Ashling Product Brief APB211

v.1.0.5, 15th April 2011

PathFinder-XD for MIPS™ Powered Devices Simulator

Contents

1. Introduction 2
2. Installation 2
2.1 Windows™ Installation 2
2.2 Linux Installation 2
3. Using PathFinder-XD with the QEMU simulator 3
4. Embedded Linux Debugging with PathFinder-XD and the QEMU Simulator 10
4.1 Preparing for debugging 10
4.1.1 Building with debug symbols 10
4.1.2 Compiler optimisations 10
4.1.3 On-demand paging 11
4.2 Stop-mode Debugging 11
4.2.1 Sample Stop-mode Linux Debugging Session 11
4.2.1.1 Invoking PathFinder-XD and connecting to the QEMU simulator 11
4.2.1.2 Loading kernel symbol information to PathFinder-XD and executing the kernel image 12
4.2.1.3 Debug a module from `init_module()` 16
4.2.1.4 Debugging a process from `main()` 18
4.2.1.5 Debugging a running process 19
4.2.1.6 Library debugging 20
4.3 Run-mode Debugging 20
4.3.1 Preparing for Run-mode Debugging 20
4.3.1.1 Setting up networking between host and QEMU simulator 20
4.3.1.2 Modify QEMU invocation parameters 21
4.3.2 Sample Run-mode Linux Debugging Session 21
4.3.2.1 Debugging the Module and Process 22
4.3.2.2 Debugging multi-threaded applications 28
4.3.2.3 Debugging more than one application at the same time 31

1. Introduction

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.

Figure 1. PathFinder-XD for MIPS™

PathFinder-XD supports both "bare-metal" (no target operating system) and Embedded Linux based debugging. This application note introduces PathFinder-XD Simulator version and covers using PathFinder-XD with the QEMU Simulator.

2. Installation

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

Please note: If you have PathFinder-XD already installed then please uninstall it before proceeding.

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
- Ubuntu 08.04 LTS 32-bit/64-bit (requires a specific version of PathFinder-XD)

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 QEMU simulator

In this section we will look at using PathFinder-XD with the QEMU simulator. Ashling provide a binary (executable) version of the QEMU simulator built for the MIPS architecture installed and ready to run. To use QEMU with PathFinder-XD complete the follow the following steps:

1. Run PathFinder-XD

2. PathFinder-XD will then load as follows:

Figure 2. PathFinder-XD for MIPS™ loading

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.

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

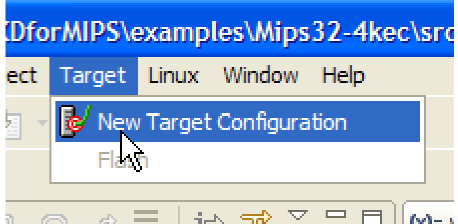


Figure 3. Target Configuration

and select the **Debug using Simulator (QEMU)** option as shown below

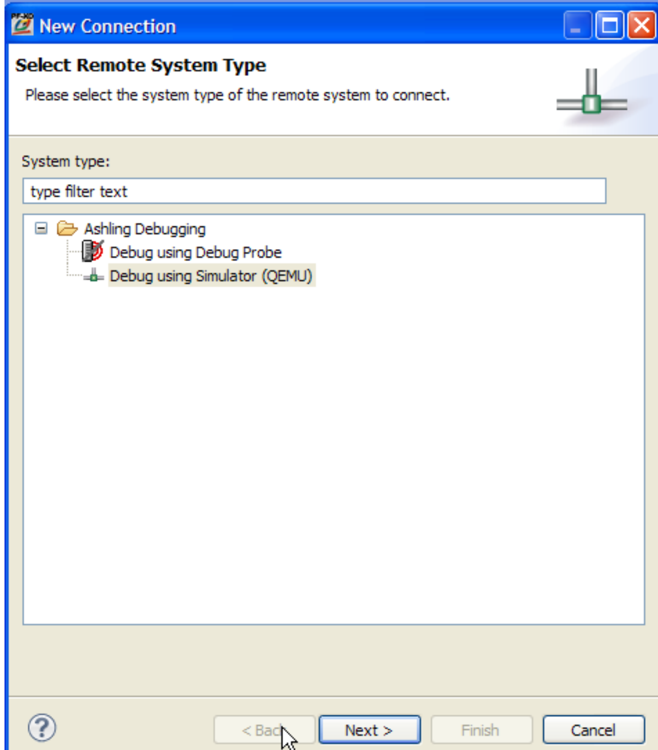


Figure 4. Selecting Remote System Type

4. Click **Next** and we can now configure our QEMU settings as shown below:

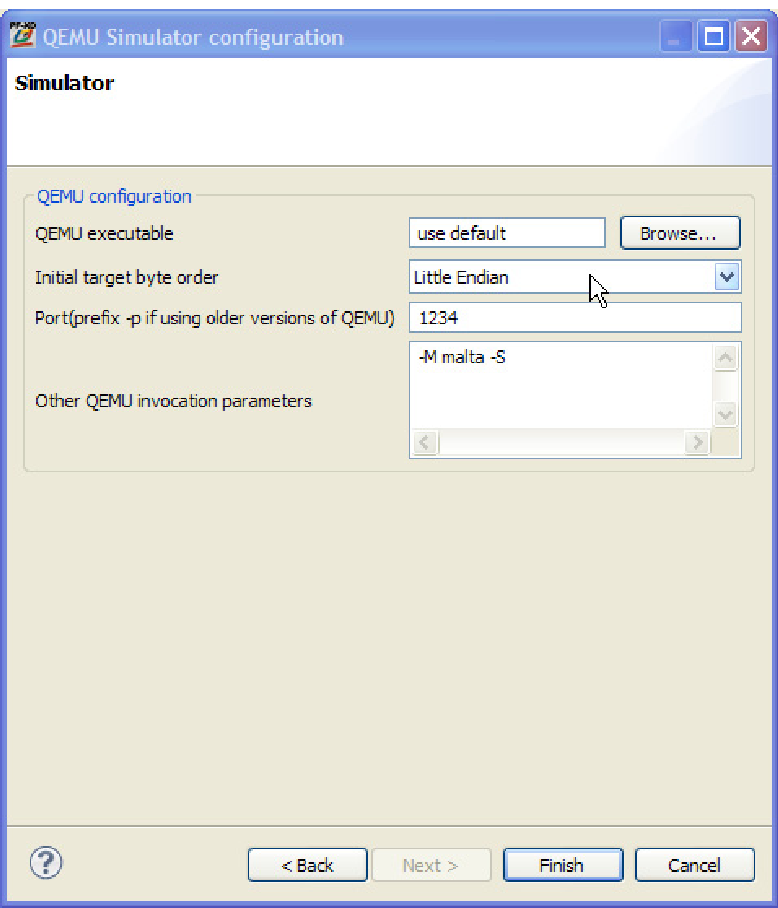


Figure 5. QEMU Simulator Configuration

Settings include:

- **QEMU executable:** The actual binary file executed by PathFinder-XD when invoking QEMU. Usually not changed
- **Initial target byte order:** Required endianness

- **Port:** The TCP/IP port used by PathFinder-XD to communicate with QEMU. Usually not changed
- **QEMU Invocation parameters:** Allows you to configure QEMU. For more details on QEMU configuration, see the following links:
  http://wiki.qemu.org/download/qemu-doc.html#sec_005finvocation
  http://wiki.qemu.org/download/qemu-doc.html#MIPS-System-emulator

Click Finish when done.

5. PathFinder-XD will now create a new QEMU Simulator setting in it's **Remote Systems** Window as shown below:

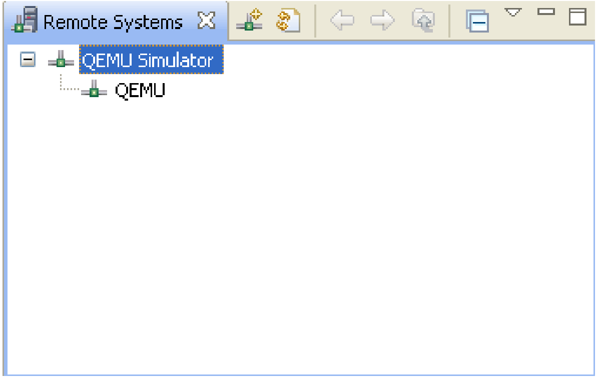


Figure 6. Remote Systems Window

Right-click on **QEMU** and click **Connect to** invoke QEMU. Once invoked, the **Remote Systems** window will update as follows:

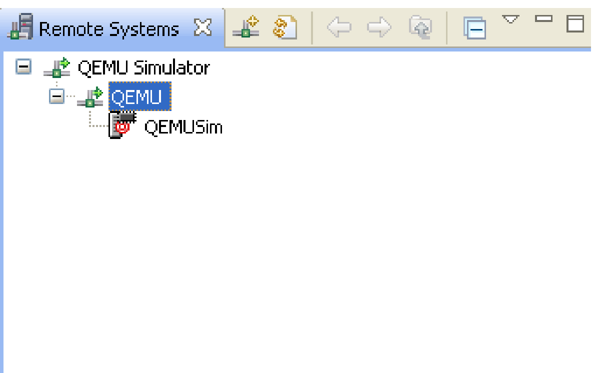


Figure 7. Remote Systems Window after QEMU connection

6. We can now download a program to QEMU by right-clicking over **QEMUSim** and selecting **Download and Launch**

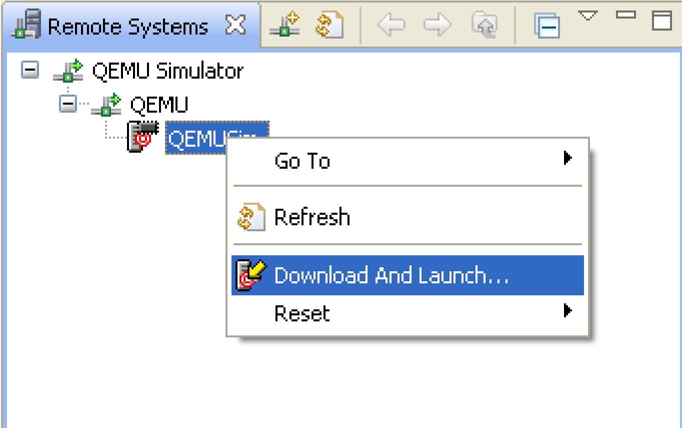


Figure 8. Download and Launch

PathFinder-XD includes a suitable program (`C:\Program Files\Ashling\PFXDMIPSIM\examples\Mips32-4kec\Debug-LE\UnCached_LE.elf`) for running with QEMU.

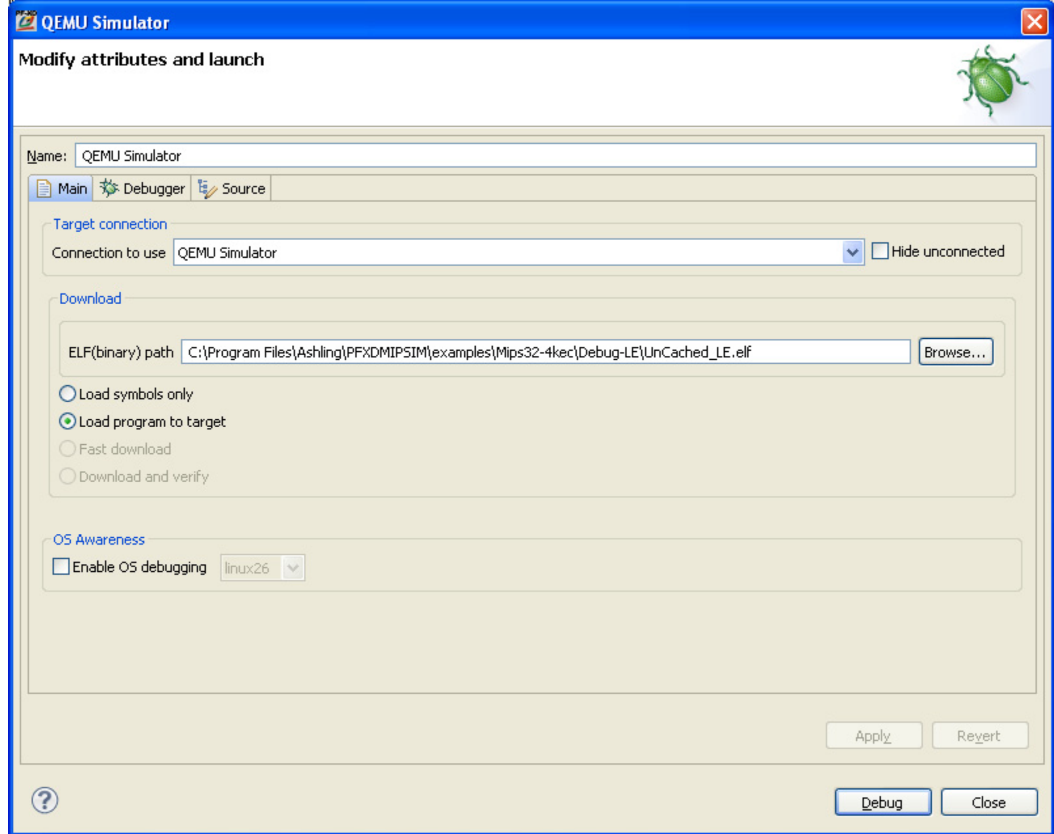


Figure 9. Specifying Target Program to Download

In the **Debugger** tab you can tell PathFinder-XD to stop at `main()` as shown below

