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.Ink 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:

MWX-ICE Debugger	1200011		Roliss		
			1	12	
Force OS Download: C Yes G No					
R MWXPOWERTAP[Not Connected] - Code		_101×	1		
Command 603 Fam Hodule: NUME File: NUME Not In Source Module					
	Connect	lions			
	CON TWO	PE SYMBOLIC	: NAME	DATA	-
					-
			-		-
			1000		
			1		
E MWXPOWERTAP[Not Connected] - Command	1253 1202	_101×		N	
NOTE: Startup file: startup.inc, not found. continuing on Debugger ready for commands		Í		-3	
[

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 \rightarrow 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 \rightarrow 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.

Define Ethernet Connection	<
Symbolic Name: my_powertap	-
Host Name: 10.30.30.95	
Cancel ?	
- MWZPOWERTAP(10.30.30.95) - Command	<u>.</u>
MOTE: Startup file: startup.inc, not found. continuing on Debugger ready for commands	1
> config my_powertep, ETHEPHET, 10.30.30.95	
<pre>> connect my powertmp Initializing the debugger Opening Ethernet emulator named 10.30.30.95 File found is Ci\PowerTMP\mac\powertmp\emulofg.det Downloading JTAG file Ci\PowerTMP\mac\powertmp\83201.jtsg JTAG file download completed</pre>	
The INITREGS values have been read from the file	
Initialization Finished	

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** \rightarrow **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.

MWXPOWERTAP - Enter Fi	le to Load:	<u>?</u> ×
File pame: ipl-sandpoint.	Eolders: c:\qnx	OK.
ipl-bin ipl-sandpoint map ipl-sandpoint_g netromgnx_May2.tgz osaware_spc_g QNV/netrompkgMarch2 samplejtagdebugJune4	C:\ QNX clipsesamp sandpoint sandstartup clipsesamp sandstartup sandstartuplib	Help
List files of type: All Files [11]	Drives:	-

MWXPOWERTAP - Append	new Directory:	? ×
Directory <u>Name</u> c:\qnxsdk\host\win3	Eolders: c:\\lbs\src\hardware\ipl\lb ipl ipl ipl ipl ipl ipl ipl ipl ipl ipl	OK Cancel Help
	Drives:	

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.

S HWXPO	wtRTAP[10.30.30.95] - Code	
Command	603 Fam Module: MAI	IN File: main.c
18	unsigned int image;	
19	int main(void)	
20		
21	(
22	//	
23	// Init superIO so the serial ports	s will work
24	11	
25	init_icache();	
26	init_ns87308();	
27		
28	//	
29	// locate the image	
30	// Image is located at 0xff800000	
31	11	
32		

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.

MWXPOW	ERTAP[10.30.30.95] - Code	
Command	603 Fan	Module: MAIN File: main.c	
19 1	nt main(void)		
20			
21 (
22 /	1		
23 /	/ Init superI0 :	so the serial ports will work	
24 /	1		
25	init_ice	che () J	
Sel	Break (double click)	08())	
Ga	To Here		
50	ipe To Here-C		
50	t Break	*	
Bre	sak Info	6 at 0xff800000	
** *	*		
32			
33	init_829	0(0xfe0003f8, 115200, 1843200, 16))	-
d 1			•
- Internal			

-S MWXP0	WERTAP[10.30.30.95] - Code	_1012
Connand	603 Fan Module: MAIN File: main.c	
19	int main(void)	
20		
21	1	
22	//	
23	// Init superIO so the serial ports will work	
24	//	1
25	init icache[];	
26	init_ns87308();	
27		
28	//	
29	// locate the image	
30	// Image is located at 0xff8000000	
31	//	
32		
33	init_8250(0xfe0003f8, 115200, 1843200, 16);	
		1
MWXDR	wtR1AP[10.30.30.95] - Command	_10);
eeset		
bist m	ain	
go /RA	IN\#25:0	
(Teap)	Break module MAIN line 25	

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.

