

NOTE

0xxxx0000 is the address printed by U-Boot at line "Now running in ram". You can also see this address in the **Disassembly** view and observe the current address space you are in.



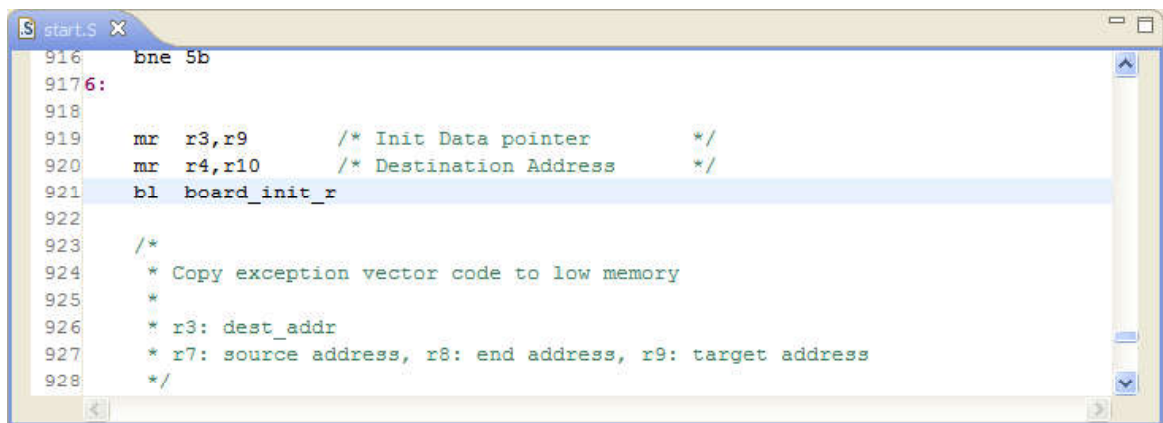
2. From the **Debug** view toolbar, select the **Instruction Stepping Mode** () command.
3. From the **Debug** view toolbar, select the **Step Into** () command to step into `blr`.

Figure 104: U-Boot Debug - Running in RAM



4. Deselect the **Instruction Stepping Mode** command when the instruction pointer is at `in_ram`.
You can now do source-level debugging and set breakpoints in all the RAM area, including `board_init_r`. See [Points to remember](#) on page 267 for more details.

NOTE

You can enter the `board_init_r`, `nand_boot`, and `uboot` functions. Beginning with the `uboot` function, the second image is relocated to RAM at `0x11000000` and you begin to execute the entire code again from RAM address space. See [Points to remember](#) on page 267 to avoid any debugging issues.

NOTE

Before closing the debug session, change back the alternate load address to flash address space by issuing the `setpicloadaddr 0xFFFF40000` command in the **Debugger Shell**. Now, you do not need to manually set it from the **Debugger Shell** in Stage 1.

7.7 Debugging the Linux Kernel

This section shows you how to use the CodeWarrior debugger to debug the Linux kernel.

The Linux operating system (OS) works in two modes, *kernel mode* and *user mode*. The Linux kernel operates in kernel mode and resides at the top level of the OS memory space, or *kernel space*. The kernel performs the function of a mediator among all the currently running programs and between the programs and the hardware. The kernel manages the memory for all the programs (processes) currently running and ensures that the processes share the available memory such that each process has enough memory to function adequately. In addition, the kernel allows application programs to manipulate various hardware architectures via a common software interface.

User mode uses the memory in the lowest level of the OS memory space, called the *user space* or the application level. At the application level, a program accesses memory or other hardware through system calls to the kernel as it does not have permission to directly access these resources.

Debugging the Linux kernel involves the following major actions:

1. [Setting Up the Target Hardware](#) on page 284
2. [Installing the Board Support Package \(BSP\)](#) on page 286
3. [Configuring the Build Tool](#) on page 287
4. [Configuring the Linux Kernel](#) on page 287
5. [Creating a CodeWarrior Project using the Linux Kernel Image](#) on page 289
6. [Configuring the kernel project for debugging](#) on page 290
7. [Debugging the kernel to download the kernel, RAM disk, and device tree](#) on page 301
8. [Debugging the kernel based on MMU initialization](#) on page 302
9. [Debugging the kernel by attaching to a running U-Boot](#) on page 305