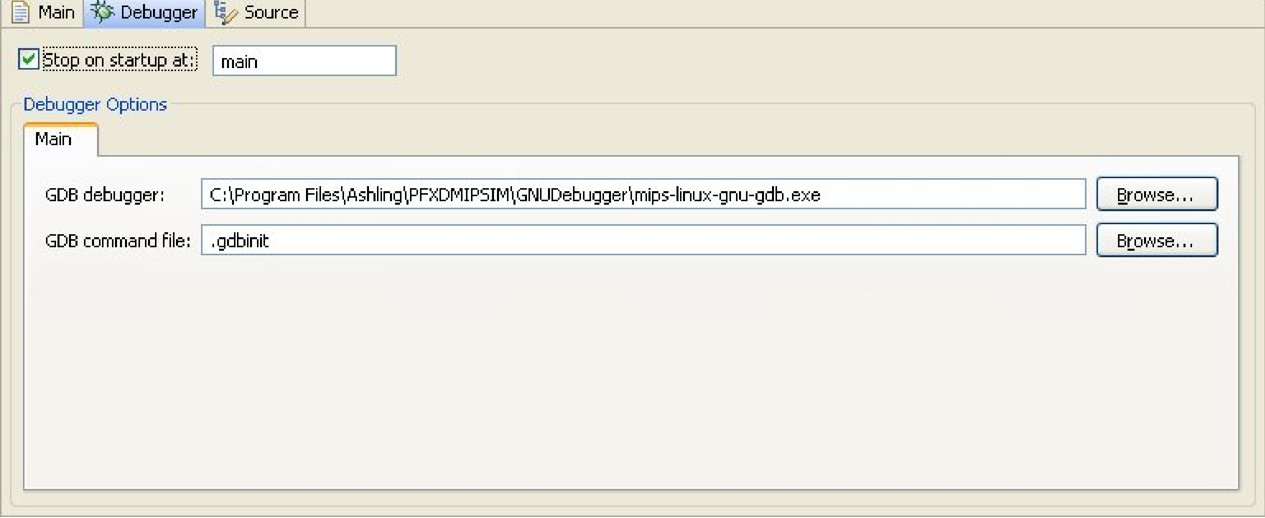


Figure 10. Stopping at `main()`

When finished, press **Debug** to load your program.

7. PathFinder-XD will now download the program to QEMU and update it's Windows as follows allowing you to start your Debug session:

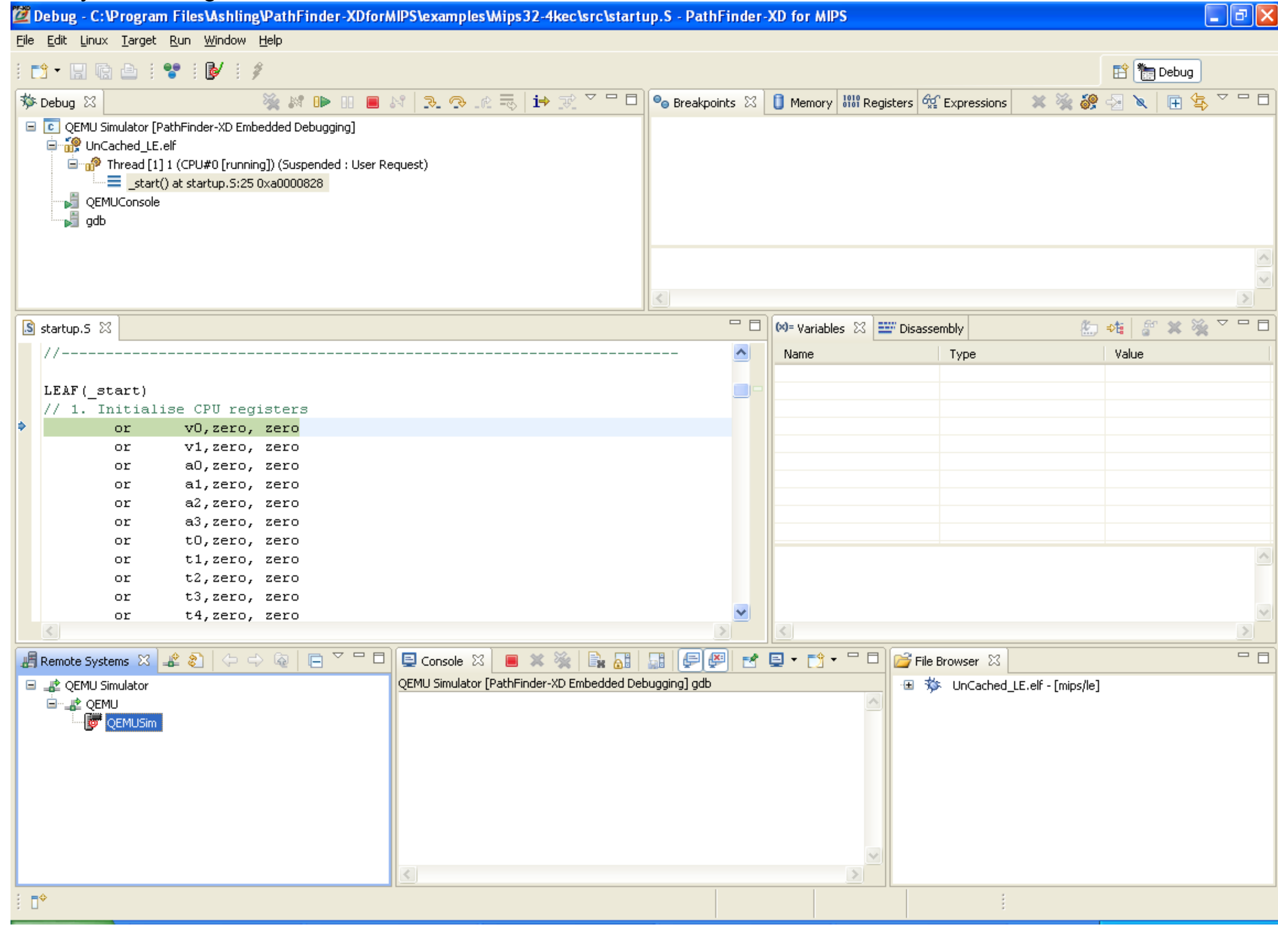
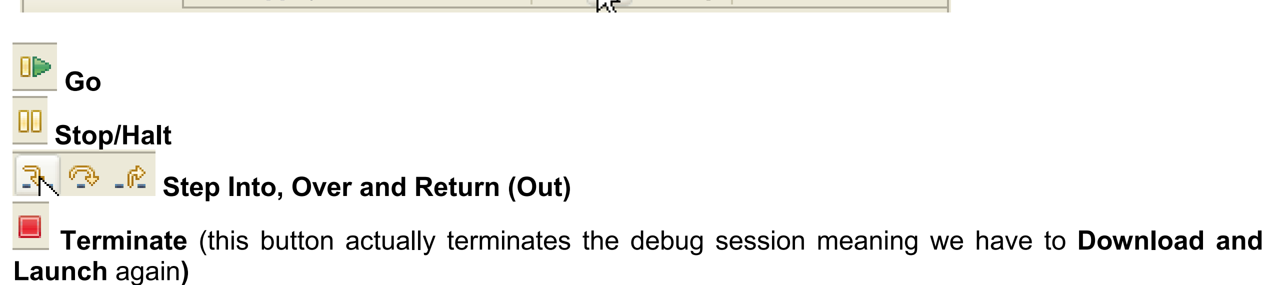


Figure 11. PathFinder-XD after program download

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



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 12. 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:

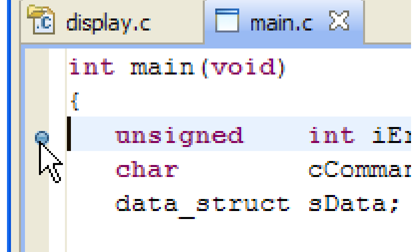


Figure 12. Setting a breakpoint

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

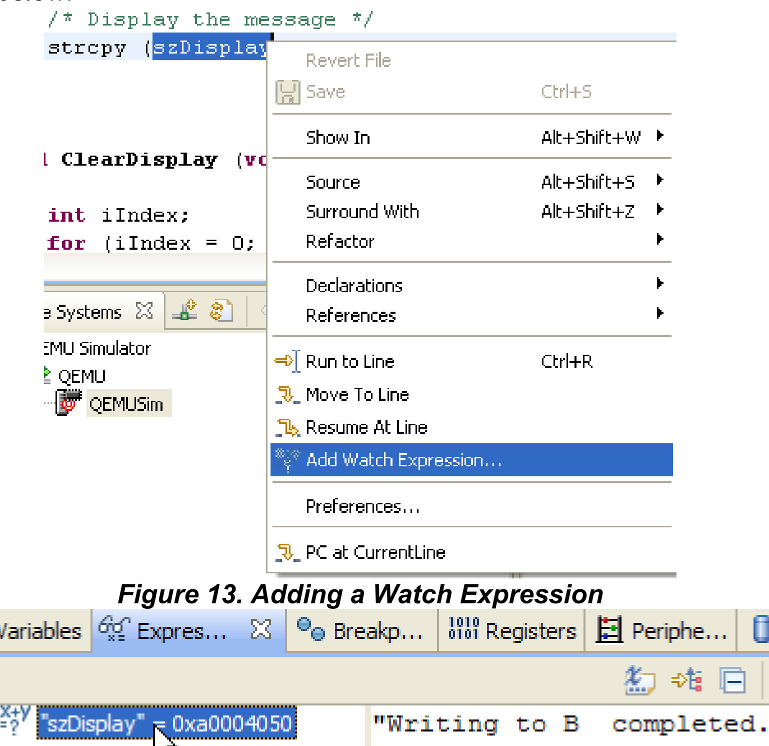


Figure 13. Adding a Watch Expression

Figure 14. Expression window showing watched expression

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

Figure 15. Quick watch via mouse hover

10. PathFinder-XD supports a Console Windows (or Views) which can be opened from the Window menu:

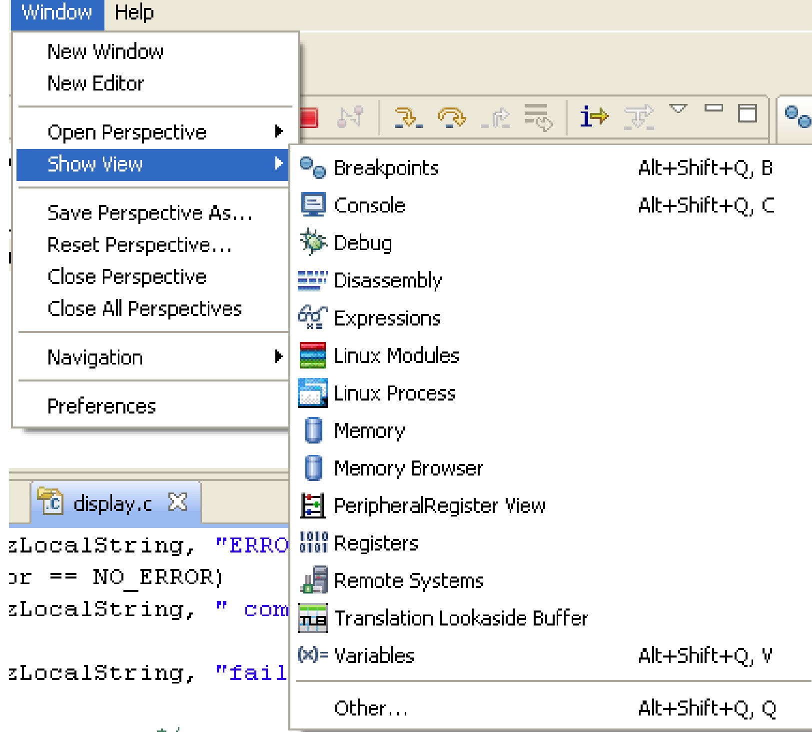


Figure 16. PathFinder-XD Views

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

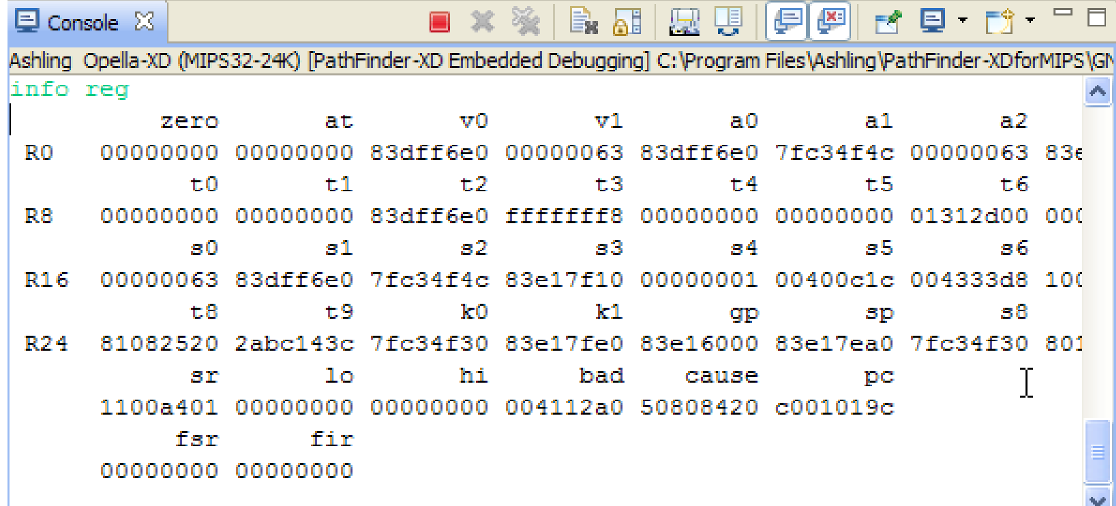


Figure 17. PathFinder-XD Console showing the output of the info reg command

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.

