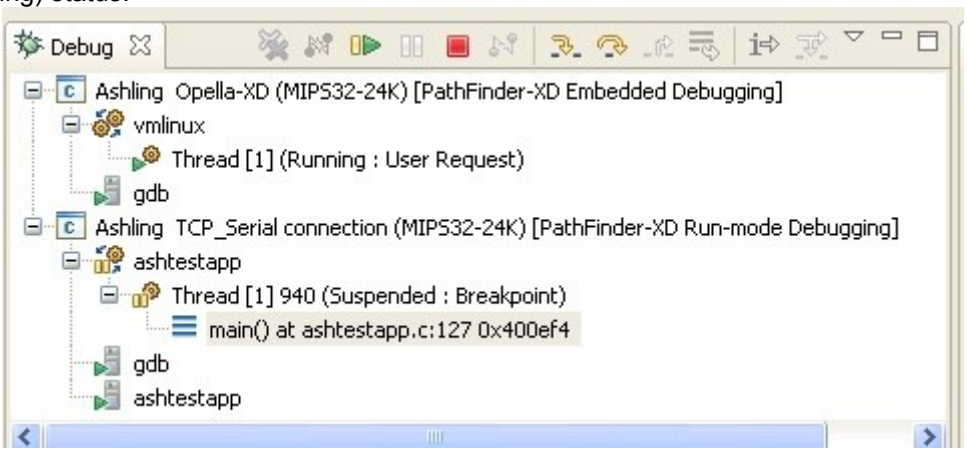


Figure 65. PathFinder-XD Debug Window showing Kernel and Process (Kernel Run-mode) status

- The **File Browser** shows the Module, Process and Kernel sources:

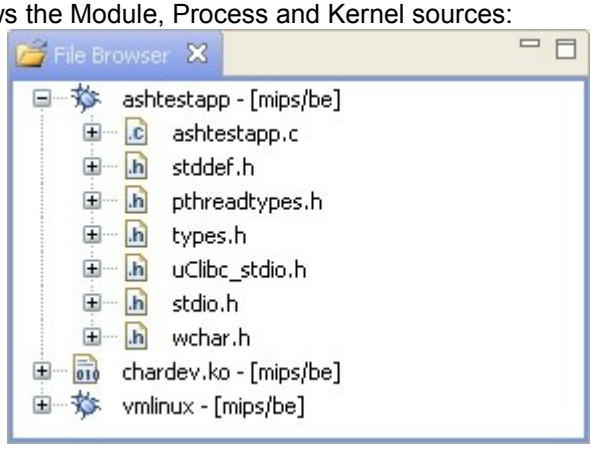


Figure 66. PathFinder-XD File Browser showing Module, Process and Kernel sources

- The **Source** window shows the source code for our **Process** from `main()`

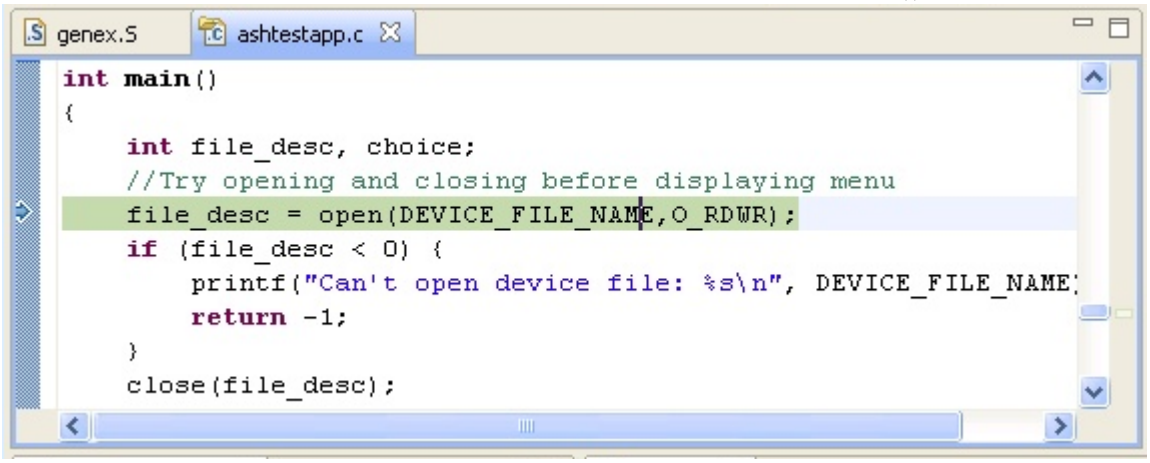


Figure 67. Process Source

7. We can now debug our Process as normal with the Kernel running in the background. When our Process calls functions located in the Module which have a breakpoint set then the Module/Kernel will halt and PathFinder-XD's Debug window will update as follows:

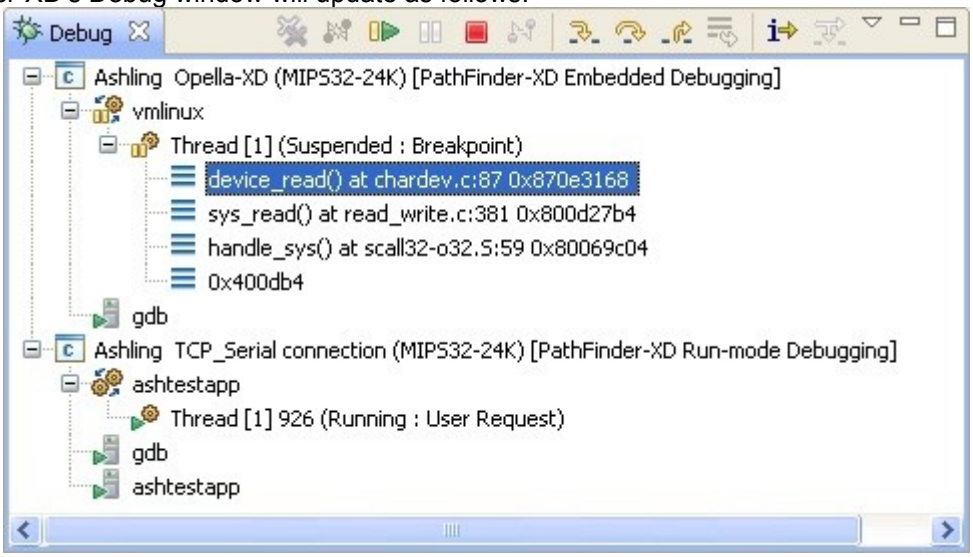


Figure 68. PathFinder-XD Debug Window showing the Kernel halted

Notice how the Kernel is now shown as halted (i.e. PathFinder-XD has automatically switched from run-mode to stop-mode as the kernel is halted due to the breakpoint in the Module). This demonstrates how PathFinder-XD easily switches between stop-mode and run-mode within the same debug session.

4.3.2.2 Debugging multi-threaded applications

Multi-threaded applications are supported in run-mode debugging. All the application threads and the associated Call Stack for each thread are listed. In addition, it is possible to set thread specific breakpoints.

To debug a multi-threaded application in run-mode:

1. Launch gdbserver on the target (i.e. in the Linux shell) specifying the application we wish to debug (`threadtestapp`). Notice how we tell `./gdbserver` which port to listen on (1234 in the below example)

```
root@OpenWrt:~# gdbserver :1234 ./threadtestapp
Process ./threadtestapp created; pid = 933
Listening on port 1234
```

Figure 69. Launching multi-threaded Process

2. Choose **Debug A Process in Run-mode** from PathFinder-XD (the Kernel is now running) as follows:

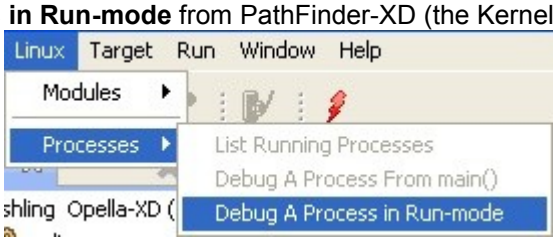
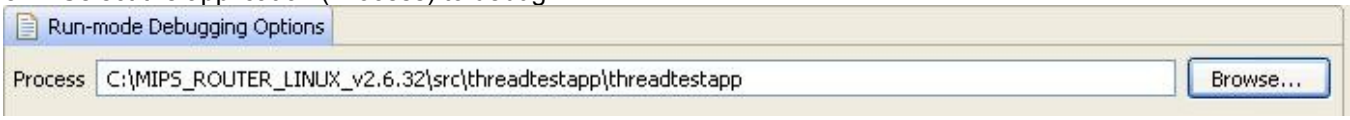


