Using JTAG Hardware Debuggers with the QNX Neutrino RTOS

by Sheridan Ethier

1. Introduction

This document describes how to use JTAG hardware debuggers with the QNX Neutrino real-time operating system for embedded software development. With QNX Neutrinos microkernel architecture, you rarely have to use a hardware debugger. On those occasions when you need to use one, you'll use it in a different manner from on other operating systems. This article describes how JTAG hardware debuggers fit into the QNX development cycle, what capabilities they provide, and what's required in order to use them with the QNX Neutrino RTOS. Finally, an example is provided that demonstrates debugging a QNX target using a commercially available JTAG hardware debugger.

2. Software vs. Hardware Debuggers (pdebug vs. JTAG)

This section examines fundamental differences between QNX Neutrino and conventional RTOSs, and how these differences affect debugging embedded software.

The QNX Neutrino RTOS architecture differs significantly from more traditional monolithic operating systems, and as a result, it enhances the software development and debugging processes.

Traditional embedded software development often requires hardware debuggers connected through a JTAG interface. This is necessary for development of drivers, and possibly user applications, because they're linked into the same memory space as the kernel. If a driver or application crashes, the kernel and system may crash as a result. This makes using software debuggers difficult, because they depend on a running system.

Debugging target systems with the QNX Neutrino RTOS is different because its architecture is significantly different from other embeddable RTOSs. All QNX applications (including drivers) run in their own memory-protected virtual address space. This has the advantage that the software is more reliable and fault tolerant. However, conventional hardware debuggers rely on decoding physical memory addresses, making them incompatible with debugging user applications based in a virtual memory environment. Furthermore, QNX Neutrino lets you develop multi-threaded applications, which hardware debuggers generally don't support.

QNX provides a software debugging agent called **pdebug** that makes it easier for you to debug system drivers and user applications. The **pdebug** agent runs on the target system and communicates with the host debugger over a serial or Ethernet connection. It can debug virtual-memory-based and multi-threaded applications. Even more, because all applications run in their own memory-protected address space, using a software debugger is much more reliable than on traditional embedded RTOSs. In the event that a driver or user application crashes, the system is protected and can be recovered by simply restarting the process. In addition, you can stop and restart drivers and applications as required on a running system, making the debugger easy to use and highly reliable.

Using **pdebug** for debugging applications and drivers is extremely reliable because crashing applications don't crash the kernel. However, the major constraint of using **pdebug** is that the kernel must already be running on the target.

A Board Support Package (BSP) is the software responsible for initializing the system in preparation for the kernel. Each target board requires its own BSP, which includes the Initial Program Loader (IPL) and

Startup program. The role of the IPL is to find where the QNX Neutrino image is located and then set up an environment so that the Startup program (present in the image) can run. The role of the Startup program is to configure the processor and hardware, detect system resources, and start the OS.

The IPL and Startup must run properly, so that the kernel and **pdebug** can run, and drivers and user applications can be developed. In the case of BSP development, a software debug agent such as **pdebug** isn't available. However, the IPL and Startup program run with the CPU in physical mode, allowing them to be debugged with conventional hardware debuggers. This is the primary function of the JTAG debugger throughout the QNX software development phase. You use the hardware debugger to debug the QNX BSP (IPL and Startup), and **pdebug** to debug drivers and applications once the kernel is running. You can also use a hardware debugger to examine registers and view memory while the kernel and applications are running, if you know the physical addresses.

Some hardware debuggers have built-in QNX Neutrino RTOS Awareness, which lets you use a JTAG to debug applications. These debuggers can interpret kernel information as well as perform the necessary translation between virtual and physical memory addresses to view application data. Currently, Lauterbach provides QNX Neutrino RTOS integration support for their Trace32 hardware debugger, and Applied Microsystems Corporation (AMC) is working on integrating PowerTAP and WireTAP products with QNX Neutrino.

3. Producing Debug Symbol Information for IPL and Startup

You can use hardware debuggers to debug QNX IPL and Startup programs without any extra information. However, in this case, you're limited to assembly-level debugging, and assembler symbols such as subroutine names aren't visible. To perform full source-level debugging, you need to provide the hardware debugger with the symbol information and C source code.

This section describes the steps necessary to generate the symbol and debug information required by a hardware debugger for source-level debugging. The steps described are based on the PPC (PowerPC) Board Support Package available for QNX Neutrino 6.2 for both IPL and Startup of the Motorola Sandpoint hardware reference platform. The steps below are described for a QNX 6.2 self-hosted environment, but the commands are similar under other development platforms. These examples assume that you're logged in on the development host with root privileges.

To generate symbol information for the IPL, you must recompile both the IPL library and the Sandpoint IPL with debug information.

The general procedure is as follows:

1. Modify the IPL source.
2. Build the IPL library and Sandpoint IPL.
3. Burn the IPL into the flash memory of the Sandpoint board using a flash burner or JTAG.
4. Modify the *sandpoint.lnk* file to output ELF format.
5. Recompile the IPL library and Sandpoint IPL source with debug options.
6. Load the Sandpoint IPL ELF file containing debug information into the hardware debugger.

Note: Be sure to synchronize the source code, the IPL burned into flash, and the IPL debug symbols.

To build the IPL library with debug information:

```
# cd /usr/src/bsp-6.2.0/libs/src/hardware/ipl/lib/ppc/a.be
# make clean
# make CCOPTS=-g
```

```
# cp libipl.a /usr/src/bsp-6.2.0/ppc/sandpoint/scratch/ppcbe/lib
# make install
```

The above steps recompile the PowerPC IPL library (*libipl.a*) with DWARF debug information and copy this library to the Sandpoint scratch directory. The Sandpoint BSP is configured to look for this library first in its scratch directory. The `make` **install** is optional, and copies *libipl.a* to */ppcbe/usr/lib*.

Note: If you're using the AMC hardware debugger, use the STABS format instead of DWARF, by specifying **-gstabs+** instead of the **-g** option.

The Sandpoint BSP has been set up to work with SREC format files. However, to generate debug and symbol information to be loaded into the hardware debugger, you must generate ELF-format files.