4. Embedded Linux Debugging with PathFinder-XD and the QEMU Simulator

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

- Stop-mode: In stop-mode debugging, the whole system is halted (e.g. kernel and applications) whenever a breakpoint is taken.
- Run-mode: Run-mode debugging requires an application (GDB server) running on the QEMU simulator. 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 allows debugging 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.

This section demonstrates Linux Kernel Debugging using PathFinder-XD and the QEMU simulator running v2.6.32 of the Linux Kernel on a Windows host machine. A suitable, pre-built version of the v2.6.32 Linux Kernel and root file-system is also available for download from Ashling (`http://www.ashling.com/support/MIPS/QEMU_LINUX/MIPS_QEMU_LINUX_v2.6.32.ZIP`); this should be installed by unzipping on to your local hard-disk (ensure you preserve the directory structure as present in the ZIP file).

Ashling recommend a high-spec PC (e.g. dual-core with 4GB memory) for use with this example given the processing-power required to run Linux on the QEMU simulator.

4.1 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.

4.1.1 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.

4.1.2 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
```

4.1.3 On-demand paging

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***.

4.2 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

4.2.1 Sample Stop-mode Linux Debugging Session

This section demonstrates Linux Kernel Debugging using PathFinder-XD and the QEMU simulator running v2.6.32 of the Linux Kernel.

4.2.1.1 Invoking PathFinder-XD and connecting to the QEMU simulator

Invoke PathFinder and configure for the QEMU simulator as shown in section **3**. The **QEMU Invocation parameters** need to be modified to load the Linux kernel as follows:

```
-M malta -kernel <drive>:\<path>\vmlinux -hda <drive>:\<path>\rootfs.ext2 -append "root=/dev/hda rw"
```

Where `<drive>:\<path>\` is the location of your Linux kernel. In our walk-through, we assume this is `c:\MIPS_QEMU_LINUX_v2.6.32\`.

For example:

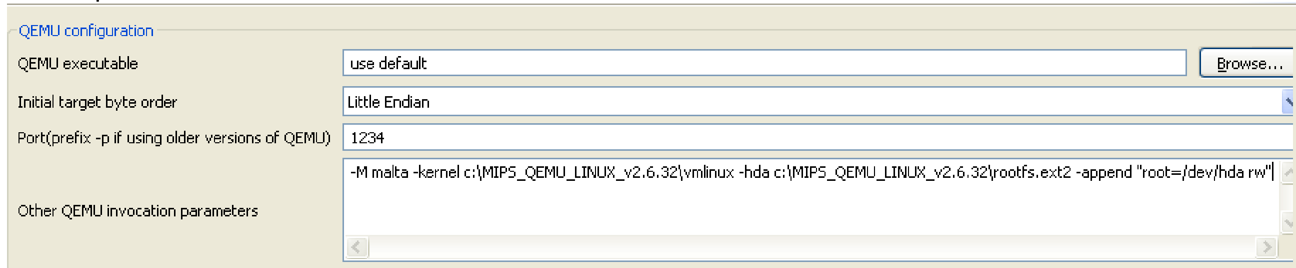


Figure 18. QEMU invocation parameters for Linux debugging

Please note that if you need to debug the kernel from start up, the **"-S"** option should be appended with the invocation parameters as follows:

```
-M malta -kernel <drive>:\<path>\vmlinux -hda <drive>:\<path>\rootfs.ext2 -append "root=/dev/hda rw" -S
```

These invocation parameters ensure the Linux kernel (`vmlinux`) is loaded to the simulator and supplied root file system (`rootfs`) is used. Invocation parameters include:

- `-kernel` This option loads the Linux kernel (`vmlinux`) file to the QEMU simulator
- `-hda` This option mounts a hard disk image (`rootfs.ext2`) for the simulator. This hard disk contains the root file system (`rootfs`) required for the kernel to load.
- `-append` This option passes the kernel arguments.

4.2.1.2 Loading kernel symbol information to PathFinder-XD and executing the kernel image

Connect to the simulator (right-click **QEMU** and select **Connect**), right-click over **QEMUSim** and select **Download and Launch** as follows:

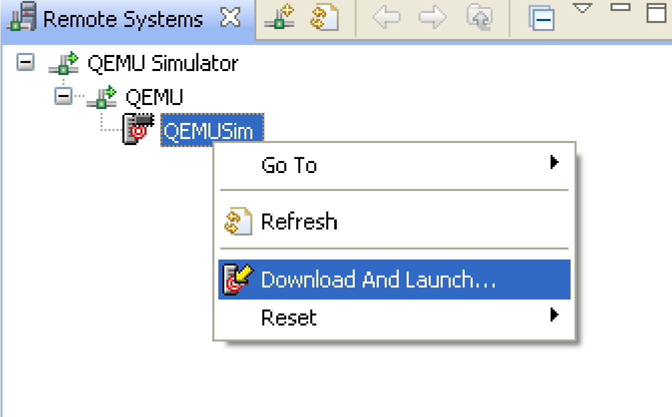


Figure 19. Download and launch

Now, 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 kernel binary image is already downloaded to the simulator (via the QEMU invocation), hence, we only need to select **Load symbols only** (for the kernel image), specify the **ELF(binary) path** (`c:\MIPS_QEMU_LINUX_v2.6.32\vmlinux`) and press **Debug.** This will load the kernel symbols into PathFinder-XD to allow symbolic kernel debug.

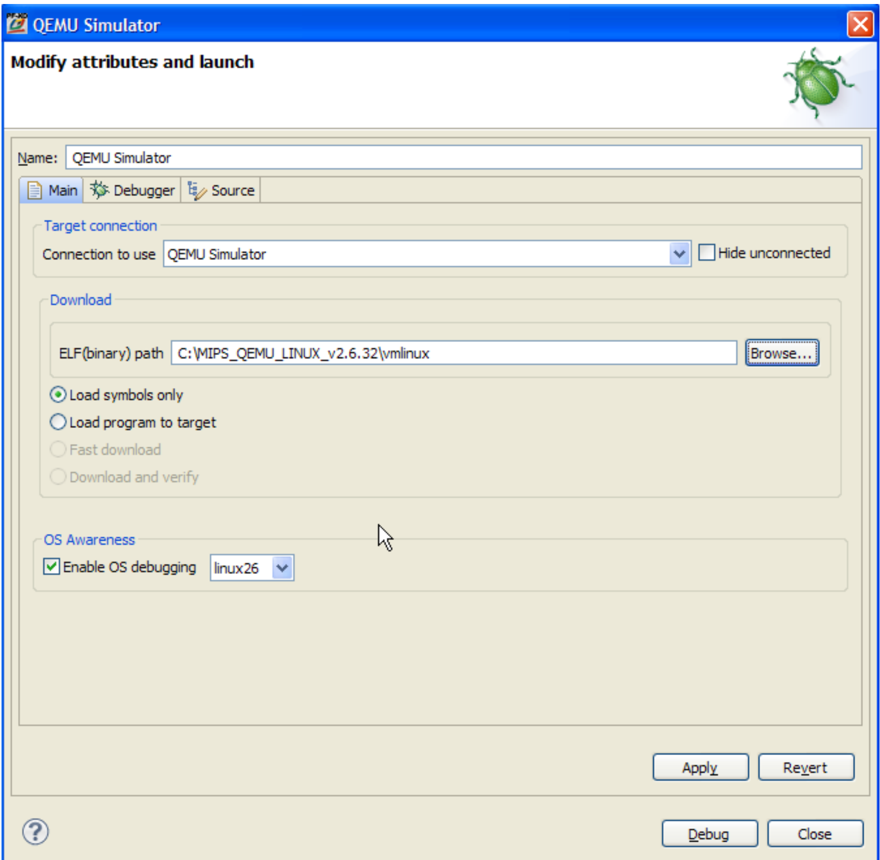


Figure 20. Loading the kernel symbols

Run the simulator using button and the kernel will boot in a seperate QEMU console as shown below (this may take a few minutes depending on your PC speed):

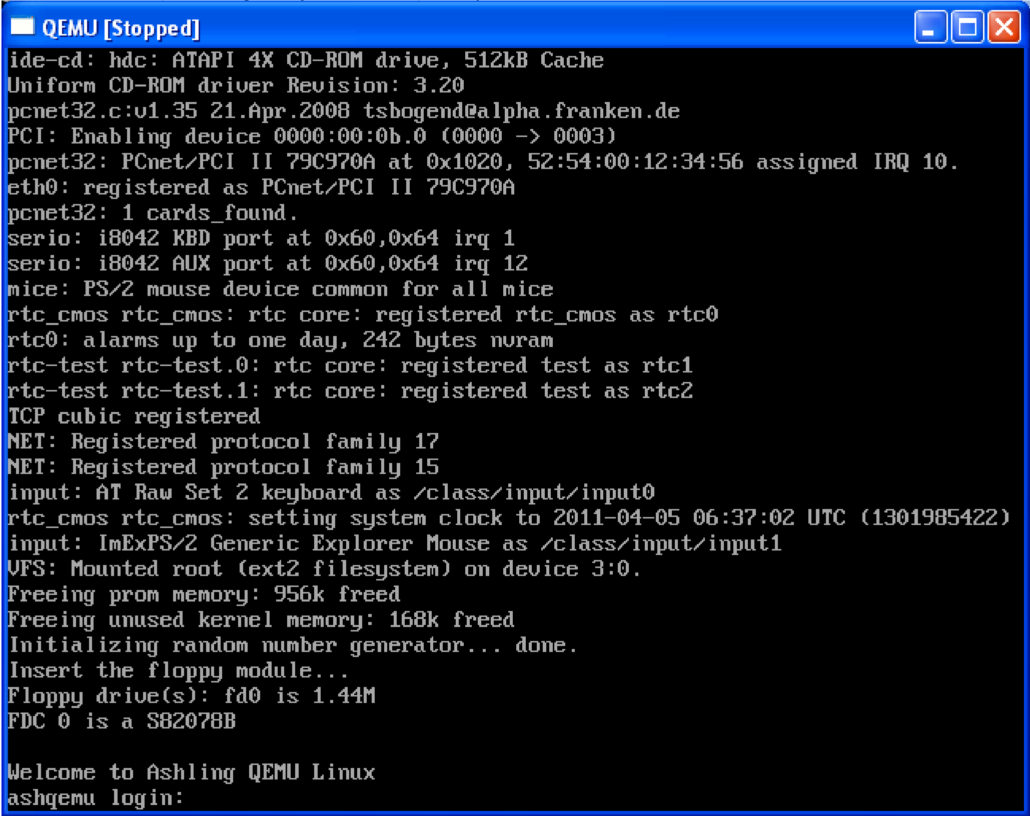


Figure 21. QEMU Linux console

You can login in to the kernel using root (no password required). Once the kernel is booted, we can halt it within PathFinder-XD (by pressing Stop/Halt). PathFinder-XD then updates as follows:

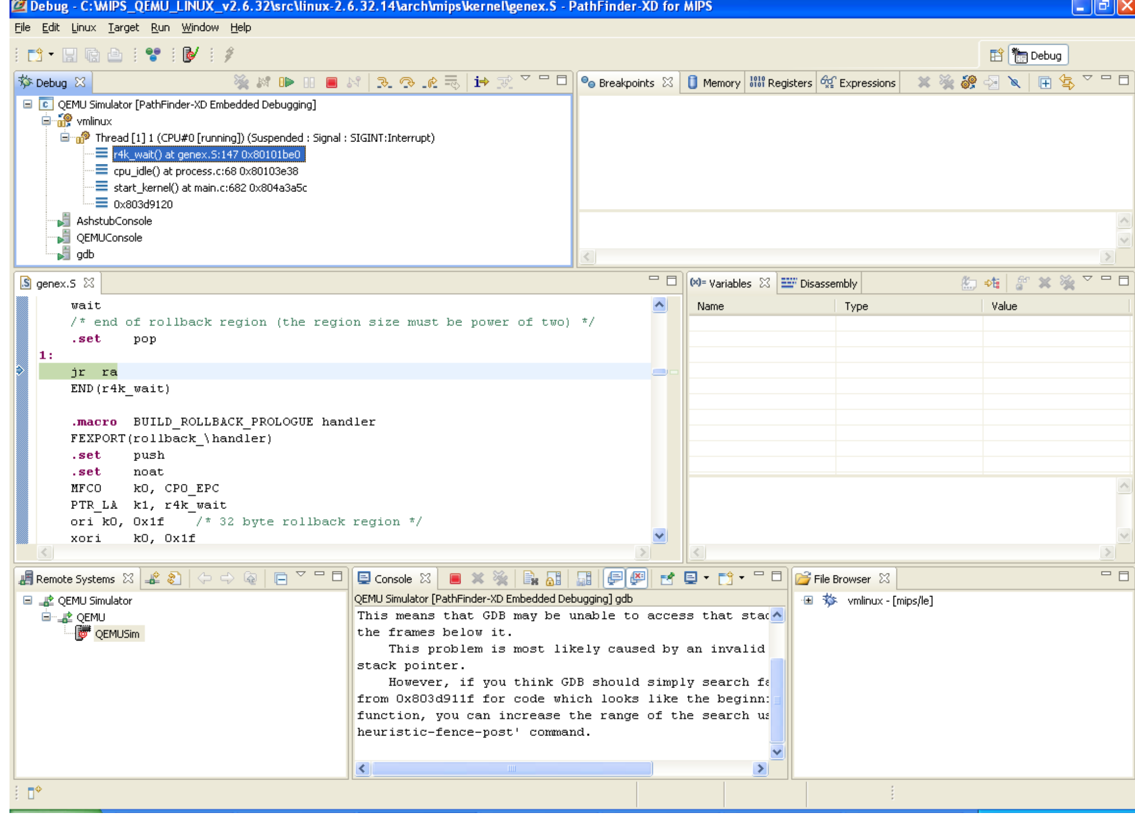


Figure 22. PathFinder-XD after halting the kernel

If no source is shown, then select a function in the **Debug** window

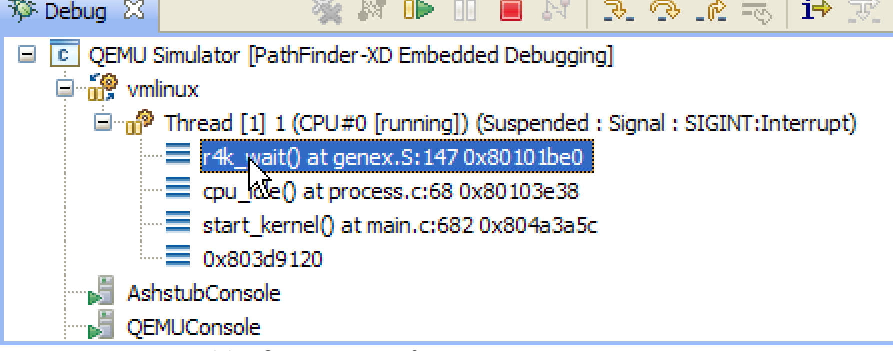


Figure 23. Selecting a function to display it's source

which will open a new tab in the source window allowing you to select **Edit Source Lookup Path…** button.

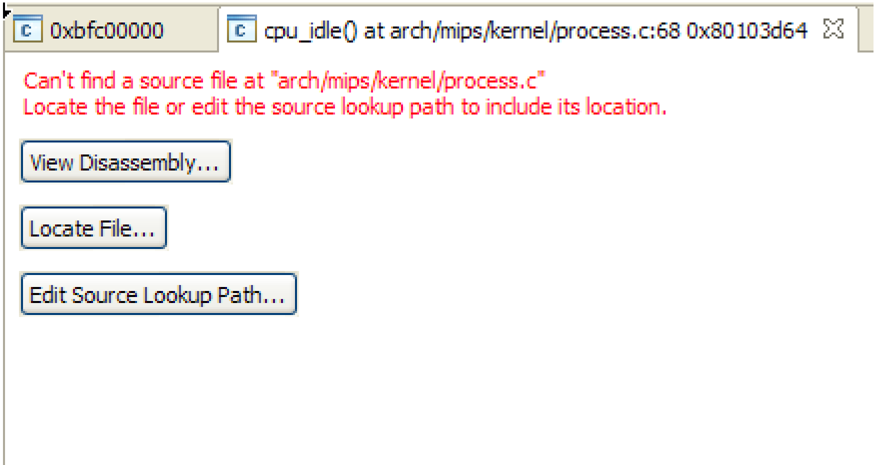


Figure 24. Edit Source Lookup Path

Specify your source-directory as shown below (make sure **Search subfolders** is selected)

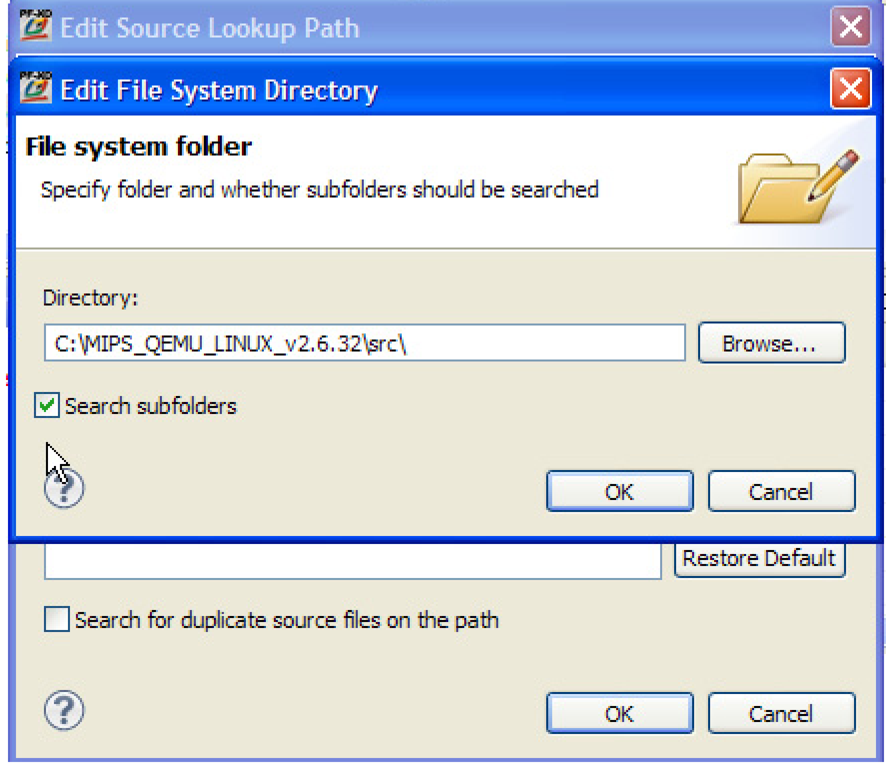


Figure 25. Edit File System Directory

Notice the following additional windows in PathFinder-XD:

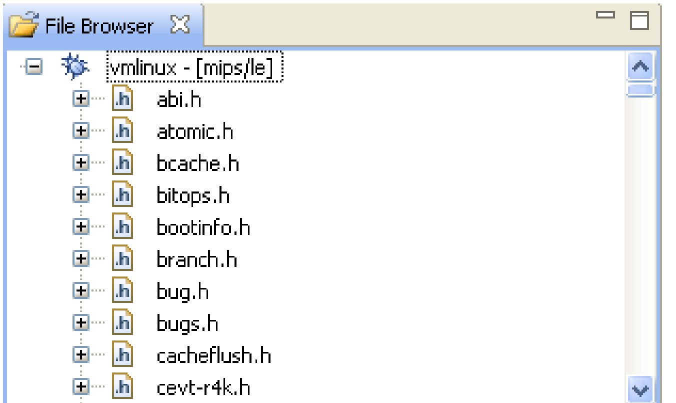
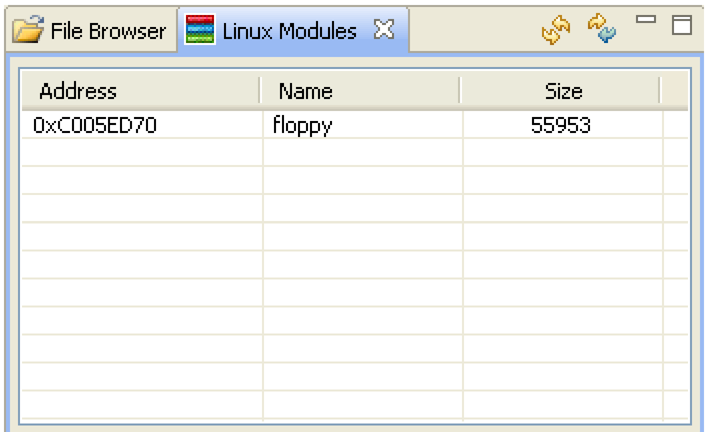
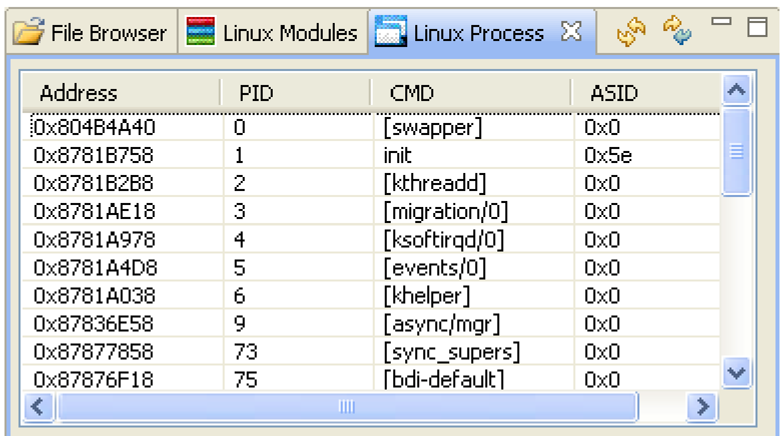


Figure 26. File Browser view showing all kernel source-files



Address	Name	Size
0xC005ED70	floppy	55953

Figure 27. Linux Module view showing all currently loaded kernel modules (enabled via Linux menu)



Address	PID	CMD	ASID
0x804B4A40	0	[swapper]	0x0
0x8781B758	1	init	0x5e
0x8781B2B8	2	[kthreadd]	0x0
0x8781AE18	3	[migration/0]	0x0
0x8781A978	4	[ksoftirqd/0]	0x0
0x8781A4D8	5	[events/0]	0x0
0x8781A038	6	[khelper]	0x0
0x87836E58	9	[async/mgr]	0x0
0x87877858	73	[sync_supers]	0x0
0x87876F18	75	[bdi-default]	0x0

Figure 28. Linux Process view showing all processes (enabled via Linux menu)

Full kernel source-level debug is now possible.

4.2.1.3 Debug a module from `init_module()`

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

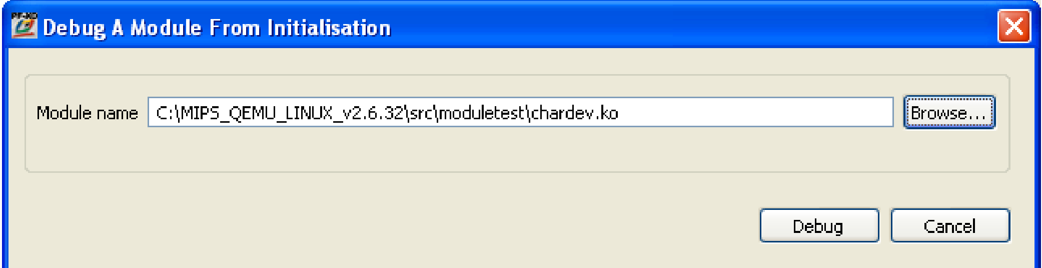
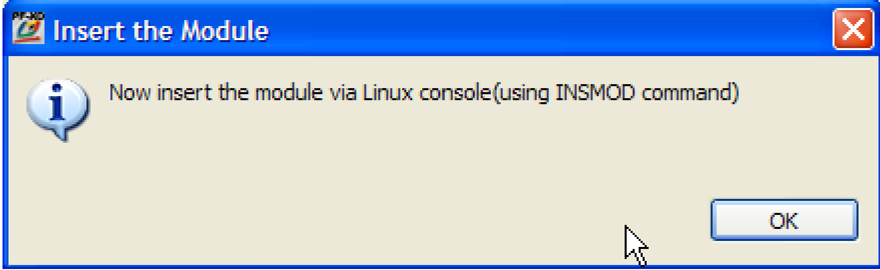


Figure 29. Specifying the module to debug

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



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

```
# insmod chardev.ko
chardev: module license 'unspecified' taints kernel.
Disabling lock debugging due to kernel taint
```

Figure 30. Inserting (running) the module

PathFinder-XD then halts the module at `init_module()` allowing module debug. You may have to specify the location of your source-file as before (`C:\MIPS_QEMU_LINUX_v2.6.32\src\moduletest`)

```
genex.S  notifier.c  chardev.c

/*
 * Initialize the module - Register the character device
 */
int init_module()
{
    int ret_val;
    /*
     * Register the character device (atleast try)
     */
    ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);
    printk("\nretval=%d\n",ret_val);