Figure 70. Debugging a Process in Run-mode

3. Select the application (Process) to debug



4. Enable **Non-stop mode** and **Force thread list update on suspend**. This allows debug of a single thread while other threads are running:

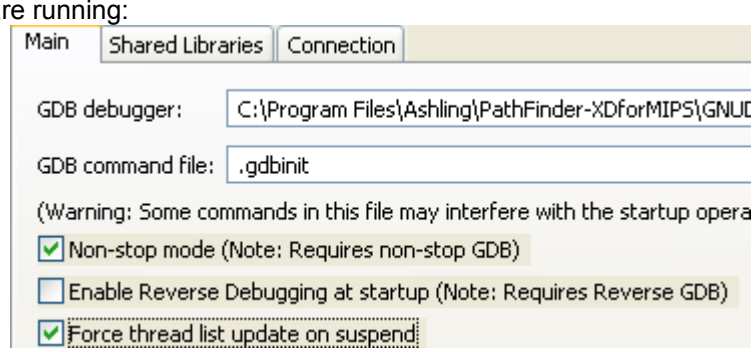


Figure 71. Choosing Non-stop mode

5. Set the location of shared libraries:

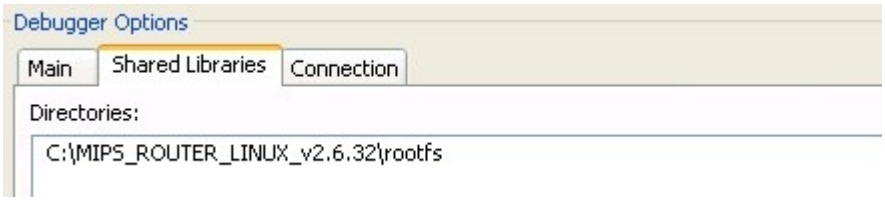


Figure 72. Specifying the Shared Library location

6. And finally, the connection mechanism (TCP in our example), the IP address of the target system (i.e. the RouterStation Pro at 192.168.1.1 which is running gdbserver on port 1234):

Figure 73. Specifying the Connection mechanism

7. Now press **Debug** to start debugging the Process

8. The program will now run to the `main()` function.

9. Put breakpoints in the following locations:
   - `threadfin()` function which is called by each thread

```
void* threadfn(void* tr_data)
{
    while(1)
    {
    // print thread and count
    printf("%s [%08lX]\n",((thread_data *)
    usleep(500);
```

Figure 74. Setting a breakpoint in `threadfin()`

   - `main()` thread

```
while (1)
  {
  printf("In main [%08lX]\n",main_count++);
  usleep(500);
  }
```