Modify the `sandpoint.lnk` file to output ELF format:

```
# cd /usr/src/bsp-6.2.0/ppc/sandpoint/src/hardware/ipl/boards/sandpoint
```

Edit the file *sandpoint.lnk*, changing the first lines from:

```
TARGET(elf32-powerpc)
OUTPUT_FORMAT(srec)
ENTRY(entry_vec)
```

to:

```
TARGET(elf32-powerpc)
OUTPUT_FORMAT(elf32-powerpc)
ENTRY(entry_vec)
```

You can now rebuild the Sandpoint IPL to produce symbol and debug information in ELF format.

To build the Sandpoint IPL with debug information:

```
# cd /usr/src/bsp-
6.2.0/ppc/sandpoint/src/hardware/ipl/boards/sandpoint/ppc/be
# make clean
# make CCOPTS=-g
```

The *ipl-sandpoint* file is now in ELF format with debug symbols from both the IPL library and Sandpoint IPL.

Note: To rebuild the BSP, you need to change the *sandpoint.lnk* file back to outputting SREC format. It's also important to keep the IPL that's burned into the Sandpoint flash memory in synch with the generated debug information; if you modify the IPL source, you need to rebuild the BSP, burn the new IPL into flash, and rebuild the IPL symbol and debug information.

You can use the **objdump** utility to view the ELF information.

To view the symbol information contained in the ipl-sandpoint file:

```
# objdump -t ipl-sandpoint | less
```

You can now import the *ipl-sandpoint* file into a hardware debugger to provide the symbol information required for debugging. In addition, the hardware debugger needs the source code listings found in the following directories:

- */usr/src/bsp-6.2.0/ppc/sandpoint/src/hardware/ipl/boards/sandpoint*
- */usr/src/bsp-6.2.0/libs/src/hardware/ipl/lib*
- */usr/src/bsp-6.2.0/libs/src/hardware/ipl/lib/ppc*

To generate symbol information for Startup, you must recompile both the Startup library and Sandpoint Startup with debug information.

The general procedure is as follows:

Modify the Startup source:

1. Build the Startup library and Sandpoint Startup with debug information.
2. Rebuild the image and symbol file.
3. Load the symbol file into the hardware debugger program.
4. Transfer the image to the Sandpoint target (burn into flash, transfer over a serial connection).

To build the Startup library with debug information:

```
# cd /usr/src/bsp-6.2.0/libs/src/hardware/startup/lib/ppc/a.be
# make clean
# make CCOPTS=-g
# cp libstartup.a /usr/src/bsp-6.2.0/ppc/sandpoint/scratch/ppcbe/lib
# make install
```

The above steps recompile the PowerPC Startup library (*libstartup.a*) with DWARF debug information and copy this library to the Sandpoint scratch directory. The Sandpoint BSP is configured to look for this library first in its scratch directory. The **make install** is optional, and copies *libstartup.a* to */ppcbe/usr/lib*.

Note: Once again, if you're using the AMC hardware debugger, use the STABS format instead of DWARF, by specifying **-gstabs+** instead of the **-g** option.

To build the Sandpoint Startup with debugging information:

```
# cd /usr/src/bsp-
6.2.0/ppc/sandpoint/src/hardware/startup/boards/sandpoint/ppc/be
# make clean
# make CCOPTS=-g
# make install
```

The above steps generate the file *startup-sandpoint* with symbol and debug information. Again, you can use the **gstabs+** debug option instead of **-g**. The **make install** is necessary, and copies *startup-sandpoint* into the Sandpoint scratch directory, */usr/src/bsp6.2.0/ppc/sandpoint/scratch/ppcbe/boot/sys*.

Note: You can't load the *startup-sandpoint* ELF file into the hardware debugger to obtain the debug symbols, because the **mkifs** utility adds an offset to the addresses defined in the symbols according to the offset specified in the build file.

Modify the build file to include the `+keeplinked` attribute for Startup:

```
# cd /usr/src/bsp-6.2.0/ppc/sandpoint/images
```

Modify the startup line of your build file to look like:

```
[image=0x10000]
[virtual=ppcbe,binary +compress] .bootstrap = {
[+keeplinked] startup-sandpoint -vvv -D8250
PATH=/proc/boot procnto-600 -vv
}
```

The **+keeplinked** option makes **mkifs** generate a symbol file that represents the debug information positioned within the image filesystem by the specified offset.

Rebuild the image to generate symbol file:

```
# cd /usr/src/bsp-6.2.0/ppc/sandpoint/images
# make clean
# make all (if you're using one of the provided .build files)
```

or:

```
# mkifs v r ../scratch myfile.build image
```

These commands create the symbol file, *startup-sandpoint.sym*. You can use the **objdump** utility to view the ELF information.

To view the symbol information contained in the `startup-sandpoint.sym` file:

```
# objdump -t startup-sandpoint.sym | less
```

You can now import the *startup-sandpoint.sym* file into a hardware debugger to provide the symbol information required for debugging startup. In addition, the hardware debugger needs the source code listings found in the following directories:

- */usr/src/bsp-6.2.0/libs/src/hardware/startup/lib*
- */usr/src/bsp-6.2.0/libs/src/hardware/startup/lib/public/ppc*
- */usr/src/bsp-6.2.0/libs/src/hardware/startup/lib/public/sys*
- */usr/src/bsp-6.2.0/libs/src/hardware/startup/lib/ppc*
- */usr/src/bsp-6.2.0/ppc/sandpoint/src/hardware/startup/boards/sandpoint*

If you have access to the kernel source files, you can use a hardware debugger to debug the QNX Neutrino kernel. Follow a similar procedure as described above, modifying the build file to use the **keeplinked** attribute on **procnto**:

Modify the build file to include the `+keeplinked` attribute for `procnto`:

```
[image=0x10000]
[virtual=ppcbe,binary +compress] .bootstrap = {
[+keeplinked] startup-sandpoint -vvv -D8250
[+keeplinked] PATH=/proc/boot procnto-600 -vv
}
```

4. Example using AMC PowerTAP JTAG and MWX-ICE

This section describes how to use a JTAG hardware debugger to debug QNX IPL and Startup programs for the Motorola Sandpoint reference platform. The example is based on the AMC MWX-ICE debugger connected to a PowerTAP JTAG.