- MWXPC	rwERTAP[10.30.30.95] - Code	
Command	755	Module: INIT_ICACHE File: init_icache.c
17	void init_icache(void)	(
18	unsigned	spr_hid0;
19		
20	asa("synd")J	
21	spr_hid0 = get	_spc(PPC603_SPR_HID0) J
22		
23	// Turn on I-co	ache
24	set_spx(PPC603)	_SPR_HIDO, spr_hid0 PPC603_SPR_HIDO_ICE);
25	asa("sync") J as	Sh("1Synd") J
26		
27	// Flush it	
28	set_spr(PPC603	_SFR_HIDO, spx_hid0 PFC603_SFR_HIDO_ICE PFC60
29	asa("sync")) as	sa ("isynd") J
30		
31	// Leave it on	
32	set_spr(PPC603	_SFR_HIDO, spx_hid0 (PPC603_SFR_HIDO_ICE) J
33	asa("sync")) as	sa("isynd"))
. 34	1	<u>بر</u>
4		• k

MWXPC	WERTAP[10.30.30.95] -	Code	
Command	755	Module: MAIN File: main.c	
19	int main(void)		
20			
21	(
22	11		
23	// Init superIO so	the serial ports will work	
24	11		
25	init_icache	0.1	_
26	init ns873	10()3	
27			
28	11		
29	// locate the image	a superior a	
30	// Image is located	at 0xff800000	
31	11		
32			
33	init_8250(0	hte0003f8, 115200, 1843200, 16);	
34			
35	put_byte(0)	(fe0003f8, 'D'))	
36	put byte(0)	(fe0003f8, 'o');	-
•			• //

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.