    /*
     * Negative values signify an error
     */
    if (ret_val < 0) {
```

Figure 31. Module source

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

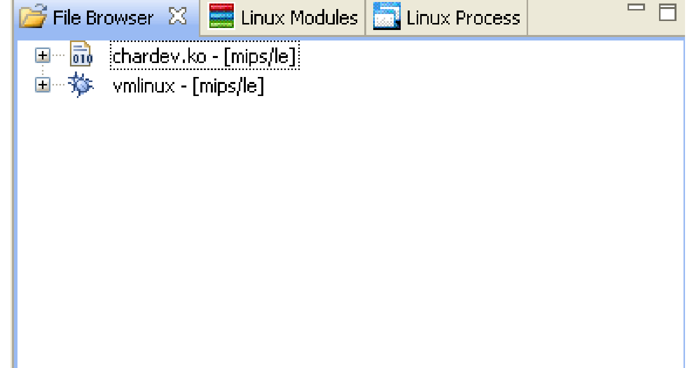


Figure 32. File Browser view showing modules sources

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

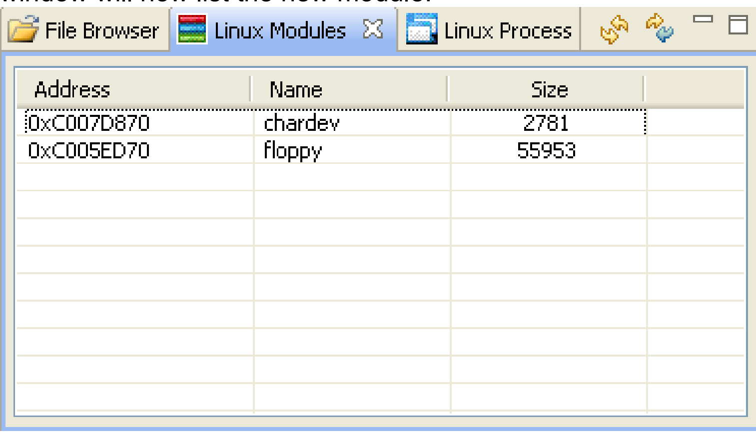
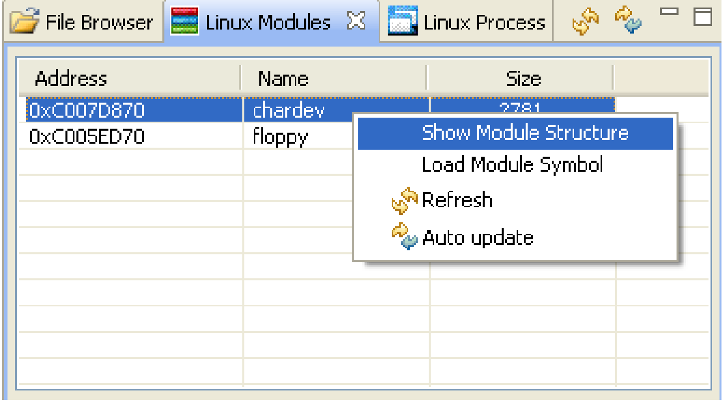


Figure 33. Linux Modules window listing the new module

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



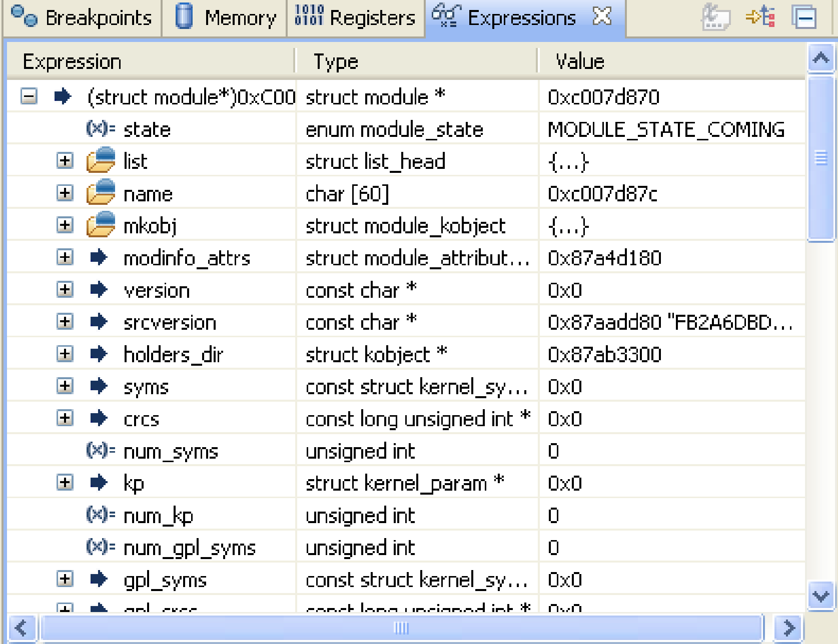


Figure 34. Viewing the internal kernel module structures

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) as follows:

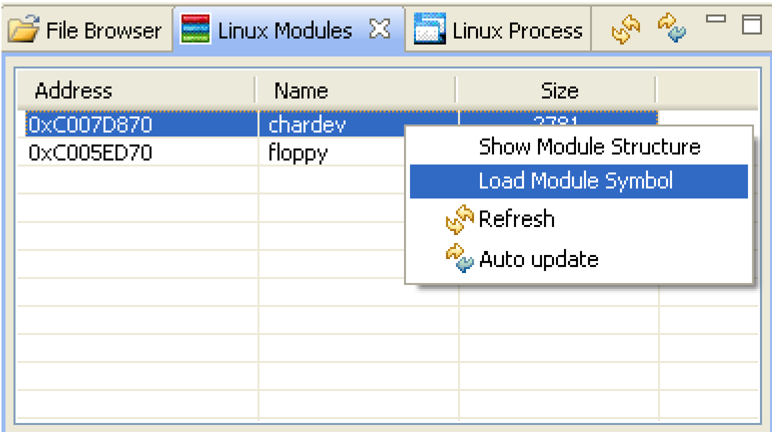


Figure 35. Viewing the internal kernel module structures

4.2.1.4 Debugging a process from `main()`

Use the **Linux|Processes|Debug A Process From `main()`** to debug a process from it's entry point (this feature is only available in stop-mode i.e. when the kernel is halted) as follows:

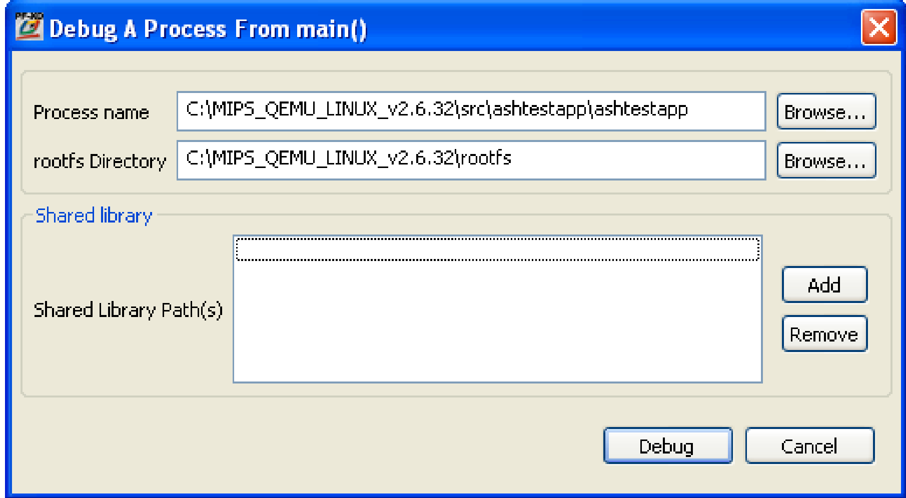
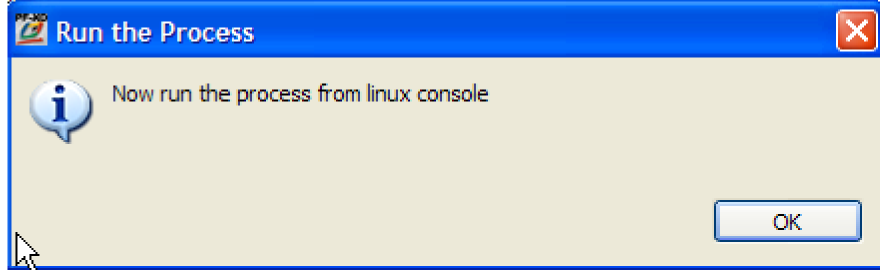


Figure 36. Debugging a process from `main()`

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:

```
# ./ashtestapp
```

Figure 37. Running the process

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

```
genex.S    chardev.c    ashtestapp.c

int main()
{
    int file_desc, choice;
    //Try opening and closing before displaying menu
    file_desc = open(DEVICE_FILE_NAME,O_RDWR);
    if (file_desc < 0) {
        printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
        return -1;
    }
    close(file_desc);
    do
    {
        printf("Menu\n====\n");
        printf("1. Read from device\n2. Write to device\n");
        printf("3. Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
```

Figure 38. PathFinder-XD halted at the process's `main()` function

The **File Browser** window will update to show the process's source-code.

Please note: When using the QEMU simulator it is not possible to list processes whilst debugging a process in stop-mode (e.g. in PathFinder's Linux Process Window); this restriction is not present when debugging a process in run-mode or when using hardware based target debugging (e.g. Opella-XD).

4.2.1.5 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:

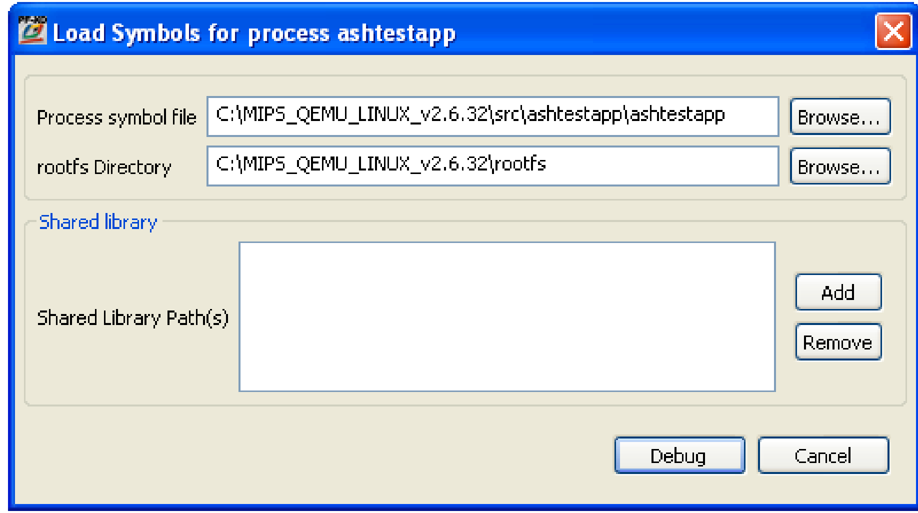


Figure 39. Loading a process's symbols

4.2.1.6 Library debugging

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

4.3 Run-mode Debugging

Run-mode debugging is done via a "simulated" Serial/Ethernet interface and requires an application (GDB server) running on the simulator. 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).

4.3.1 Preparing for Run-mode Debugging

For run-mode debugging, a simulated Ethernet connection needs to be established between the host machine and the simulator (QEMU). This section explains how to configure a networking connection between the host OS and simulator so that run-mode debugging can be done through Ethernet.

4.3.1.1 Setting up networking between host and QEMU simulator

QEMU uses TAP (http://en.wikipedia.org/wiki/TUN/TAP) interfaces to provide full networking capability with the host OS. TAP simulates an Ethernet device entirely in software.

The Windows host OS does not provide a TAP adapter, hence, Windows users will need to download and install a TAP adapter. In our example we will use OpenVPN which can be downloaded from http://openvpn.net/index.php/open-source/downloads.html. Running as administrator, download and install OpenVPN (selecting all options). Once installed, you will have a new Network Adapter (under Network Connections). Configure its IP address via the right-mouse Properties menu as follows:

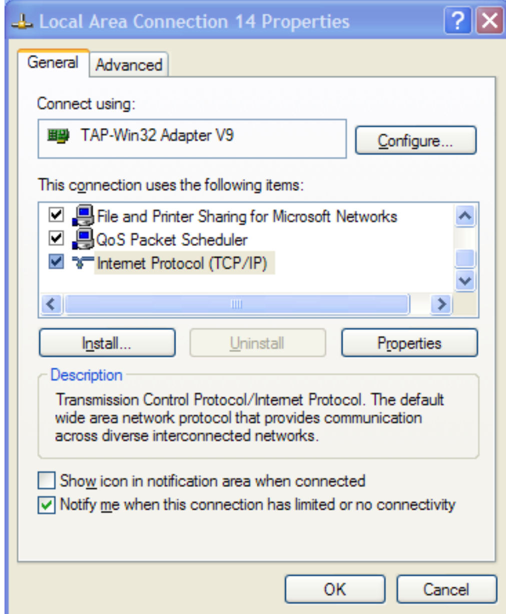


Figure 40. TAP Adapter

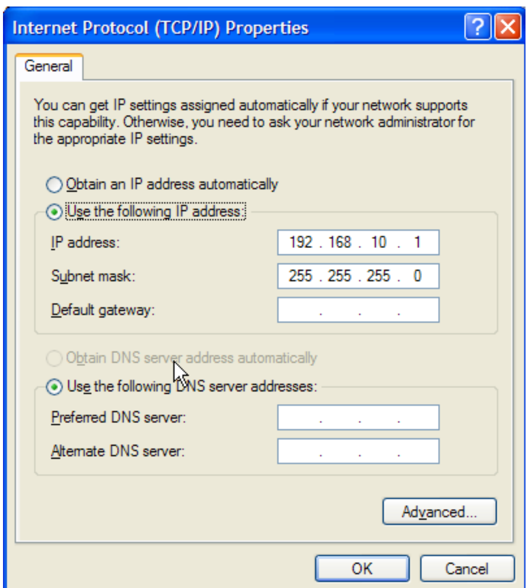


Figure 41. TAP adapter properties

In our example, we will assign the address 192.168.10.1 as shown above. In **Network Connections**, rename (via right-click mouse menu) the TAP network adapter **openvpn** (note this is case-sensitive, hence, make sure you specify “openvpn”)

4.3.1.2 Modify QEMU invocation parameters

We now need to modify QEMU invocation parameters to ensure QEMU creates a new virtual Network Interface Card (NIC) and that it connects to the TAP adapter running on our Windows host.

The **QEMU Invocation parameters** need to be modified to as follows:

```
-M malta -kernel C:\MIPS_QEMU_LINUX_v2.6.32\vmlinux -hda C:\MIPS_QEMU_LINUX_v2.6.32\rootfs.ext2 -append "root=/dev/hda rw ip=192.168.10.2" -net nic -net tap,ifname=openvpn
```

Where:
`-net nic` will create a virtual NIC
`-net tap,ifname=openvpn` will ensure that TAP interface is connected to the `openvpn` TAP adapter installed on the host

Note in our example above we have assigned 192.168.10.2 as the IP address of the QEMU simulator and that the above invocation will also work for Stop-mode debugging as outlined in the previous section.

Copy and Paste Warning! *Do not copy and paste the above into PathFinder-XD; it will not work due to PDF issues. The above invocation line can be copied and pasted from the README.TXT supplied with PathFinder-XD.*

4.3.2 Sample Run-mode Linux Debugging Session

This section demonstrates Linux Run-mode Process Debugging using PathFinder-XD and the QEMU simulator 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). As before we have to prepare our kernel for debug, download it to the simulator, execute it and load the kernel symbols into PathFinder. See previous sections **4.1, 4.2.1.1 and 4.2.1.2**. Once these steps are complete we are ready to begin debugging our Module and Process as follows:

4.3.2.1 Debugging the Module and Process

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

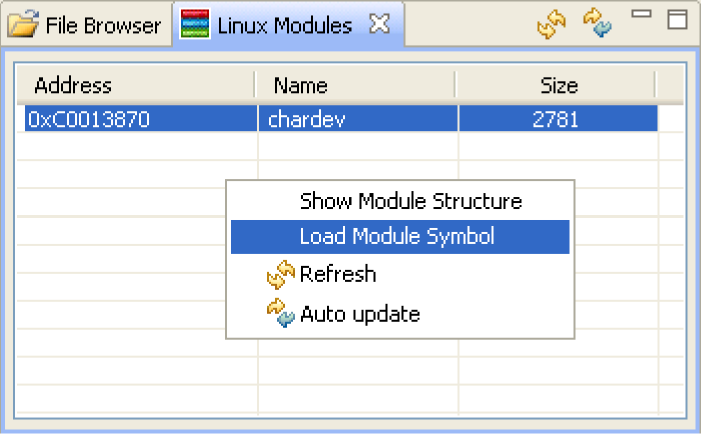
```
QEMU
     device=eth0, addr=192.168.10.2, mask=255.255.255.0, gw=255.255.255.255,
     host=192.168.10.2, domain=, nis-domain=(none),
     bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem) on device 3:0.
Freeing prom memory: 956k freed
Freeing unused kernel memory: 196k freed
Initializing random number generator... EXT2-fs error (device hda): ext2_lookup:
 deleted inode referenced: 1284
Remounting filesystem read-only
EXT2-fs error (device hda): ext2_lookup: deleted inode referenced: 1284
Remounting filesystem read-only
read-only file system detected...done

Welcome to Ashling QEMU Linux
ashqemu login: root
# insmod chardev.ko
chardev: module license 'unspecified' taints kernel.
Disabling lock debugging due to kernel taint

retval=0
Registeration is a success The major device number is 100.
If you want to talk to the device driver,
you'll have to create a device file.
We suggest you use:
mknod char_dev c 100 0
The device file name is important, because
the ioctl program assumes that's the
file you'll use.
# _
```

Figure 42. Loading the Module to be debugged

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



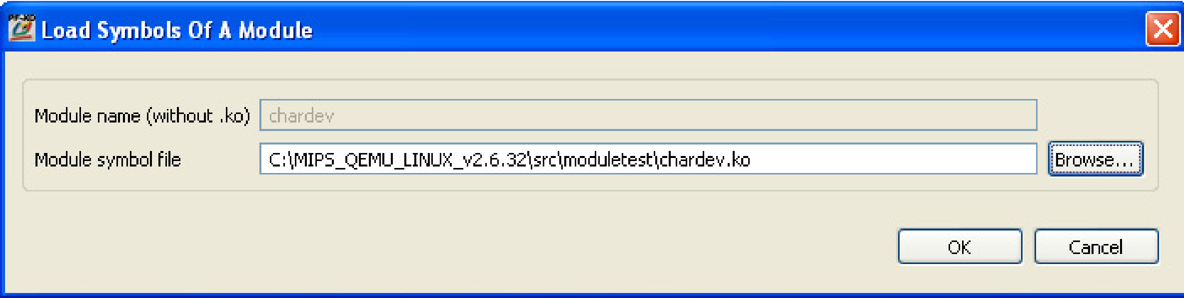


Figure 43. Loading the Module symbols

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

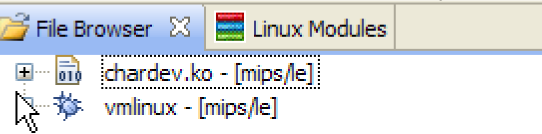


Figure 44. File Browser showing Kernel and Module symbols

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 screen-shot we have opened the Module source-file (`chardev.c`) and set a breakpoint at the function `device_write` which we wish to debug (i.e. this function located in the Module is called from the the Process)

genex.S | chardev.c | ashtestapp.c

```
 * This function is called when somebody tries to
 * write into our device file.
 */
static ssize_t
device_write(struct file *file,
         const char __user * buffer, size_t length, loff_t * offset)
{
    int i;

#ifdef DEBUG
    printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, length);
#endif

    for (i = 0; i < length && i < BUF_LEN; i++)
        get_user(Message[i], buffer + i);
```

Figure 45. Setting a Breakpoint in the Module

4. Next, we resume execution of the Kernel in PathFinder-XD

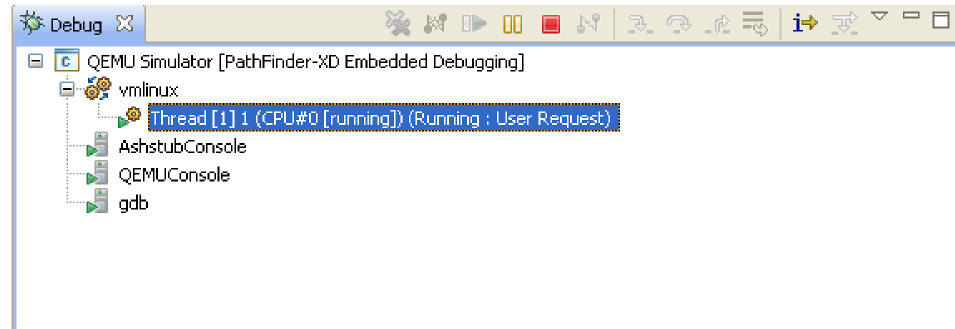


Figure 46. Running the Kernel

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