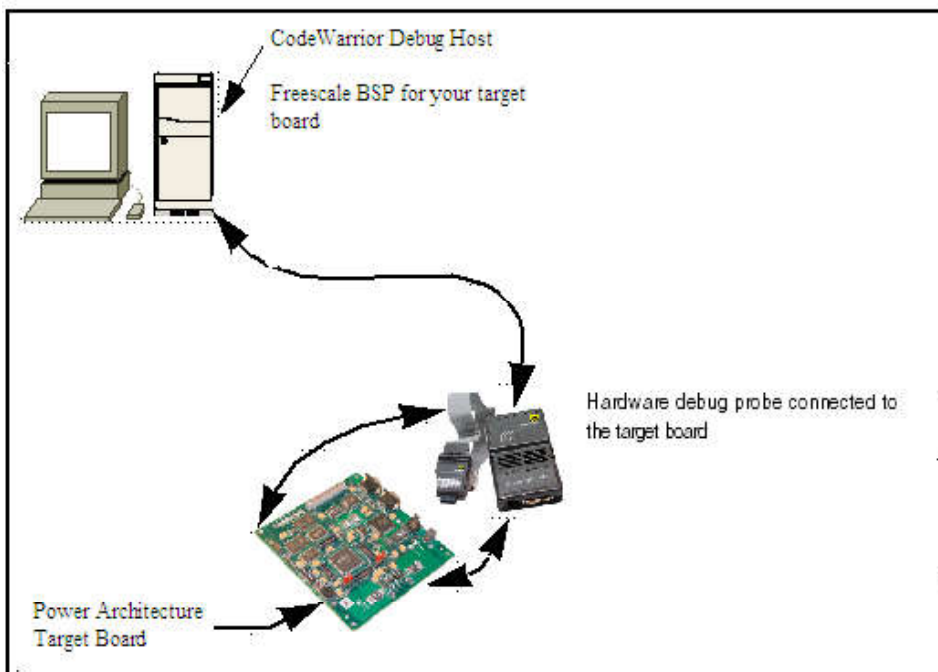
7.7.1 Setting Up the Target Hardware

Before you use the CodeWarrior IDE to debug the Linux kernel, you need to set up the target hardware.

One requirement of the setup is to have a debug probe connected between the CodeWarrior debug host and target board.

The figure below illustrates the setup required to use the IDE to debug the Linux kernel running on a Power Architecture target board.

Figure 105: Setup for Kernel Debugging Using the CodeWarrior IDE



Connect the hardware debug probe between the target board and CodeWarrior debug host. Kernel debugging is possible using a Linux-hosted or Windows-hosted CodeWarrior installation. There are a variety of debug

probes. The current kernel debugging example uses the USB TAP. Connection information for other debug probes can be determined from documentation provided with the probes.

The following subsections provide the steps to set up the target hardware:

1. [Connect USB TAP](#) on page 285
2. [Establish a Console Connection](#) on page 285

7.7.1.1 Connect USB TAP

This section explains how to connect the USB TAP between the CodeWarrior debug host and target board.

To connect the USB TAP, perform these steps:

1. Ensure that the power switch on the target board is OFF.
2. Connect the square end (USB "B" connector) of the USB cable to the USB TAP.
3. Connect the rectangular end (USB "A" connector) of the USB cable to a free USB port on the host Linux machine.
4. Connect the ribbon cable coming out of the USB TAP to the 16-pin connector on the target board.
5. Connect the power supply to the USB TAP.

7.7.1.2 Establish a Console Connection

You need to establish a console connection before applying power to the target board, so that boot messages can be viewed in a terminal window.

Establishing the console connection allows you to:

- View target generated log and debug messages
- Confirm successful installation of the bootloader (U-Boot)
- Use the bootloader to boot the Linux OS
- Halt the booting of the Linux OS

The bootloader receives keyboard input through a serial port that has default settings 115,200-8-N-1.

Follow these steps to establish a console connection to the target hardware.

1. Connect a serial cable from a serial port of the CodeWarrior debug host to a serial port of the target board.
2. On the CodeWarrior debug host computer, open a terminal-emulator program of your choice (for example, `minicom` for a Linux host).
3. From the terminal-emulator program, open a console connection to the target hardware.

Use the connection settings given in the table below.

Table 132: Terminal Connection Settings

Name	Setting
Baud rate	115, 200 bits per second
Data bits	8
Parity	None
Stop bits	1
Flow control	Hardware

NOTE

See the board specific README file inside the stationery wizard project to find out more details on the serial connection settings, changing the serial port on the board, and the type of serial cable to use.