'canand	603 Far	Nodule: INACK DOWNLOAD 8250 Files in	ane downla
20	COUR FRM	abouter mana_sound_ouse Fire. in	ade opena
0.3			
64	//		
65	// Set Bags	rate	
66	//		
67	10 102 1	en la de la de la dela de la dela dela del	
68	value = clk/	(Dang . gialaor))	
Set Br	sak (double ciki)		
Go To	Here he		
Scope	To Here 50	REASHNED_DC, DCH_DLAB, DCH_DLAB) J	
Set Br	nak	iress+REG_DLO, Owff, value 4 Owff)J	
Break	Info	iress+REG_DL1, OKTE, value >> 8);	
***	and a second	RESSHREG_DC, DCR_DLAB, 0)1	
75			
76	11	10	
77	// Set data	bits to 8	
78	11		
			and the second sec
			1
MWXPOWER	TAP[10.30.30.95] - C	ode	<u>الا</u> د املی
MW30POwer command	TAP[10.30.50.95] - C 603 Taa	ede Rodule: IMAGE_DOWNLOAD_0250 File: 1m	L L L L L L L L L L L L L L L L L L L
MWEPOWER command 63	TAP[10.30.30.95] - Co 603 Fan	ode Module: INAGE_DOWNLOAD_0250 File: im/	
MWEPOWER commend 63 64	TAP[10.30.50.95] - C 603 Faa //	ode Rodule: IRAGE_DOWNLOAD_0250 File: im	ر م در ایر sys_down1
MWSROWER canand 63 64 65	[AF[1030.50.95] - C 603 Tam // // Set Baud	nde Module: INAGE_SOUNIOAD_0250 File: im Tote	
NWERDWER can and 63 64 65 66	(AF(10.30.30.35) = C 603 Fan // // Set Baud //	ode Module: INAGE_DOWNLOAD_0250 File: ind Fate	 age_down1_
MW3POwter commod 63 64 65 66 67	TAF[10.30.30.95] - C	ede Module: INAGE_DOWNLOAD_8250 File: im rate	<u>د</u> ار مرتب مرتبع مرتب مرتبع
MW20000000 63 64 65 66 67 68	[AF[1030.30.95] - C 603 Fam // // Set Baud // value = clk/	nde Rodule: INAGE_DOWNLOAD_0250 File: im rate (baud * divisor);	ی م age_down1a
Conserved Conserved 63 64 65 65 66 65 66 65 66 69	7A4{10.30.30.95]=C 603 Fan // // Set Baud // value = clk/	ode Rodule: INAGE_DOWNLOAD_0250 File: im rate (beud * divisor);	sye_down1
MW2POwer casesard 63 64 65 66 67 68 69 70	TAF[10.30.30.95] - CC	nde Rodule: IMAGE_SOUNIOAD_0250 File: im rate (baud * divisor);	LINX sge_down12
Comment Comment 63 64 65 66 67 68 68 69 70 71	7A9 (10.30.30.95) - C 603 Tam // // Set Baud // value = clk/ met Bottladd	ode Module: INAGE_DOWNLOAD_8250 File: im rate (baud * divisor); ress=REG LC, LCE DLAB, LCE DLAB);	⊥. sge_domi_
MW3POwer connard 63 64 65 66 66 67 68 69 70 71 72	TAF[10.30.30.95] = C 603 Fan // // Set Baud // value = clk/ set_port(add ast port(add	ode Module: INAGE_DOWNLOAD_0250 File: im rate '(baud * divisor); iress+REG_DG, LGP_DLAS, LGP_DLAS); iress+REG_DG, Orf, value 4 Orff;	≥ . spe_domia
AWASPOWER (samand 63 64 65 66 67 68 69 70 71 72 23	<pre>////(10.30.30.35) = C 603 Fam // // Set Baud // value = clk/ set_port(add set_port(add set_port(add set_port(add set_port(add)</pre>	<pre>cdc Module: INAGE_DOWNLOAD_0250 File: im rate '(baud * divisor); iress=REG_DLC, LCP_DLAS, LCP_DLAS); iress=REG_DLD, 0xff, value + 0xff;; iress=REG_DLD, 0xff, value +> 0x;;</pre>	,×
MM2D/DurieR Constant 63 64 65 66 66 66 69 70 71 72 73 74	<pre>TAF[10.30.30.95] = C 603 Fam // // Set Baud // value = clk/ set_port(add set_port(add set_port(add set_port(add set_port(add set_port(add)</pre>	ode Rodule: INAGE_DOWNLOAD_0250 File: im rate (baud * divisor); iress+REG_LC, LCR_DLAB, LCR_DLAB); iress+REG_DLD, Gwff, value + Gwff); iress+REG_DLD, Gwff, value > 0;; iress+REG_LD, UCR DLAB, 0;;	ی اے ا syg_downia
AW2200 will a command 63 64 65 66 67 68 69 70 71 72 73 74 75	<pre>IAF[10:30:30:35] = c 603 Fam // // Set Baud // value = clk/ set_port(add set_port(add set_port(add set_port(add</pre>	<pre>cdc Rodule: INAGE_DOWNLOAD_0250 File: im rate (baud * divisor); iress=REG_LC, LCP_DLAB, LCP_DLAB); iress=REG_DL0, 0xff, value < 0xff); iress=REG_LC, LCP_DLAB, 0);</pre>	⊥ . spe_down1=
MW3500w18 command 63 64 65 66 67 68 69 70 71 72 73 74 75 76	TAP(10.30.30.95) = C 603 Tam // // Set Baud // value = clk/ set_port(add set_port(add set_port(add set_port(add	ode Rodule: IMAGE_DOWNLOAD_0250 File: im rate '(beud * divisor); iress=REG_LC, LCR_DLAB, LCR_DLAB); iress=REG_DL, 0xff, value 4 0xff); iress=REG_LL, 0xff, value >> 0;; iress=REG_LC, LCR_DLAB, 0);	, <u>i</u> li × sge_domi
AWODOWER cemand 63 64 65 66 67 68 69 70 71 72 73 74 73 74 75 75 76 72 73	<pre>IAF[10:30:30:35] = C 603 Tam // // Set Baud // value = clk/ set_port(add set_port(add set_port(add set_port(add set_port(add set_port(add set_port(add set_port(add) set_port(add)</pre>	<pre>cdc</pre>	±. ≂O≯

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

15 Breakpaints			
Address	Type	Sysbol	Countend
T 97900870	Tars .r	VINTUE DOIN	1040_8250\ini:_5250\#88 \TNACK_DCHNLCAD_8250\#88:0
1			2

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**.

63		and the second sec	MACE DOWNLOAD	8250 File:	inage downle
0.0					
09	11 fee bout				
60	// Sec Dava	tate			
60	11				
67	malana a sila i	Thomas in the second			
68	value = cix/	(5405 * 61918	0813		
69					
70					
71	set_port(add	cess+REG_LC,	LCR_DLAB, LCR	DLAB) J	
72	set_port(add	cess+REG_DLO,	OKEE, value	e OKEE) J	
73	set_port(add	cess+REG_DL1,	Oxff, value :	>> 8)1	
74	set_port(add	cess+REG_LC,	LCR_DLAB, 0)J		
75					
76	11				
77	// Set data 1	bits to 8			
78	11				
					•

Command	603 Fan	Module: IMAGE_DO	WNLOAD_8250 File:	image_downl.*
63 64 65 66 67	// // Set Baud rate //	d * distanti		
69 70 71 72 73 74 75 76 77 78	set_port(address set_port(address set_port(address set_port(address // // Set_data_bits //	Print (disuble citit) Monitor Inspect Print Type Print String Print * Print & Set New Value Set to 0 Increment Decrement	<pre>bg PR_DLAB)J = 4 0xtt)J = > 0)J)J</pre>	
16 > print clk 1843200	199 (1020230385) er centre	Set Break Scope To Set Instr Break. Set Instr Break Add	*	-
> print beud 115200				-

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.