```
# gdbserver :1234 ./ashtestapp
Process ./ashtestapp created; pid = 821
Listening on port 1234
```

Figure 47. Launching the Process

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

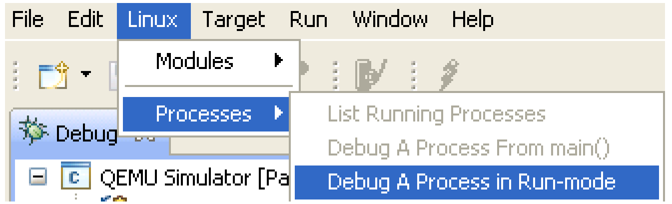


Figure 48. Debugging a Process in Run-mode

We need to specify the Process:

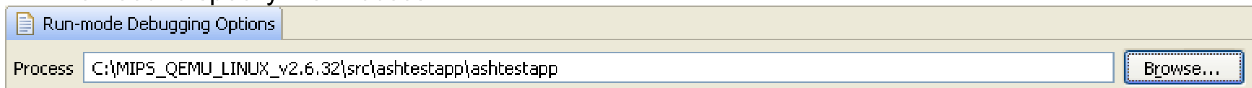


Figure 49. Specifying the Process

The location of the shared libraries:

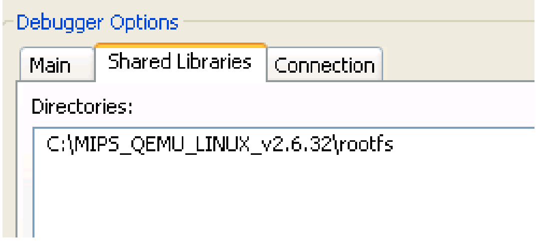


Figure 50. Specifying the Share Library location

And finally, the connection mechanism (TCP in our example) and IP address of the simulator system (which is running gdbserver):

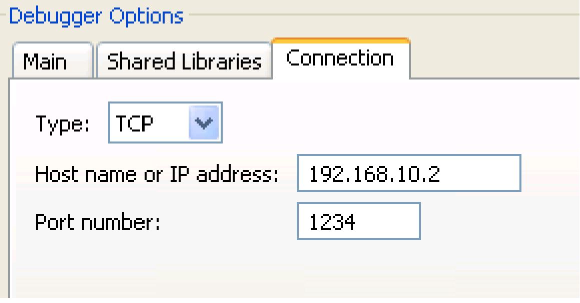


Figure 51. Specifying the Connection mechanism

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

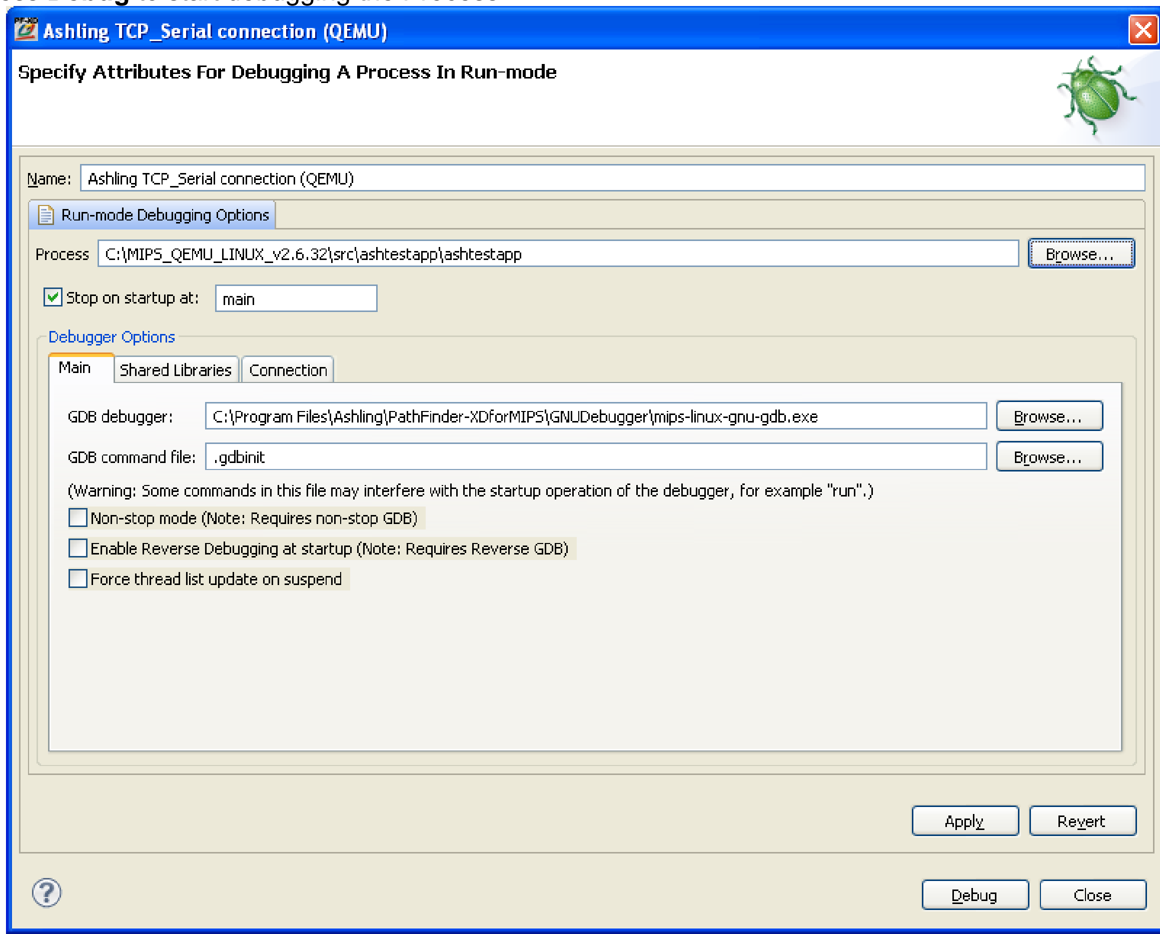


Figure 52. Debugging a Process in Run-mode dialog

6. PathFinder-XD will now update as follows:

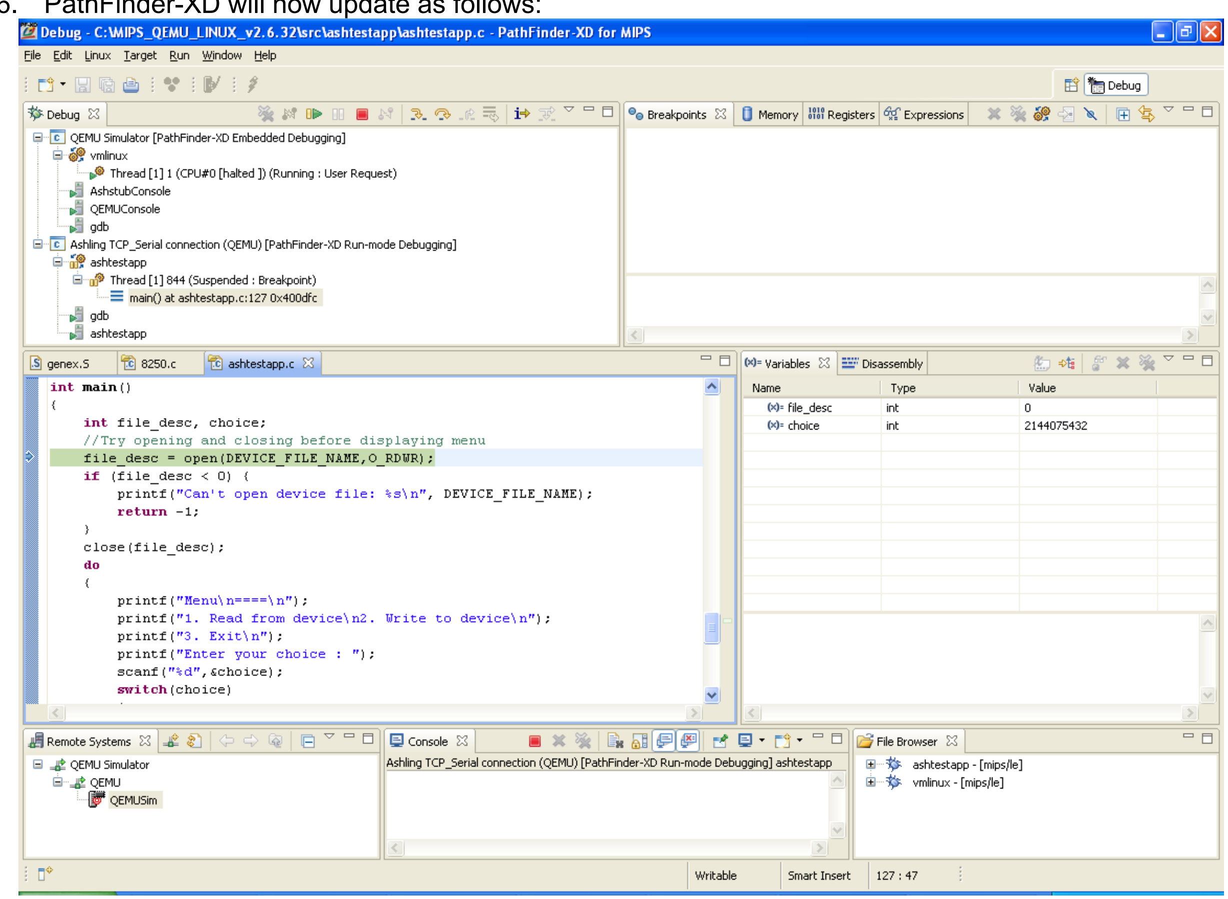


Figure 53. PathFinder-XD in Run-mode

If no source is shown then select the **Edit Source Lookup Path…** button

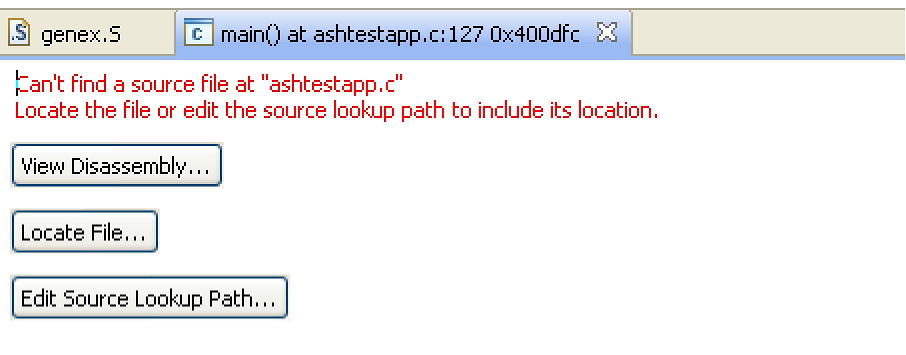


Figure 54. Edit Source Lookup Path

and specify your source-directory as shown below:

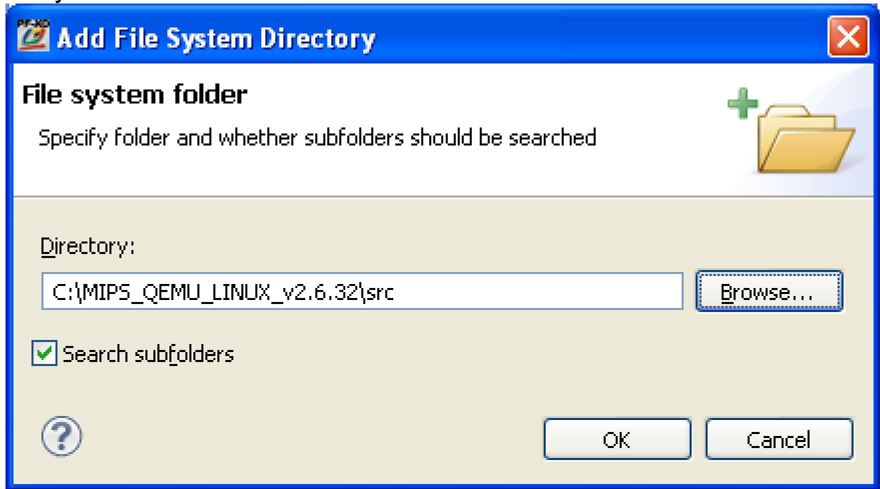


Figure 55. Edit File System Directory

Notice how:

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

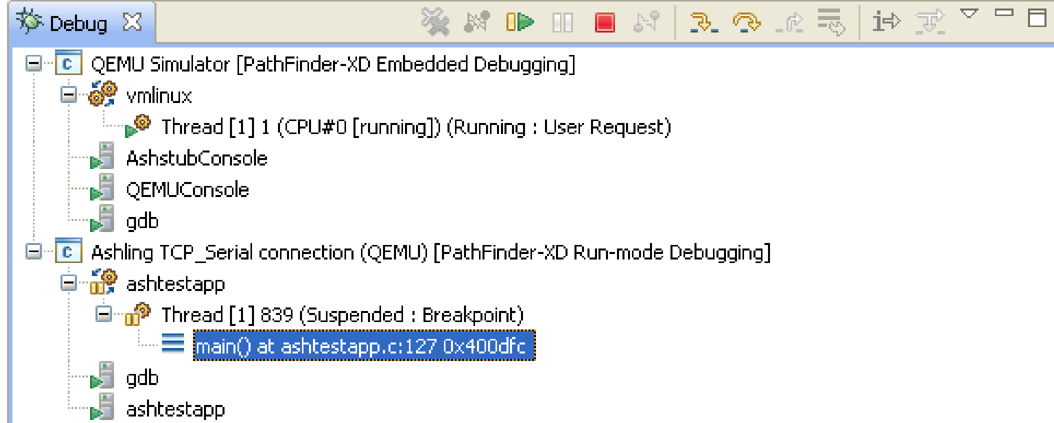


Figure 56. PathFinder-XD Debug Window showing Kernel and Process (Kernel Run-mode) status

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

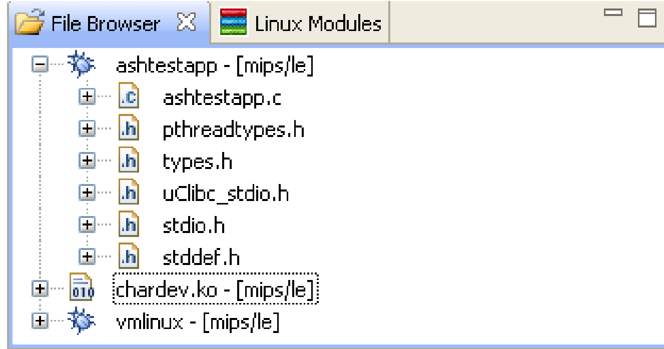


Figure 57. PathFinder-XD File Browser showing Module, Process and Kernel sources

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

0xbfc00000 | genex.S | chardev.c | ashtestapp.c

```
int main()
{
    int file_desc, choice;
    //Try opening and closing before displaying menu
    file_desc = open(DEVICE_FILE_NAME,O_RDWR);
    if (file_desc < 0) {
        printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
        return -1;
    }
    close(file_desc);
    do
    {
        printf("Menu\n====\n");
        printf("1. Read from device\n2. Write to device\n");
        printf("3. Exit\n");
```

Figure 58. Process Source

7. We can now debug our Process as normal with the Kernel running in the background. After running the process, choose the options as seen in the figure below:

QEMU [Stopped]

```
the ioctl program assumes that's the
file you'll use.
# rmmod chardev.ko
# insmod chardev.ko

retval=0
Registeration is a success The major device number is 100.
If you want to talk to the device driver,
you'll have to create a device file.
We suggest you use:
mknod char_dev c 100 0
The device file name is important, because
the ioctl program assumes that's the
file you'll use.
# gdbserver :1234 ./ashtestapp
Process ./ashtestapp created; pid = 826
Listening on port 1234
Remote debugging from host 192.168.10.1
device_open(8786d300)
device_release(87425624,8786d300)
Menu
====
1. Read from device
2. Write to device
3. Exit
Enter your choice : 2
Enter data to write :
Ashling
device_open(87a5b800)
_
```

Figure 59. Running the process

This will invoke the functions with a breakpoint set and PathFinder-XD's Debug window will update as follows:

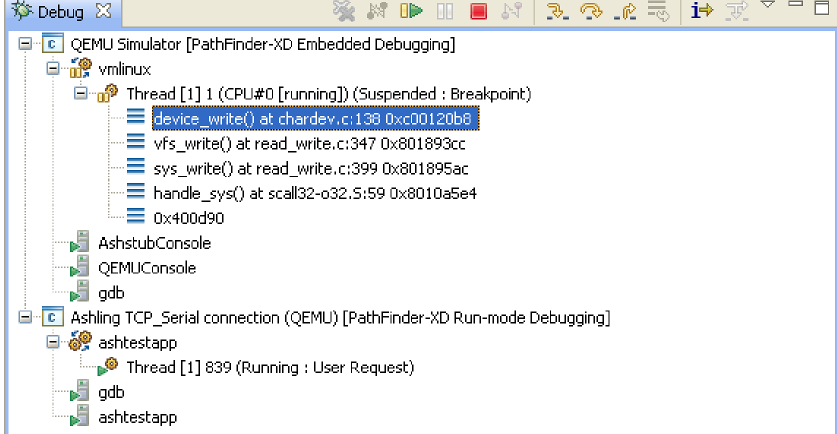


Figure 60. PathFinder-XD Debug Window showing the Kernel halted

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 only. 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, complete the following steps:

1. Launch gdbserver from the simulator (i.e. in the Linux Console) specifying the application we wish to debug (`threadtestapp`).