We assume that you've installed the AMC PowerTAP and MWX-ICE debugger on the host and configured them properly for the Sandpoint board, and that the target is connected and powered. This example also assumes that the IPL already has been burned into flash memory.

Launch the MWX-ICE debugger, shown below:

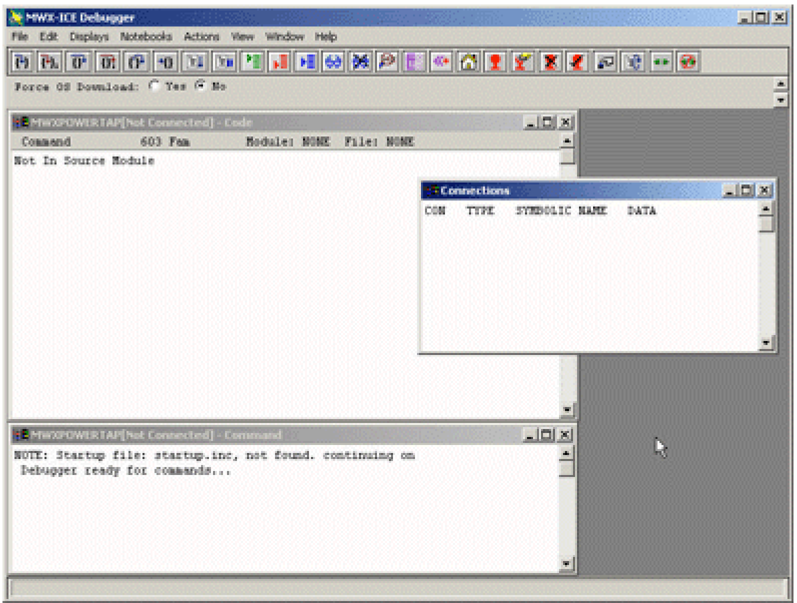


Figure 1 - MWX-ICE Debugger main window.

Across the top of the main window is a row of buttons that you'll use in the steps described below.

With the Connections window in focus, choose **Actions→Define Ethernet Connection** from the main menu. Enter a name for the connection, and the IP address that the PowerTAP is configured to on the network. After selecting OK, choose **Actions→Connect to connect to the PowerTAP** and target. The output is displayed in the Command window.

You need to configure the MWX-ICE debugger for the particular target being used; consult AMC for details. For this example, we've entered the following configuration commands in the Command window:

```
size read 4
mem_rd_del 0x500
pcimap mapb
bptype onchip
```

You can also use the Emulator Config window (**Displays→Emulator Config**) to set these values.

Note: This isn't an exhaustive list of configurations required for this target. Also, AMC recommends the memory read delay be between `0x500` and `0xF00`.

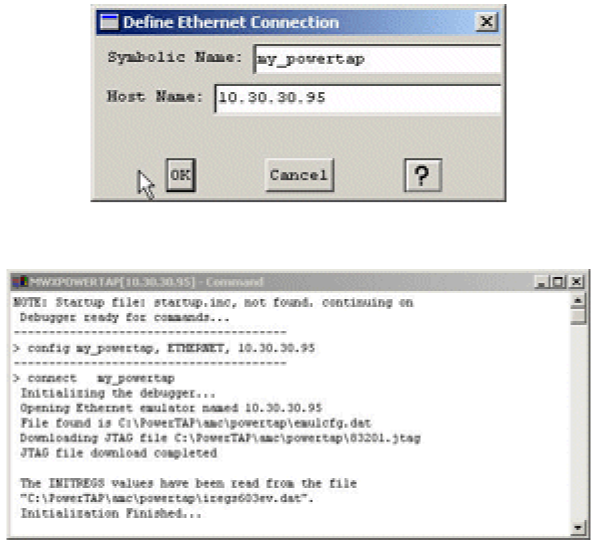


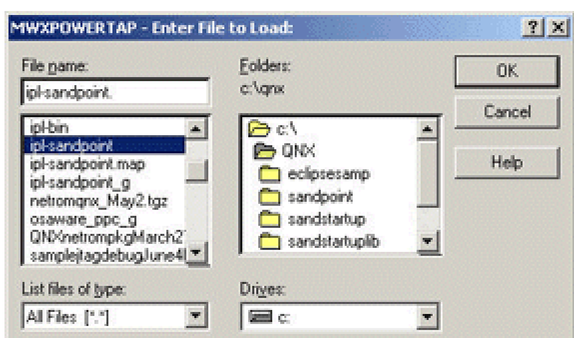
Figure 2 The Define Ethernet Connection and Command windows.

At this point, the MWX-ICE debugger should be configured and ready to begin debugging the Sandpoint reference platform via the PowerTAP.

This section describes how to debug the QNX IPL using the symbol information generated using the Sandpoint BSP.

First, you must import the IPL symbols into the debugger.

1. From the main menu, choose **File→Load** to display the Load window.
2. Under List Files of type, select All Files [*.*].
3. Select the *ipl-sandpoint* ELF file generated in the previous section, and click **OK**.
4. After loading the symbol information, the debugger likely displays the Append new Directory popup to request the location of the source files.
5. Select the directory where the IPL library source is located.
6. Repeat this for each of the IPL-related source directories listed at the end of the ***Generating IPL Debug*** Symbols section, by selecting **File→Append Source Path**.



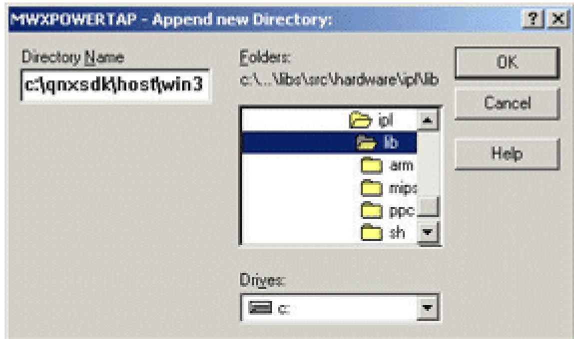


Figure 3 The Load Symbols and Append Directory windows.


The debugger is now ready to debug the IPL. To reset the target, click the **Reset** button.