4. Test the connection by turning on the test board with the power switch and viewing the boot messages in the console connection.

7.7.2 Installing the Board Support Package (BSP)

This section describes installation of a BSP on a Linux computer.

NOTE

The BSP versions keep changing frequently. For different BSP versions, you might encounter build environments based on Itib, bitbake, or other tools. The subsequent sections will describe necessary procedures and use specific examples from real Freescale BSPs for illustration. The examples in these sections will need to be adapted based on the BSP versions or build tools you are currently using.

To install a BSP, perform the following steps:

1. On the Linux computer, download the Board Support Package (BSP) for your target hardware to install kernel files and Linux compiler toolchains on your system.

Board Support Package image files for target boards are located at <http://www.freescale.com/linux>.

2. Download the BSP image file for your target board.

NOTE

You will need to log in or register to download the BSP image file.

The downloaded image file has an `.iso` extension.

For example,

```
QorIQ-DPAA-SDK-<yyyymmdd>-yocto.iso
```

3. Mount the image file to the CDRom as root, or using "sudo":

```
<sudo> mount -o loop QorIQ-DPAA-SDK-<yyyymmdd>-yocto.iso /mnt/cdrom
```

NOTE

`sudo` is a Linux utility that allows users to run applications as root. You need to be setup to run `sudo` commands by your system administrator to mount the BSP image files.

4. Execute the BSP install file to install the build tool files to a directory of your choice, where you have privileges to write files:

```
/mnt/cdrom/install
```

NOTE

The BSP must be installed as a non-root user, otherwise the install will exit.

5. Answer the questions from the installation program until the file copy process begins.

You will be prompted to input the required build tool install path. Ensure you have the correct permissions for the install path.

6. Upon successful installation, you will be prompted to install the ISO for the core(s) you want to build.

For example, if you want to build the SDK for P4080, that is a e500mc core, then you have to install the ISO images for e500mc core:

```
c23174e5e3d187f43414e5b4420e8587 QorIQ-SDK-V1.2-PPCE500MC-20120603-yocto.iso.part1  
292c6e1c5e97834987fbb5f69635a1d QorIQ-SDK-V1.2-PPCE500MC-20120603-yocto.iso.part2
```

NOTE

You can see the SDK User Manual for instructions about how to build the BSP images and run different scenarios from the `iso/help/documents/pdf` location.

7.7.3 Configuring the Build Tool

After installing the BSP, you need to configure the build tool and build the Linux kernel and U-boot images for CodeWarrior debug.

For more information on configuring the build tool, see the SDK User Manual from `iso/help/documents/pdf`.

7.7.4 Configuring the Linux Kernel

After you complete the BSP configuration, configure the Linux kernel to enable CodeWarrior support.

To configure the Linux kernel, perform the following steps:

1. Launch a terminal window and navigate to the `<yocto_installtion_path>/build_<board>_release` folder.
2. Execute the following command to get a new and clean kernel tree:

```
bitbake -c configure -f virtual/kernel
```

3. Configure the Linux kernel using the various configuration options available in the kernel configuration user interface. For example, run the following command to display the kernel configuration user interface:

```
bitbake -c menuconfig virtual/kernel
```

The kernel configuration user interface appears.

4. CodeWarrior supports both SMP and non-SMP debug. To change the default settings, you can make changes by selecting the Processor support options.
5. To run a monolithic kernel, you do not need to enable loadable module support. However, during the debug phase of drivers, it is easier to debug them as loadable modules to avoid rebuilding the Linux kernel on every debug iteration. If you intend to use loadable modules, select the **Loadable module support** menu item.
6. Select the **Enable loadable module support** option.
7. Select the **Module unloading** option.

NOTE

If you want to use the `rmmod -f <mod_name>` command for kernel modules under development, select the **Forced module unloading** option.

8. Select **Exit** to return to the main configuration menu.
9. Select **Kernel hacking**.

10. Select **Include CodeWarrior kernel debugging** by pressing Y. Enabling this option allows the CodeWarrior debugger to debug the target. Select other desired configuration options for Linux kernel debug.
11. Select **Exit** to return to the main configuration menu.
12. Select the **General Setup** option.
13. Select **Configure standard kernel features (expert users)** and ensure that the **Sysctl syscall support** option is selected.
14. If you are using the Open Source Device Tree debugging method, under the **General Setup > Configure standard kernel features (expert users)** option, then select:
 - Load all symbols for debugging/ksymoops.
 - Include all symbols in kallsyms.