```
[root@ashqemu ~]# gdbserver :1234 ./threadtestapp
Process ./threadtestapp created; pid = 850
Listening on port 1234
```

Figure 61. Launching multi-threaded Process

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

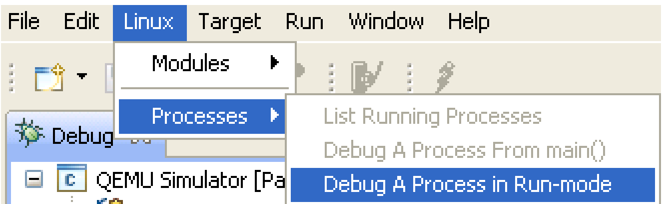
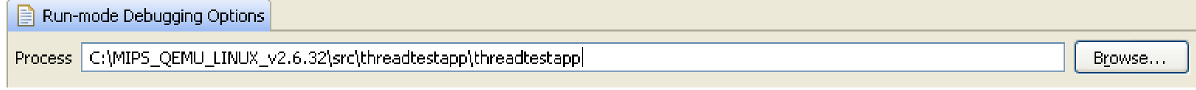


Figure 62. Debugging a Process in Run-mode

3. Select the application (Process) to debug



4. Choose the location of shared libraries:

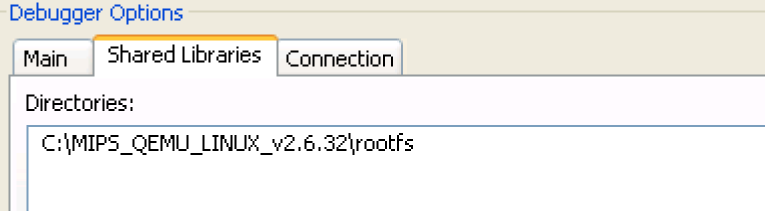


Figure 63. Specifying the Share Library location

5. And finally, the connection mechanism (TCP in our example) and IP address of the simulator (which is running gdbserver):

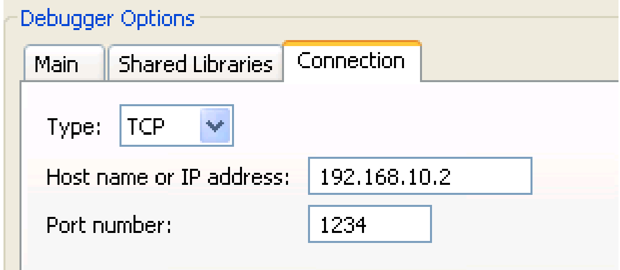


Figure 64. Specifying the Connection mechanism

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

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

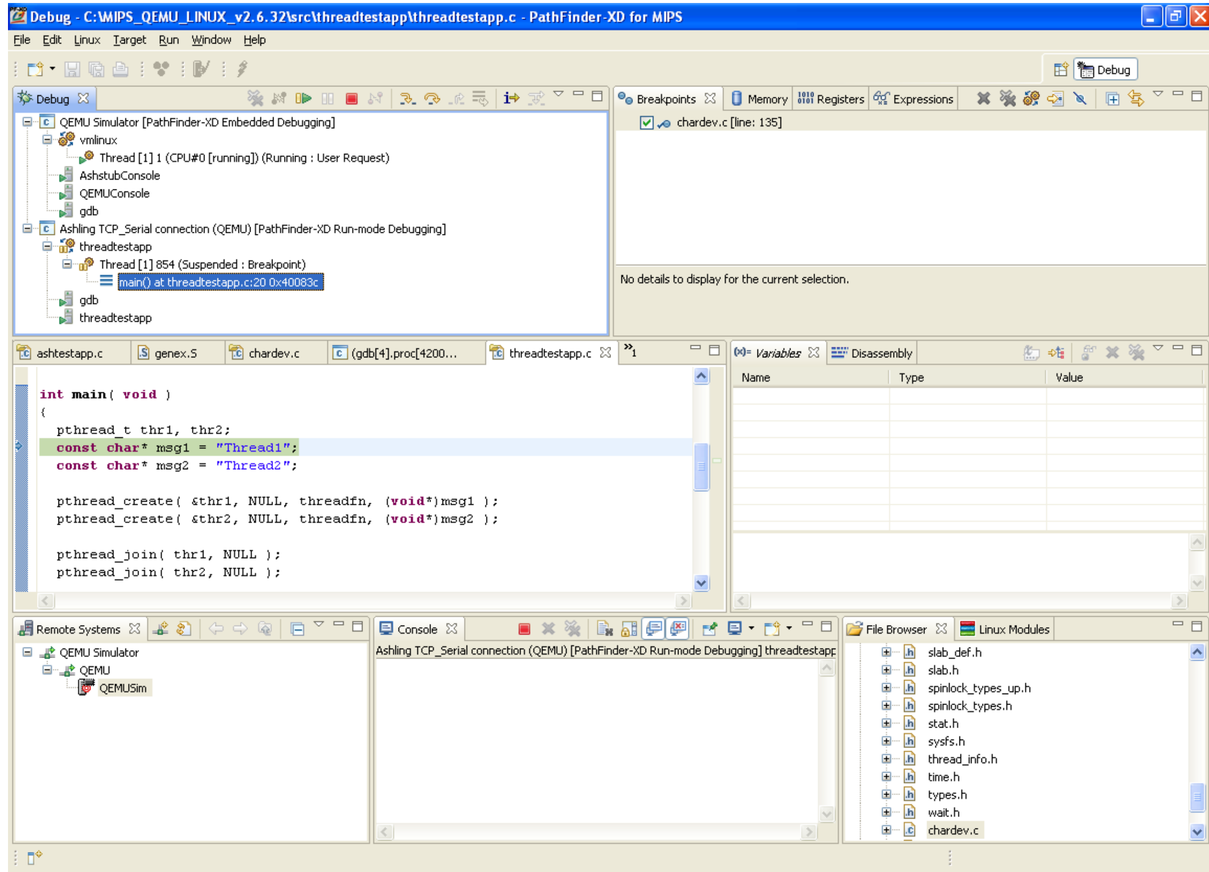


Figure 65. Program runs to main()

8. Set a breakpoint in **`threadfn()`** and run to that point, the Debug view updates as follows:

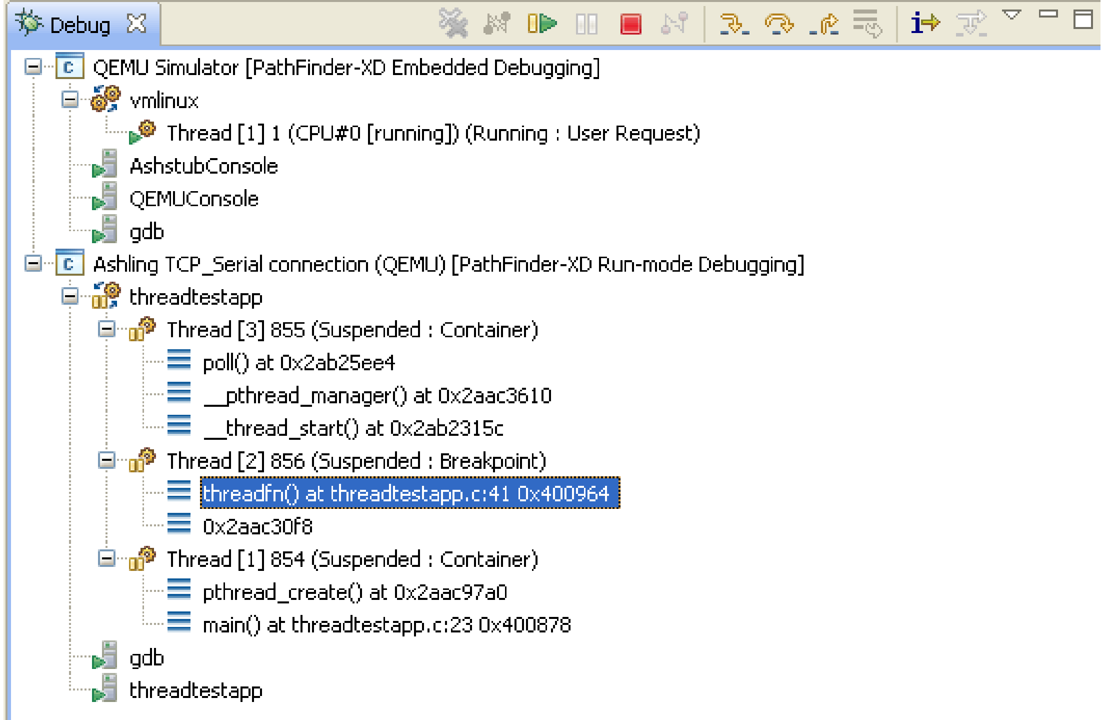


Figure 66. Multi-threaded Debug View

Notice how the threads are listed and the call stack for each thread is shown in the Debug view. While clicking on each thread context, all PathFinder-XD windows will update accordingly (i.e. thread specific).

To set a thread specific breakpoint:

- Set a breakpoint in a location by double clicking on the ruler.
- Right click the on the breakpoint in the ruler and choose **Breakpoint Properties**

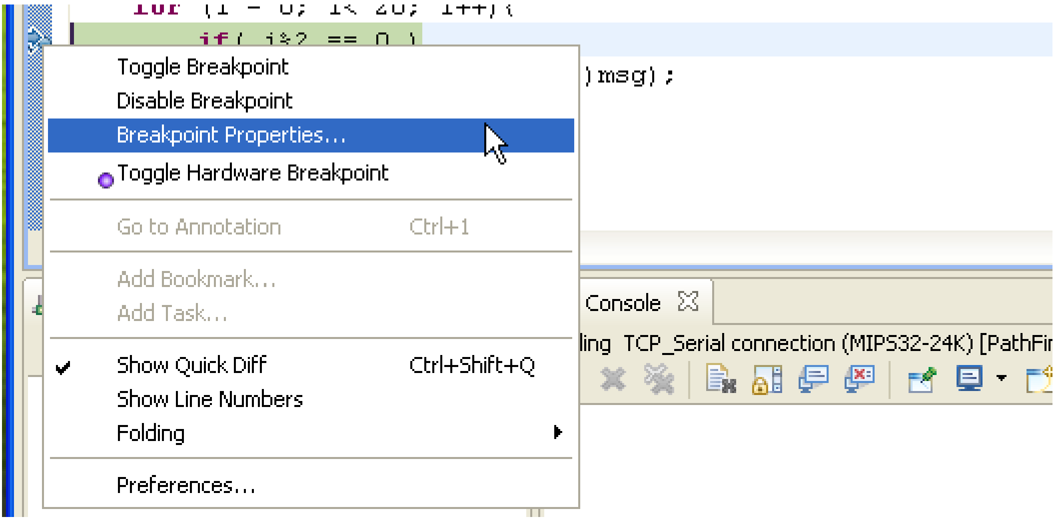


Figure 67. Selecting Breakpoint Properties

- And in the filtering section, check the threads you wish to associate with the breakpoint.

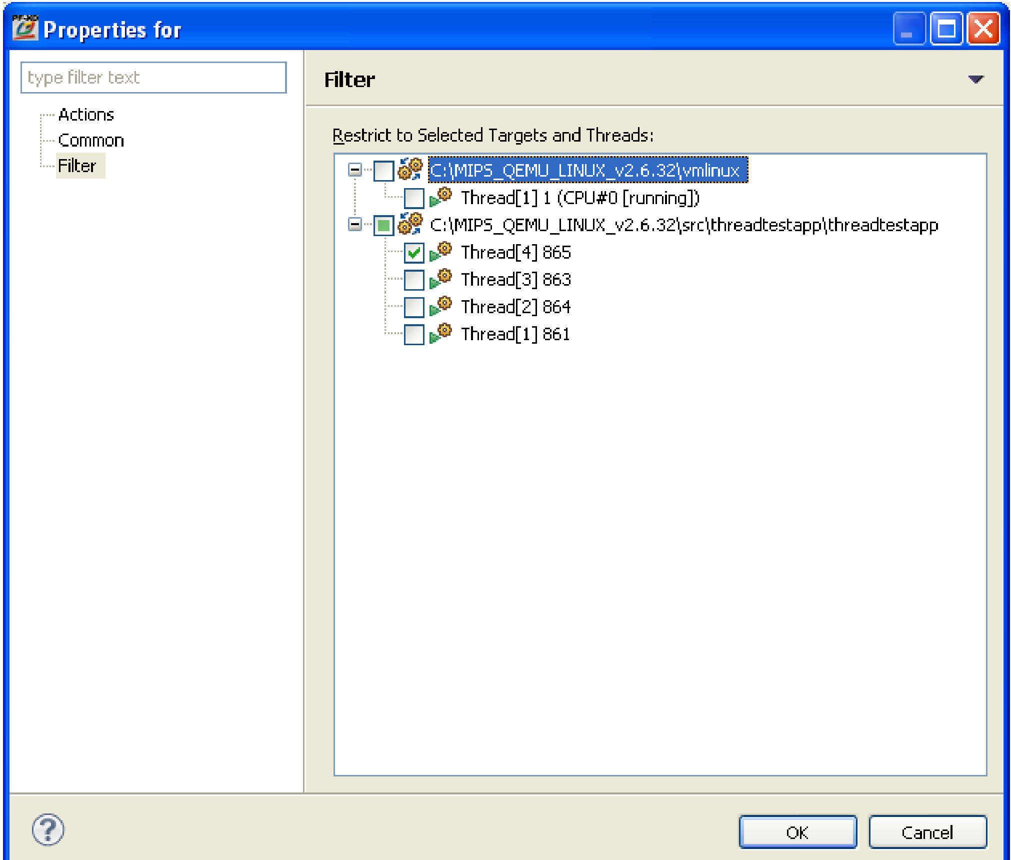


Figure 68. Breakpoint Properties Dialog

- Click OK to set the breakpoint

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 for each process. Use separate port number for each GDB server and use the "&" symbol at the end of the command. Each gdbserver process will start in the background as shown in screenshot.

```
[root@ashqemu ~]# gdbserver :1234 ./threadtestapp  &
[1] 859
[root@ashqemu ~]# Process ./threadtestapp created; pid = 862
Listening on port 1234

[root@ashqemu ~]# gdbserver :2345 ./ashtestapp  &
[2] 864
Process ./ashtestapp created; pid = 867
[root@ashqemu ~]# Listening on port 2345
```

Figure 69. Launching Multiple Processes

You can now connect to each process via the **Debug A Process in Run-mode** menu. The Debug View will show each process as follows:

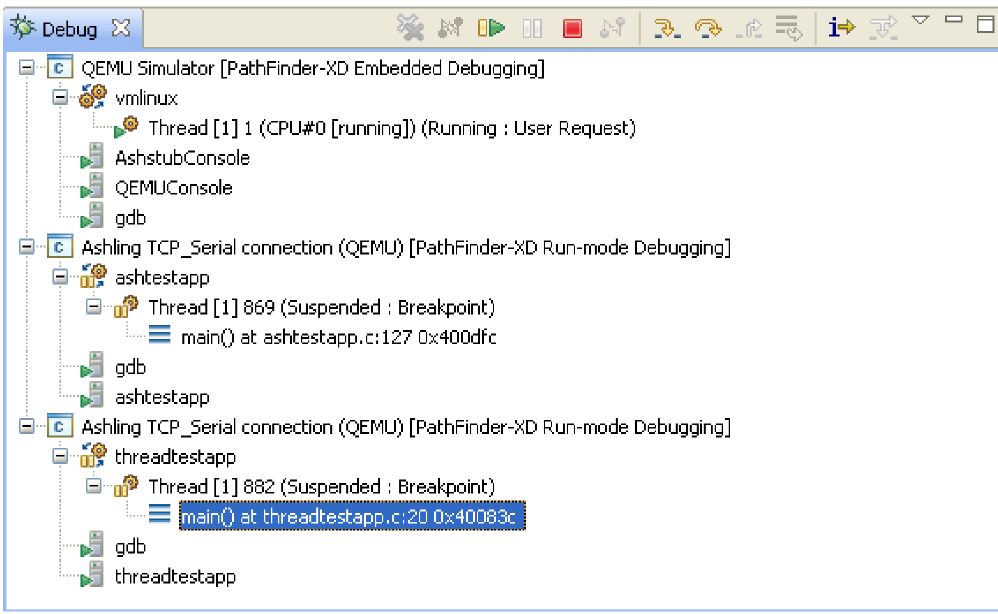


Figure 70. Debugging Multiple Processes

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: APB211-PF-XD_MIPS_SIM, Hugh O'Keeffe and Suresh PC, Ashling Microsystems