To view the **main**() function from the Sandpoint IPL source code, bring the Code window in focus, set the **Mode** to **Source**, type `main` in the text box, and click the **Display** button.

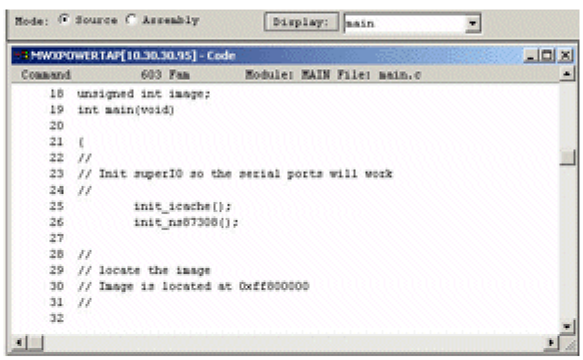
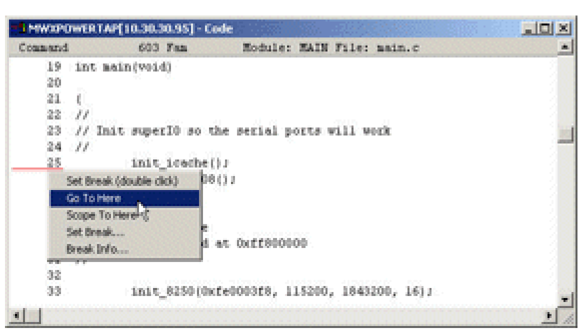


Figure 4 Viewing the source for the *main*() function.

The Code window displays the C source code for the **main**() function. In the Code window, right-click on the line number in **main**() that you wish to break at, and select **Go To Here** from the popup menu. The debugger runs the IPL program, and breaks on this line.



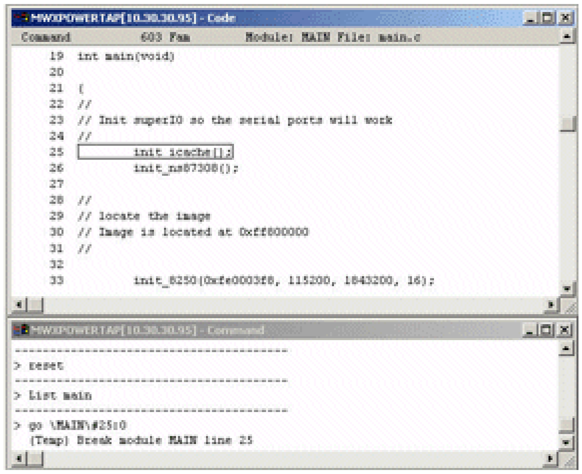


Figure 5 Running the IPL and breaking on the first line in *main*().


You can use the **Stepin Src** button, to step into the **init_icache**() function, and the **Stepout** button, to step back out of it.

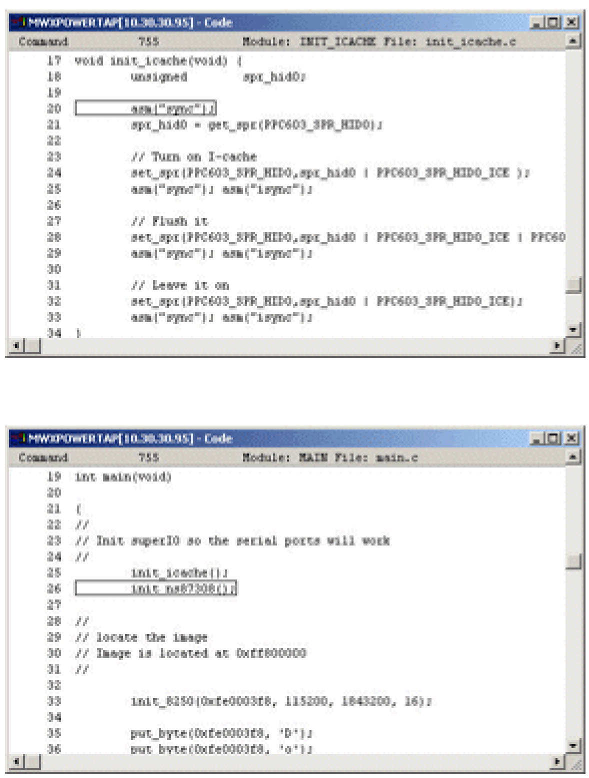


Figure 6 Stepping in and out of the *init_icache*() function.

You can also debug the source code in the IPL library. For example, in the Code window, type `init_8250` in the text box and push the Display button to view this function.

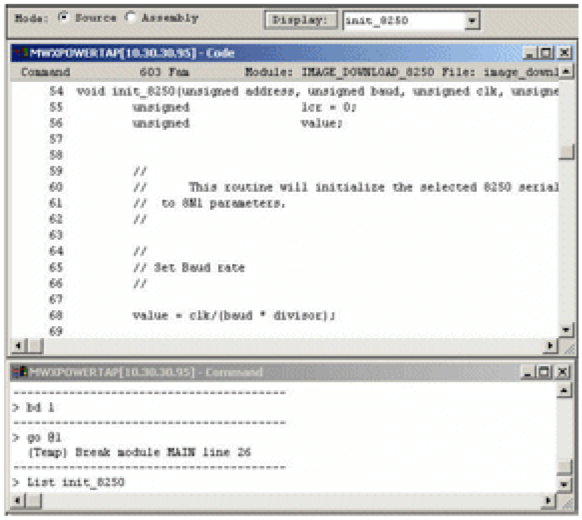


Figure 7 Viewing IPL library source code.

To set a breakpoint in the **`init_8250`**() function, right-click over a line number and select **Set Break** from the popup menu. The breakpoint is now indicated. You can also open the breakpoint display window (**Displays→Breakpoints**) to see the breakpoint.