	Address	Туре	Symbol	Conmand
2	FFF00E70	Instr	\IMAGE_DOWNLOA	D_8250\init_8250\#68
	Disable Break (double	e click) 📐		
	Clear Break	N		
	Modify			
	Copy			
	Scope To			

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** \rightarrow **Memory**. The Memory window lets you enter an address to view system memory. This interface also lets you modify the contents of memory.

lGe	ieral Purpose	6.7.8	
IP	fff00e70	MIR	00000040 -
LR	£££00684	CTR	00000000
CR	20000000	XER	00000000
RO	£££00684	R16	fed7fdfe
R1	00001178	R17	4776bff3
9 2	effeff77	R18	676cbff3
R3	£e0003£8	R19	7fdbff72
R4	0001c200	R20	6ffdfffe
RS	001c2000	R21	3cccccd7
R6	00000010	R22	32292272
R7	00000061	R23	6fdffd6e
R8	ETETETES	R24	95547555
89	£e000154	R25	6£45£4£7
P10	00000030	R26	00000000
R11	fe00015c	R27	b7dffffs
P12	deeseee3	R28	£e0003£8
R13	67647464	R29	fe0003f8
P14	afdffdfd	R30	ffeddff7
			•

Ptart Adds		: [n f f	100	644	1		•]	Di	(sp)	lay	Γ		cel	VIBICLE	1000
Memory	1	- 15	Ъ.,	4	86	. 3	87.				- 27	82.					
Address	0	1	2	3	4	5	6	7	8	9	A	8	с	Ð	E	F	-
FFF00644	94	21	TT	EO	70	08	02	A 6	93	A1	00	14	93	C1	00	18	
FFF00654	93	E1	00	10	90	01	00	24	-0	77	73	59	-00	77	n	95	
FFF00664	30	60	72	00	60	63	03	F8	30	80	00	01	60	84	C2	00	
FFF00674	30	A0	00	10	60	4.5	20	00	38	CO	00	10	40	00	07	CD	١.,
FFF00684	30	60	72	00	60	63	03	Fö	38	80	00	44	48	00	08	70	-
FFF00694	30	60	72	00	60	63	03	FB	38	80	00	67	48	00	08	60	
FFF006A4	30	60	72	00	60	63	03	10	38	80	00	77	48	00	08	5D	
FFF00684	30	60	72	00	60	63	03	F8	38	80	00	68	48	00	08	40	
FFF006C4	30	60	12	00	60	63	03	10	38	80	00	60	48	00	08	30	-

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** \rightarrow **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.

MWXPOWERT	AP[10.30.30.95]	Code		
Command	755	Module:	MAIN File: main.c	
fff000fer	00000000	INVLD		
entry_veci				
£££001001	48000004	- b	system peset	
system_reset:				
£££00104:	3c000000	addis	x0,0,0x0	
fff00108:	7c6000a6	BÉBSE	£3	
fff0010c1	3c80ffff	eddis	x4,0,0xffffffff	
fff00110:	6084ffcf	OXI	z4,z4,0xffof	
fff00114:	7c632038	end	13,13,14	
fff00118:	7c600124	hThSE	x3	
fff0011ci	4c00012c	isync		
fff001201	7c0004ec	synd		
fff00124:	3ceOfec0	eddis	x5,0,0xfffffec0	
fff00128:	3oc0fee0	eddis	x6,0,0xfffffee0	
fff0012c1	3d00fe00	eddis	z8,0,0xfffffe00	
fff00130:	34200000	eddis	£9,0,0x0	
fff00134:	7d1143a6	atops	sprg1,r8	
fff00138:	7cb243a6	ALSO C	SDE02,25	*
4				• 4

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.