NOTE

These settings are optional. They aid the debugging process by providing the vmlinux symbols in `/proc/kallsyms`.

15. Select **Exit** to exit the configuration screen.
16. Select **Yes** when asked if you want to save your configuration.
17. Execute the following command to rebuild the Linux kernel:

```
bitbake virtual/kernel
```

The uncompressed Linux kernel image with debug symbols, `vmlinux.elf`, is created.

NOTE

The location of the images directory might differ based on the BSP version being used. For the correct location of where the Linux kernel images are stored, see the SDK User Manual from `iso/help/documents/pdf`.

You just created a Linux kernel image that contains symbolic debugging information.

Now, you can use this image and create a CodeWarrior project for debugging the Linux kernel. The various use cases for the Linux kernel debug scenario are:

- CodeWarrior allows you to download this Linux kernel image (`vmlinux.elf`), RAM disk, and dtb files to the target.
- You can start the Linux kernel and RAM disk manually from U-Boot. The U-Boot, the kernel, RAM disk, and dtb images are written into flash memory.
- You can download the Linux kernel and RAM disk from CodeWarrior without using U-Boot.
- You can perform an early kernel debug before the MMU is enabled or debug after the Linux kernel boots and the login prompt is shown.

The Linux kernel debug scenarios are explained in the following sections:

- [Creating a CodeWarrior Project using the Linux Kernel Image](#) on page 289
- [Configuring the kernel project for debugging](#) on page 290
- [Debugging the kernel to download the kernel, RAM disk, and device tree](#) on page 301
- [Debugging the kernel based on MMU initialization](#) on page 302
- [Debugging the kernel by attaching to a running U-Boot](#) on page 305

7.7.5 Creating a CodeWarrior Project using the Linux Kernel Image

After creating a Linux kernel image with symbolic debugging information, you need to create a CodeWarrior project using the kernel image.

To create a CodeWarrior project:

1. Start the CodeWarrior IDE from the Windows system.
2. Select **File > Import**. The **Import** wizard appears.
3. Expand the **CodeWarrior** group and select **CodeWarrior Executable Importer**.
4. Click **Next**.

The **Import a CodeWarrior executable file** page appears.

5. Specify a name for the project, to be imported, in the **Project name** text box.
6. If you do not want to create your project in the default workspace:
 - a. Clear the **Use default location** checkbox.
 - b. Click **Browse** and select the desired location from the **Browse For Folder** dialog.
 - c. In the **Location** text box, append the location with the name of the directory in which you want to create your project.

NOTE

An existing directory cannot be specified for the project location.

7. Click **Next**.

The **Import C/C++/Assembler Executable Files** page appears.

8. Click **Browse** next to the **Executable** field.
9. Select the vmlinux file obtained.
10. Click **Open**.
11. From the **Processor** list, expand the processor family and select the required processor.
12. Select the **Bareboard Application** toolchain from the **Toolchain** group.

Selected toolchain sets up the default compiler, linker, and libraries used to build the new project. Each toolchain generates code targeted for a specific platform.
13. Select the **Linux Kernel** option from the **Target OS** list.

NOTE

Selecting Linux Kernel will automatically configure the initialization file for kernel download, the default translation settings (these settings need to be adjusted according to the actual Linux kernel configuration) in the OS Awareness tab, and the startup stop function to `start_kernel`.

14. Click **Next**.

The **Debug Target Settings** page appears.

15. From the **Debugger Connection Types** list, select the required connection type.
16. Specify the settings, such as board configuration, launch configuration, connection type, and TAP address if you are using Ethernet or Gigabit TAP.
17. Click **Next**.

The **Configurations** page appears.

18. From the **Core index** list, select the required core.

19. Click **Finish**.

The wizard creates a project according to your specifications.

You can access the project from the CodeWarrior Projects view on the workbench.

7.7.5.1 Updating the Linux Kernel Image

By modifying the Linux kernel image, you can update the project you just created.