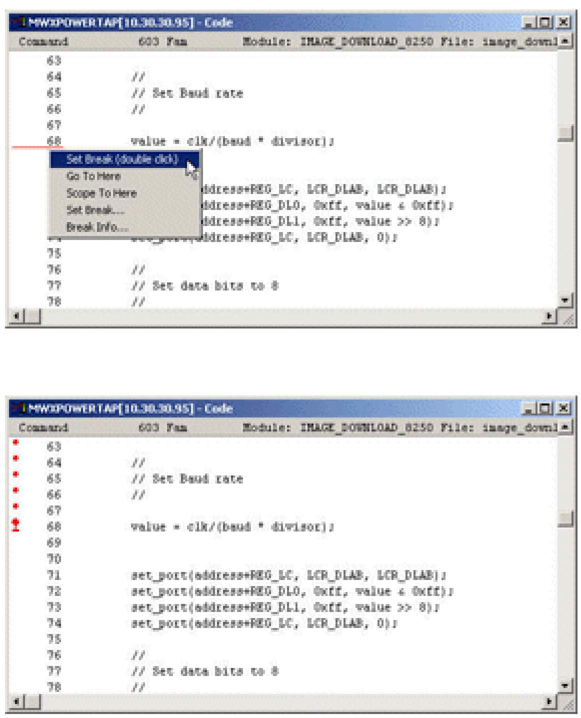


Figure 8 Setting and viewing a breakpoint in the *init_8250*() function.

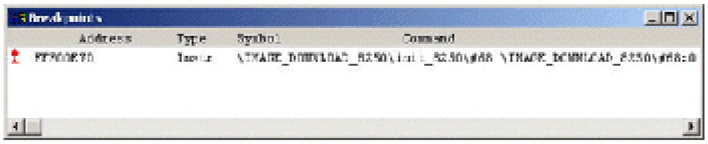


Figure 9 The Breakpoints window.

 You can click the **Go** button, to let the program run until it reaches the breakpoint. Once the debugger has stopped on the breakpoint, you can print data values by right-clicking on top of a variable and selecting **Print**.

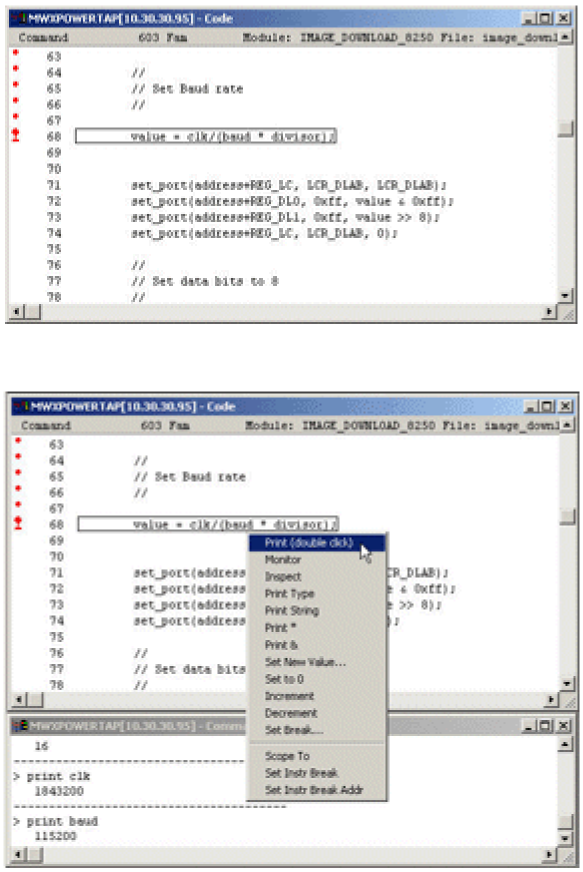


Figure 10 Hitting the breakpoint, and printing variable data.

Processors have a limited number of hardware breakpoints. After stopping at the hardware breakpoint, you should disable it because some stepping commands (such as stepping over or stepping out of source) require the use of breakpoints. To disable, place the cursor over the breakpoint in either the Code or Breakpoint window, right-click, and select **Disable Break** from the pop-up menu.

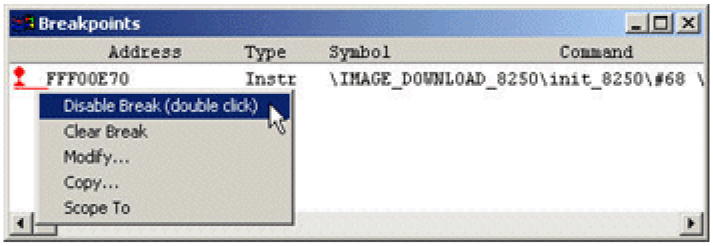


Figure 11 Disabling breakpoints.

The MWX-ICE debugger provides many useful debugging tool displays, including Registers, Memory, Stack, and Trace. For example, to view the IP and other common CPU registers, select **Displays→Registers→General Purpose**. Through this interface, the debugger lets you modify the contents of the registers - you can even modify the CPUs instruction pointer.

To view target memory, select **Displays→Memory**. The Memory window lets you enter an address to view system memory. This interface also lets you modify the contents of memory.

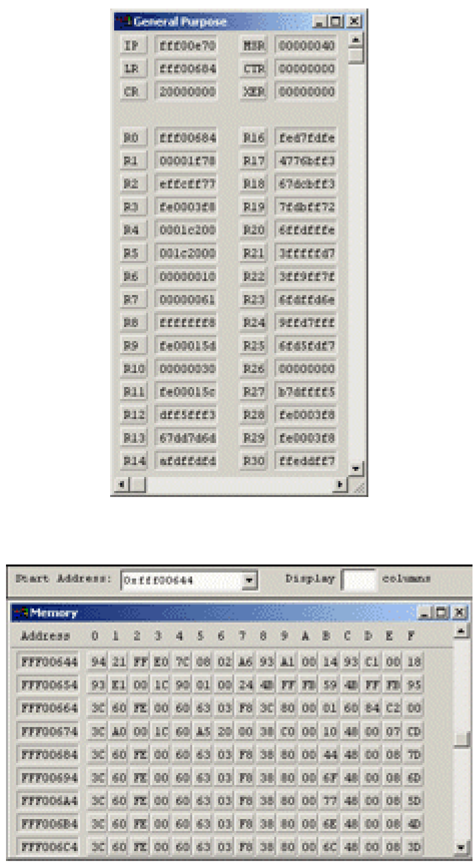


Figure 12 The General Purpose Register and Memory Display windows.