MWXPOWERT	AP[10.30.30.95]-	Code	11 ANI ANI 473		_101 ×
Commend	785	Module:	INIT_ICACHE File:	init_icache.c	-
fff00188:	38600000	addi	£3,0,0x0		
:lbinval:					
fff0018c:	7c001a64	tlbie	E3		
fff00190:	7c0004ac	aync			
fff00194:	38631000	addi	r3,r3,0x1000		
fff00198:	7c034800	cape	r3,r9		
fff0019c:	4180fff0	blt	tlbinwal		
fff001a0:	38000000	addi	z0,0,0x0		
fff001a4:	3c200000	addis	r1,0,0x0		_
fff001a8:	60212000	ori	r1,r1,0x2000		
fff00lac:	9401ffb8	atanu	r0,-0x0048(r1)		
fff001b0:	48000495	b1	0xfff00644		
	** MODULE: INIT	ICACHE ****	******		
1 /*					
2 * 0	opyright 2001,	QMC Software	Systems Ltd. Unpub	lished Work A	11 Righ
3 * 8	eserved.				
4 *					
. 5 * T	his source code	contains co	nfidential informat	tion of QMDC So	ftware .
•					2

MWXPOWERT	AP[10.30.30.95]-	Code		1991.02		
Command	755	Module:	INIT_ICACHE	File:	init_icache.c	-
fff001881	38600000	eddi	£3,0,0x0			
lbinvalt						
fff0018c1	7c001a64	tibie	£3			
fff001901	7c0004ec	sync				
£££001941	38631000	eddi	£3,£3,0x10	000		
fff001981	7c034800	capw	13,19			
fff0019e1	4180fff0	bit	tlbinval			
fff001a01	38000000	edd1	E0,0,0x0			
fff001e41	3c200000	eddis	£1,0,0x0			
fff001a81	60212000	OEI	£1,£1,0x20	000		
fff001act	9401ffb8	stwa	£0,-0x0040	8(11)		
fff001b01	48000495	b1	0xfff00644	4		
Set Break ((double click)	T_ICACHE *****				
Go To Here	•					
Scope To P	tere 10	QNX Software	Systems Ltd.	. Unpro	blished Work Al	1 Righ
Set Break						
Frenk Info						
Dreat pro-		e contains cos	fidential in	nforme	tion of QMX Set	tware
6 * L	td. (QSSL). An	y use, reprodu	action, modif	ficati	on, disclosure,	distr
7 * 0	r transfer of	this software,	OE any soft	twaze	which includes	OE 15
			-			

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.

MWXPOWERT	AP[10.30.30.95] -	Code		
Command	755	Module:	INIT_ICACHE File: init_icache.c	
fff001881	38600000	eddi	x3,0,0x0	
libinvalı				
£££0018c1	7c001a64	tibie	x3	
fff001901	7c0004ac	sync		
£££001941	38631000	eddi	£3,£3,0x1000	
fff001981	7c034800	CRDW	13,19	
fff0019cs	4180fff0	bit	tlbinval	
fff001a01	38000000	eddi	£0,0,0x0	
fff001e4i	3c200000	edd1#	£1,0,0x0	
fff001a81	60212000	OEI	r1,r1,0x2000	
fff001acs	9401ffb8	stwa	z0,-0x0048(z1)	
fff001b01	48000495	bl	0xfff00644	
1 /* 2 * C 3 * R 4 * 5 * T 6 * L	opyright 2001, eserved. his source code td. (03SL). Any	ONX Software contains con y use, reprod	Systems Ltd. Unpublished Work Al nfidential information of QMX Sof uction, modification, disclosure,	l Righ tware distr
· · · ·	E LEANSCEE OF	this sortware,	, or any sortware which includes	01 15

CGE ECH 200	Module:	MAIN File: main.c	
19 int main(void) 20 21 (
fff00644: 9421ffe0	stava	<pre>x10x0020(x1)</pre>	
fff00648: 7c0802a6	af spc	£0,1E	
fff0064c1 93a10014	stw	£29,0x0014(£1)	
fff00650: 93c10018	stw	£30,0x0018(£1)	
fff00654: 93e1001c	stw	£31,0x001c(£1)	
fff006581 90010024 22 //	stw	£0,0x0024(£1)	
23 // Init superI0 24 //	so the serial p	orts will work	
25 init_ice	t () sdoe		
dint			
26 init_nst	b1 97308();	init_icache	
27 28 // 29 // locate the in	bl	init_n#87308	