Figure 75. Setting a breakpoint in `main()`

10. Now run . When the target halts the Debug view is updated with thread information:

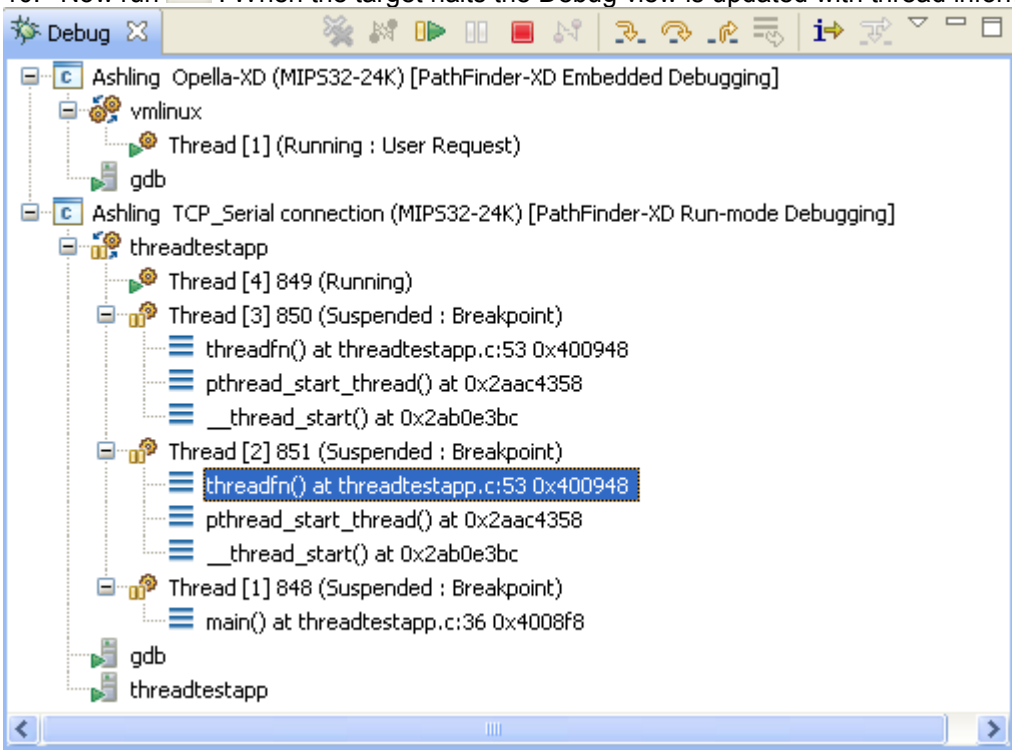


Figure 76. Multi-threaded Debug View

Threads are shown as follows:

Thread [4]	the thread manager
Thread [3]	application thread
Thread [2]	application thread
Thread [1]	main thread

Note that thread numbering order may vary for different debug sessions and clicking on each thread in the above Debug View will update PathFinder's windows to that thread's context.

11. To make a breakpoint thread specific, right-click on the breakpoint (in the ruler i.e. the left-most column of the source-window in `threadfin())` and choose **Breakpoint Properties**

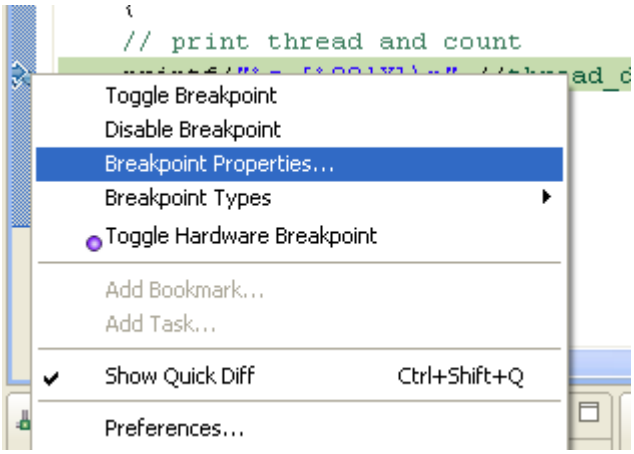


Figure 77. Selecting Breakpoint Properties

- In the **Filter**, check the threads you wish to associate with the breakpoint (Thread[3] in our example)