This example shows assembly-level debugging of the IPL from the reset vector.

In the Code window, set the mode to Assembly. Click the **Reset** button, to reset the target and cause the debugger to jump to the first assembly line instruction executed on reset labeled as **`entry_vec`** at address `0xFFF00100`.

Note: You might need to select **View→Scope to PC** to see the assembly instructions. The assembly source may not be viewable if you didn't set the memory read delay command, **`mem_rd_del`** `0x500`, as described earlier.

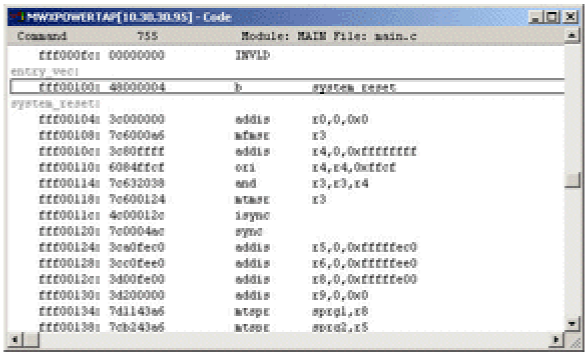
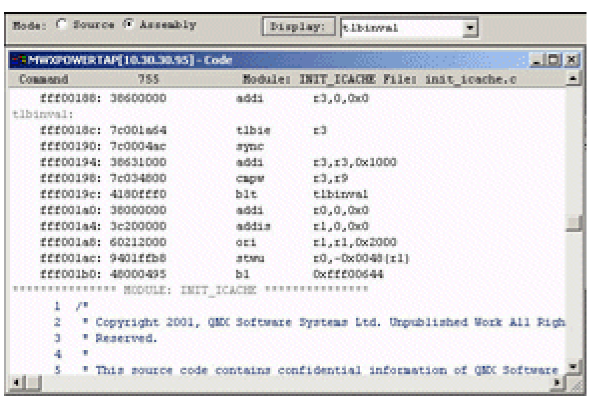


Figure 13 Debugging Assembly instructions, starting from target reset.

 You can step through the assembly instructions by selecting the **Stepin Asm** button.

For example, you can step through the initial assembly code into the IPLs **main**() function. In the Code windows text box, type `tlbinval` and click Display. The **tlbinval**() assembler routine contains the branch instruction to the IPL **main**() function. To run the debugger up to the last instruction, right-click at the very left of the last line of **tlbinval**() and select **Go To Here** from the popup menu.

Note: Ensure that you've disabled the breakpoint set in the previous section.



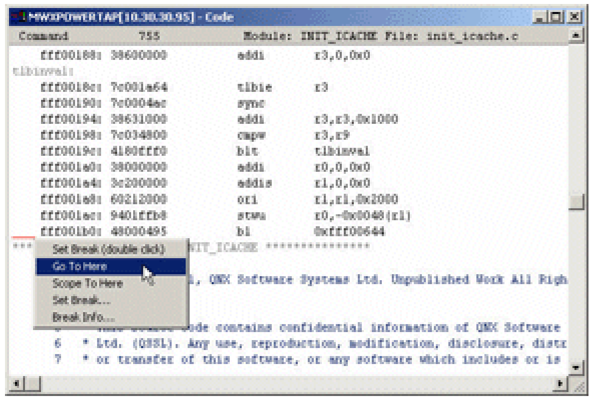


Figure 14 Displaying the tlbinval() subroutine and running to the branch instruction.

If you click the **Stepin Asm** button, the debugger jumps to the **main**() function assembly code. If you've set the debuggers **LINES=ON** option, the C source code is intermixed with assembly code in the Assembly window.

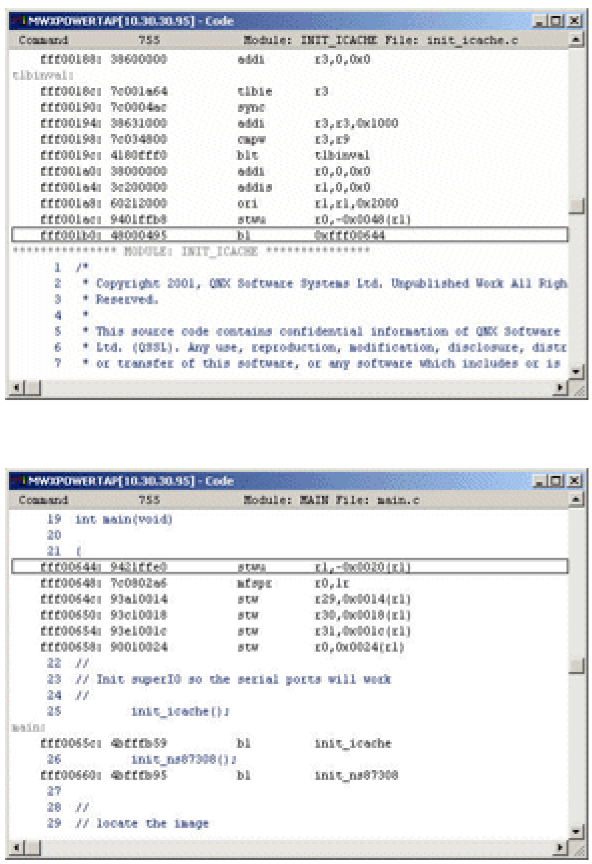


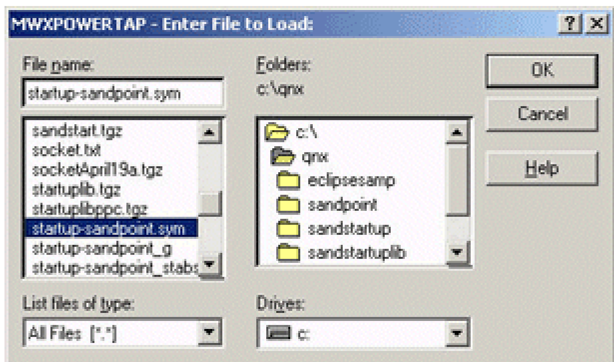
Figure 15 Stepping into the main() function.

If you set the mode to Source in the Code window, the debugger displays the C source code for the **main**() function and lets you step through the source as previously described.