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** \rightarrow **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** \rightarrow **Append Source Path...**

Eolders:	OK
c:\qnx	Cancel
C:\ C qnx C eclipsesamp Sandpoint Sandstartup Sandstartuplib	
Drives:	
	Eolders: c:\qrx @ c:\ @ qrx @ eclipsesamp @ sandstartup @ sandstartuplib Drives:

Directory <u>N</u> ame	Eolders:	OK
c:\qn×\startuplib	c:\qrx\startuplib	Cancel
	CN QNX Startupilb	Help
		1
	Drives:	
	C	-

Figure 16 The Load Symbols and Append Directory windows.

To display the **_main()** function in the Startup library, select the Code window (**Displays** \rightarrow **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:

Maxpow	IRTAP[10.30.30.95]- Code		
99		-	
100 1	bid	1 30 30 951	Consections Into
101	main() (DITO NUMP	
102	void *base;	DETC MARE	20.00.00.01
103		prercap	10120120120
104	base = do_setup();		
105			
106	cpu_startup();		
107			
108	//temp syspage starts out at bottom of stack (after arg/env a		
109	base = (void *)ROUND(base, sizeof(uint64_t));	1	
110	init_syspage_memory(base, 0x600);	-	
111		-	
112	main(_argc, _argv, envv);	-	
113			
	1		
Comman			
Concession of the same of			
	-		

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**.

MWXPOWER	TAP[10.30.30.	5] - Code	
Command	603 Fr	Module: _MAIN File: _	sain.c
99 100 voi 101 ma 1 Sope 1 Sope 1 Set Br 1 Break	d in() (esh (double cici) Hore To Hore esh Info	nsej petup();	
108 109 110 111 112 113	//temp base = init_sy main(_d	<pre>syspape starts out at bottom ((void *)ROUND (base, sizeof(uis spape_memory(base, 0x600)) spot, _argv, emvv))</pre>	of stack (after arg/env s st64_t));
			,

Command	603 Fan Module: MAIN File: main.c
99 100 void 101 _main() 102 103 104 105 106 107 108 109 110 111 111	<pre> (void *base; base = do_setup(); cpu_startup(); //temp sympose starts out at bottom of stack (after arg/env e base = (void *)ROUD(base, sizeof(wint64_t)); init_sympose_bemory(base, 0x600); main(_argo, _argo, envo); </pre>

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

To begin debugging the Startup, click the Reset,

and Run.

		52
		≣↓

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.

Exter Comm	at [-	
MHOPOWERI	AP[10.30.30.95] - Code		
braas	603 Fam Bodale: _MAIN File: _main.c		
99		and the second s	1
101 201	n() (ACCREDITE OF	Longestore
102	moid Phone:	OLIC NAME	DATA
103	tora ener	owertap	10.30.30.95
104	base = do setup();		
105			
106	cpu startup();		
107	5 - 500		
108	//temp syspage starts out at bottom of stack (after arg/env		
109	base = (void *)ROUND(base, sizeof(uint64_t));		
110	init_syspage_memory(base, 0x600);	-	
111		-	
112	main(_argc, _argv, envv);	1000000000	
113			
			N
Command		×	
		-	
bd 1		000000000000000000000000000000000000000	
		100000000000000000000000000000000000000	
Clear 10			
A L MAYNE	101.0		
DI 1_HAIA1	101:0		
reset			
		- 100000000000	

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.

Comand	603 Fan	Module:	NATE F11	e: nain.c	
00					
99					
100 901	<u>a</u>				
101 188	in() (
102	void *bas	sej.			
103					
104	base = do_se	tup();			
105					
106	cpu_startup	10.1			
107					
108	//teap system	oce starts of	ut at bots	on of stack (ad	tter arg/env
109	base = (voi)	+) POUND (bas	se, sincof	Duint64 th1:	
110	init erman	nenory than	e. 0x6001		
111	and "alabate	Country in an	.,		
112	maind arms.				
1 540	and the state of the	-arget enter			
- Det on	rak (double citor)				
60.10	Plere				
1 ScopeCto Here		local versio	on or the	slacer babe as.	As putit co
1 Set Bre	sak P	age location	we alloce	ted in init_sys	stem_private(
1 Break.	info				
1	assas alaby	he memorA();			
119					



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.