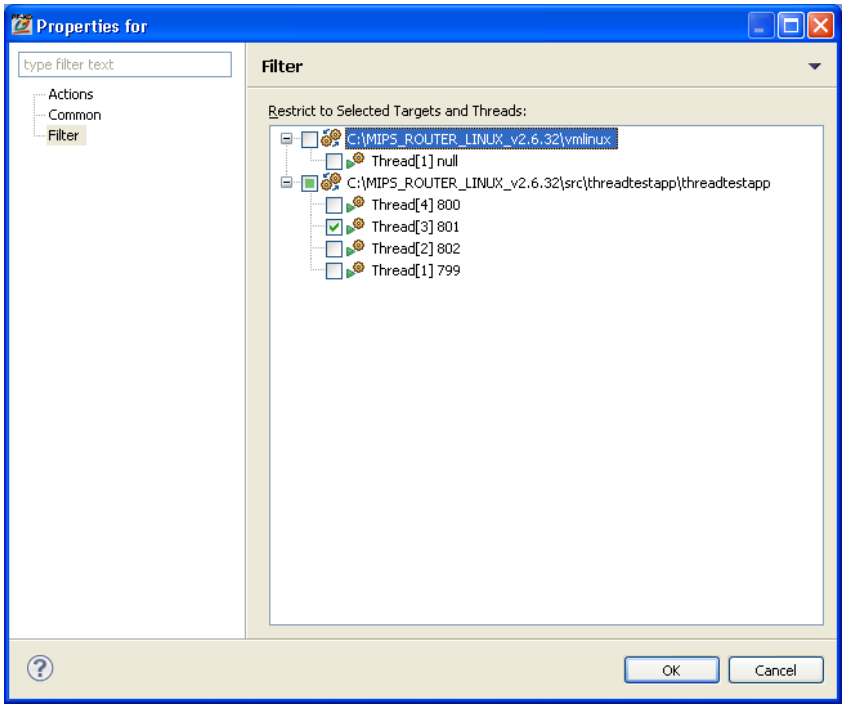


Figure 78. Breakpoint Properties Dialog

- Click OK to set the breakpoint. Now re-run the program and notice that the breakpoint is only taken for Thread[3].

4.3.2.3 Debugging more than one application at the same time

To debug more than one application or process at a time, you must launch a separate gdbserver with a unique port number for each process you wish to debug.

1. Run the first process as per the previous section **4.3.2.1 Sample Run-mode Linux Debugging Session**

2. The RouterStation Pro uses a Busybox (limited functionality/memory foot-print) shell by default which does not support running processes in the background (with "&"), hence, we need to connect to the RouterStation Pro using Telnet to open another shell to run and debug a second process. This can be done from **putty** as follows:
   - Open a new connection to the RouterStation Pro (at 192.168.1.1:23) using the **Telnet** Connection type:

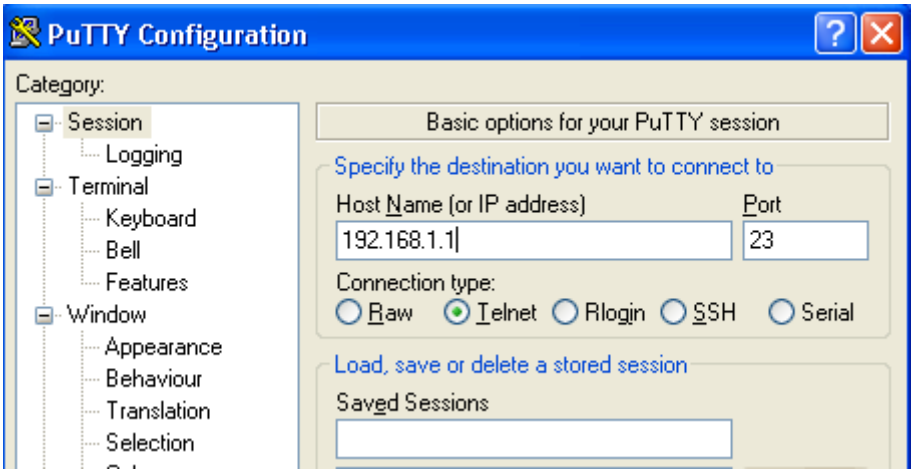


Figure 79. Opening a shell using Telnet

3. Now run gdbserver (on port 2345) and the `threadtestapp` as follows:

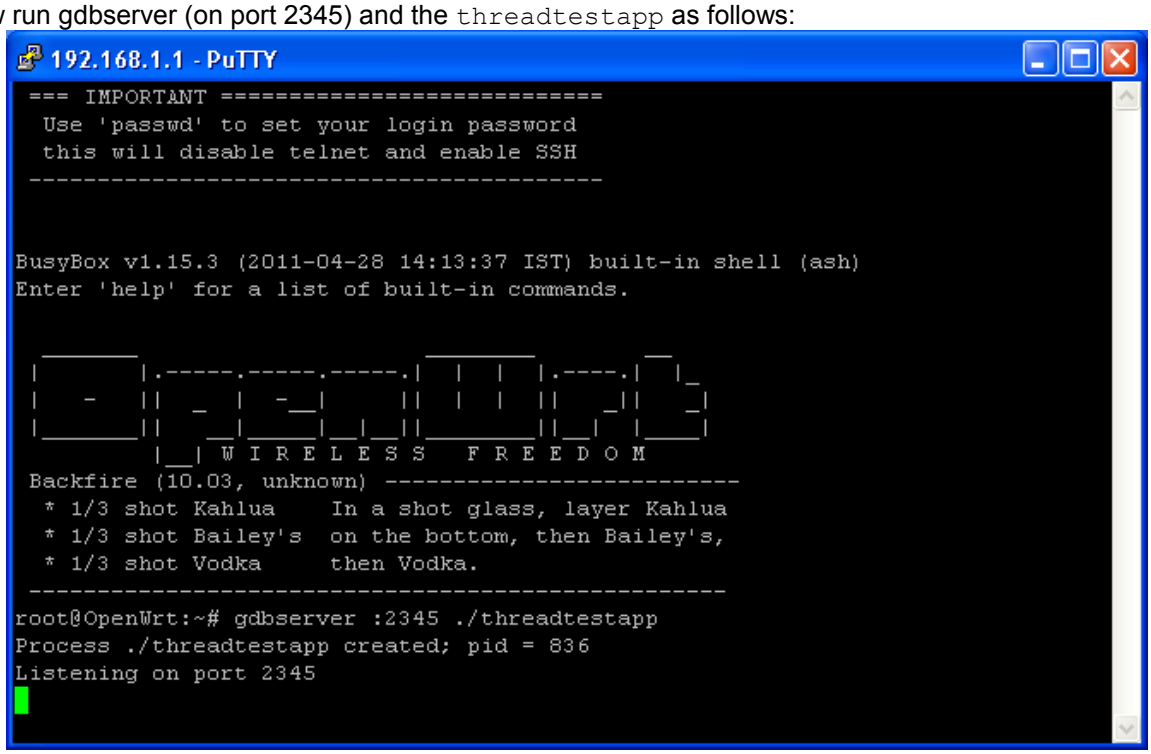


Figure 80. Running a second process via Telnet

4. You can now connect to each process via the **Debug A Process in Run-mode** menu (making sure you use the correct port-number e.g. 2345). The **Debug View** will show each process as follows:

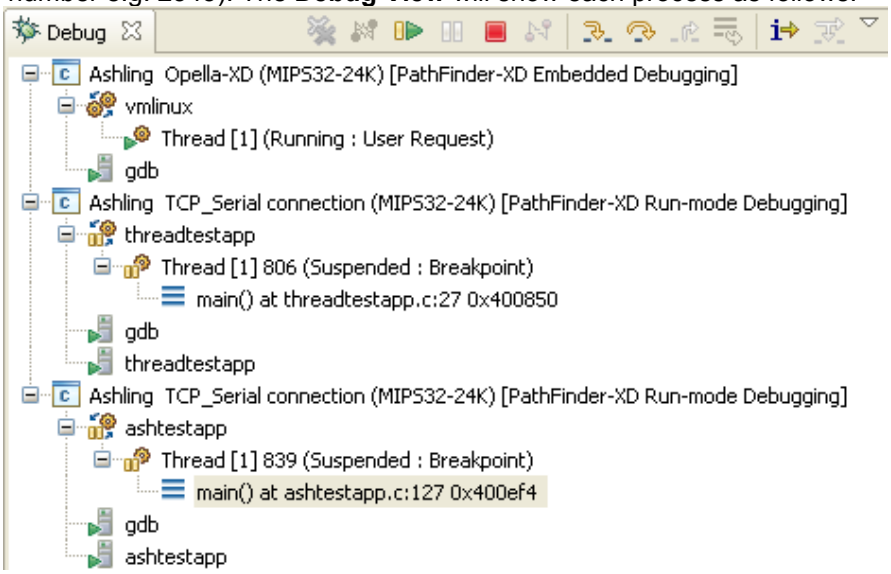


Figure 81. Debugging Multiple Processes

These examples show the power of PathFinder-XD's Embedded Linux support, in particular, the ability to debug Processes whilst the Kernel is running (Run-mode) and to debug the interaction between Processes and the Kernel (including Kernel modules). We hope you like it! Please send your feedback to hugh.okeeffe@nestgroup.net

Doc: APB207-PF-XD_MIPS, Hugh O'Keeffe and Suresh PC, Ashling Microsystems