This section describes how to begin debugging the QNX Startup program using the symbol information generated using the Sandpoint BSP. We assume that you've already connected the debugger to the PowerTAP, as described in previous sections.

To start, you must import the Startup symbols into the debugger. From the main menu, select **File→Load** to display the Load window. Under List Files of type, select All Files [*.*]. Select the *sandpoint-startup.sym* file described previously and choose OK.

After loading the symbol information, the debugger likely displays the Append new Directory popup to request the location of the source files. Select the directory where the Startup library source is located. You need repeat this for each of the Startup-related source directories listed at the end of the ***Generating Startup Debug Symbols*** section by choosing **File→Append Source Path...**



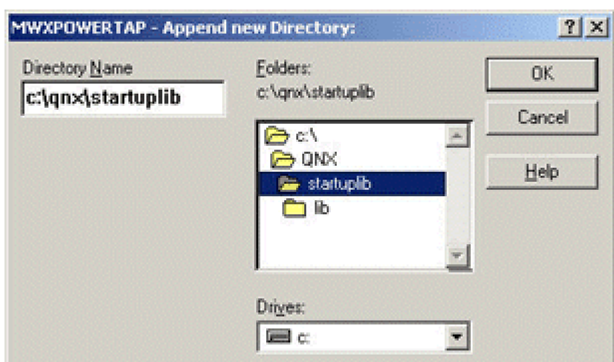


Figure 16 The Load Symbols and Append Directory windows.

To display the **_main**() function in the Startup library, select the Code window (**Displays→Code**). Set the **Mode** to **Source**, type `_main` in the text box, then click the **Display** button. This loads the source into the Code window:

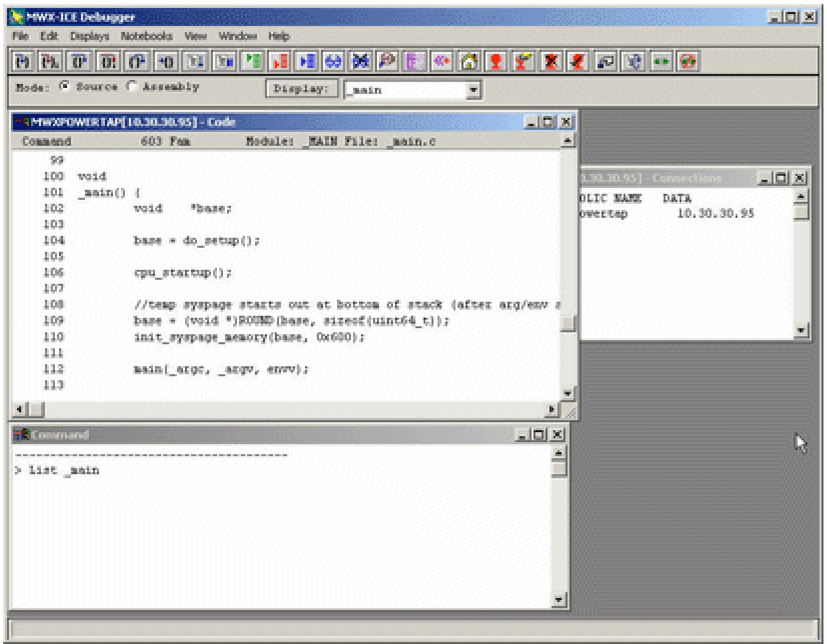
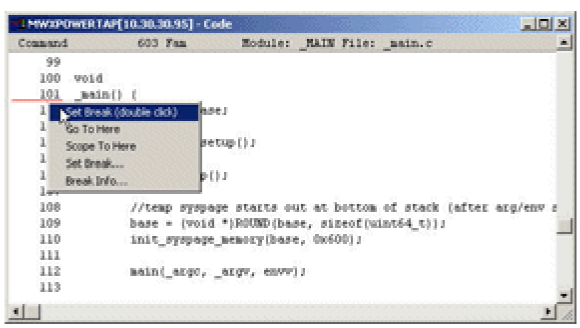


Figure 17 Displaying the _main() function.

You can now set a breakpoint in the **_main**() function.

Note: Ensure that you've set the command **bptype onchip**, and that you've disabled any previous breakpoints.

To set a hardware breakpoint on **_main**(), bring the Code window to the front, right-click over the line number associated with the start of **_main**(), and choose **Set Break**.



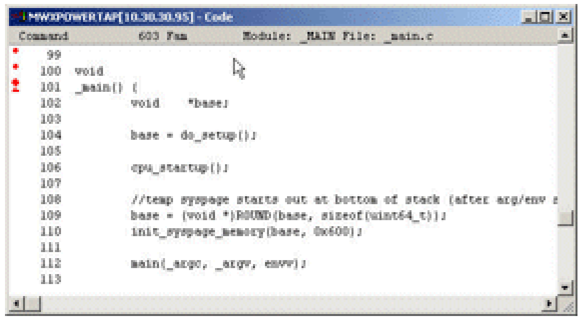
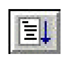


Figure 18 Setting and viewing a breakpoint in the *_main()* function.

To begin debugging the Startup, click the **Reset**,

and **Run**,



buttons. If you've burned the Startup into the Sandpoints flash memory along with the IPL, the debugger stops at the breakpoint. If you have to download the image over a serial connection (via the **sendnto** utility or the Sandpoint ROM monitor), the debugger breaks on this instruction once the image has been loaded into RAM and executed. It's also possible for you to trace through the IPL program to the point where it transfers control over to the Startup.

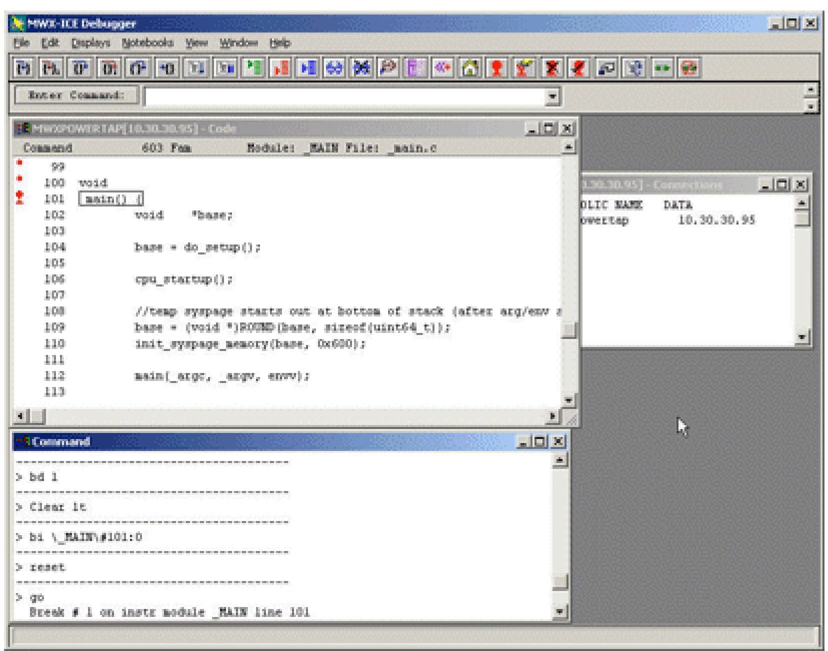


Figure 19 Debugger stopped on the *_main()* function.

You can use the debugger to debug the Startup source code in the same ways described in the **Debugging the Sandpoint IPL** section. For instance, you can let the debugger run to the Sandpoint Startup **main**() function, and then step into the source.

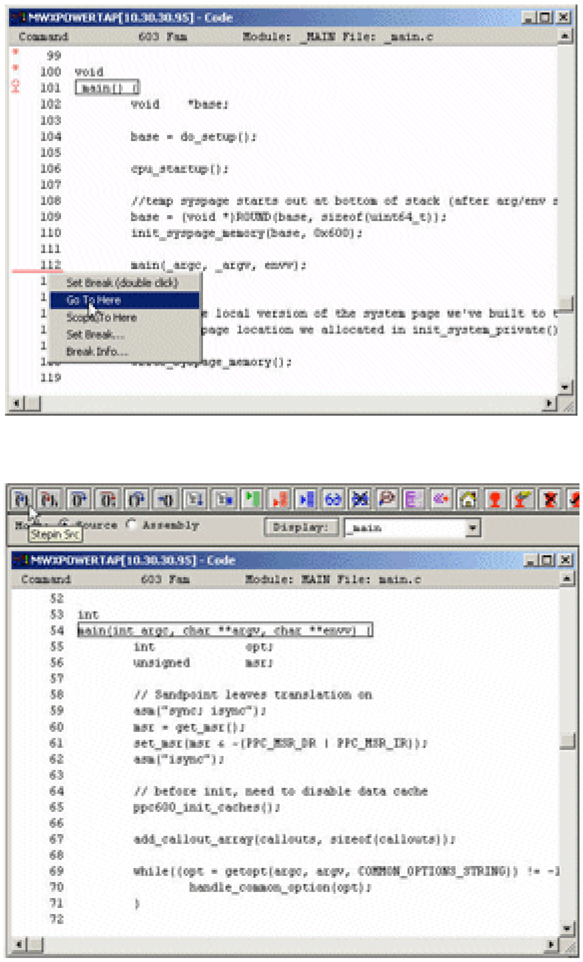


Figure 20 Stepping into the Sandpoint Startup *main()* function.

Once Startup completes initializing the environment, it transfers control over to the QNX Neutrino microkernel, **procnto**. Then, the kernel switches on the CPUs MMU, transferring the system over to virtual memory mode. From this point on, you can use a standard hardware debugger such as the AMC MWX-ICE to step through the kernels instructions, examine registers, etc., but you can't look at kernel or application data, because this memory is accessed in virtual memory mode. At this point however, the target board is up and running, and you can develop and debug user applications (including drivers) by using the standard QNX software debugging agent, **pdebug**, in conjunction with the standard QNX development tools.

The IPL program is responsible for basic setup of the CPU, including the memory controller, and copies the QNX Neutrino image into RAM. It's also possible for you to circumvent the IPL by using a hardware debugger. Hardware debuggers let you transfer an image in ELF or SREC format directly to target memory. In order to transfer a QNX Neutrino image (which includes the Startup program) to a target and run the RTOS, you must first configure the CPU and environment similar to the steps performed by the IPL. This is often done using a script that programs CPU registers and the memory controller before the image is downloaded and executed.

The **pdebug** software debugger requires a free serial port or Ethernet connection on the target board. There are a number of alternatives to debug user applications on deeply embedded targets without free communication ports:

- **ROM Emulator** - Some ROM emulators provide the capability of communication through the emulated ROM. For example, a virtual serial driver is available for QNX targets based on the AMC NetROM, letting you debug applications by using **pdebug** through this virtual serial port.
- **Serial Through JTAG** - A driver is available for the AMC PowerTAP and WireTAP, creating a virtual serial port connection between the host and target hardware over a JTAG connection to be used for debugging.
- **QNX Neutrino RTOS Aware JTAG** - Some hardware debuggers are available with knowledge of the QNX Neutrino RTOS, allowing for debugging of user applications. Contact AMC and Lauterbach for more details.

5. Summary

The software development and debugging process differs between QNX Neutrino and conventional embedded RTOSs. The QNX virtual memory based microkernel architecture protects the system from unstable applications and drivers. The modular QNX Neutrino architecture allows for reliable software debugging. You can start and stop QNX applications and drivers as desired on a running system, and you can debug them using the software debug agent **pdebug**. Hardware debuggers can also be valuable tools in the development of embedded systems, but are generally used differently than with other embedded RTOSs. Hardware debuggers are most useful in debugging software that's running in physical memory mode, such as QNX IPL, Startup, and kernel callouts.

6. References

1. **System Architecture**, QNX Software Systems.
2. **Building Embedded Systems**, QNX Software Systems.
3. **The New Business Imperative: Achieving Shorter Development Cycles while Improving Product Quality**, Paul N. Leroux, QNX Software Systems.
4. **Embedded Systems Design An Introduction to Processes, Tools, & Techniques**, Arnold S. Berger, CMP Books.
5. **An Embedded Software Primer**, David E. Simon, Addison Wesley.