You have built a new Linux kernel image file, `vmlinux.elf`, with some changes as compared to the current `vmlinux.elf` file being used in the CodeWarrior project you created. The following subsections present two scenarios to replace the current `vmlinux.elf` file with the new `vmlinux.elf` file:

- [Cache Symbolics Between Sessions is Enabled](#) on page 290
- [Cache Symbolics Between Sessions is Disabled](#) on page 290

7.7.5.1.1 Cache Symbolics Between Sessions is Enabled

This section provides steps to replace the current `vmlinux.elf` file with the new `vmlinux.elf` file when the cache symbolics between sessions is enabled.

Follow these steps:

1. Terminate the current debug session.
2. Right-click in the Debug window.
3. From the context menu, select **Purge Symbolics Cache**. The old `vmlinux.elf` file is being used by the debugger, but after you select this option, the debugger stops using this file in the disk.
4. Copy the new `vmlinux.elf` file over the old file.

Now, when you reinitiate a debug session, the updated `vmlinux.elf` file is used for the current debug session.

7.7.5.1.2 Cache Symbolics Between Sessions is Disabled

This section provides steps to replace the current `vmlinux.elf` file with the new `vmlinux.elf` file when the cache symbolics between sessions is disabled.

Follow these steps:

1. Terminate the current debug session.
2. Copy the new `vmlinux.elf` file over the old file.

Now, when you reinitiate a debug session, the updated `vmlinux.elf` file is used for the current debug session.

7.7.6 Configuring the kernel project for debugging

After you have created a CodeWarrior project using the Linux kernel image, the next action is to configure this project for debugging.

- [Configuring a download kernel debug scenario](#) on page 291
- [Configure an attach kernel debug scenario](#) on page 291
- [Setting up RAM disk](#) on page 294
- [Using Open Firmware Device Tree Initialization method](#) on page 297

7.7.6.1 Configuring a download kernel debug scenario

This section describes how to configure a download debug scenario.

For a download debug scenario, CodeWarrior:

- Resets the target
- Runs the initialization file
- Downloads the `.elf` file to the target; from the `vmlinux.elf` file, CodeWarrior writes the binary file to the target memory
- Sets the entry point based on the information available from the `.elf` file
- Runs the target

For a download debug scenario, to boot the Linux kernel, CodeWarrior requires the RAMDISK or ROOTFS file in addition to the `vmlinux.elf` file. This file is also built along with the image files when the kernel is compiled using the build tool. CodeWarrior also requires a DTB file that specifies the resources to be used by the kernel during its execution. For a download debug scenario, you need to configure the `vmlinux.elf` file, RAMDISK or ROOTFS file, and the DTB files to be downloaded into the target memory. All these files can be found in the specific target images folder.

NOTE

The location of the images directory might differ based on the BSP version being used. For the correct location of where the kernel images are stored, see the SDK User Manual in `iso/help/documents/pdf`.

These files are specified in the Download launch configuration after you have created the CodeWarrior project with the Linux kernel image. [Table 134. Kernel Project Download Launch Configuration Settings](#) on page 309 describes the settings you need to provide in the launch configuration.

7.7.6.2 Configure an attach kernel debug scenario

This section describes how to configure an attach debug scenario.

For the attach debug scenario, CodeWarrior does not download any file on the target. The kernel is started directly from U-Boot. You need to burn the U-Boot image to the flash memory of the hardware.

NOTE

See the *Burning U-Boot to Flash* cheat sheet for the entire procedure for burning U-Boot to flash. To access the cheat sheets, select **Help > Cheat Sheets** from the CodeWarrior IDE.

After the boot process, the U-Boot console is available and the Linux kernel can be started manually from U-Boot. For this, the following files can be either written into flash memory or can be copied from U-Boot using TFTP:

- Binary kernel image file, `uImage`
- Ramdisk to be started from U-Boot, for example,

```
<target version>.rootfs.ext2.gz.u-boot
```

- dtb file, for example, `uImage-<target version>.dtb`

After the Linux boot process, the Linux login appears and you can connect to debug the kernel using the CodeWarrior Attach launch configuration. As all the files are manually loaded from U-Boot, these files must not be specified in the launch configuration.

The table below describes the settings you need to provide in the launch configuration.

To specify the launch configuration settings in CodeWarrior:

1. Select **Run > Debug Configurations**.
2. Enter the launch configuration settings, given in the table below, in the **Debug Configurations** dialog.

Table 133: Kernel Project Attach Launch Configuration Settings

Debug Window Component	Settings
Main Tab	<p>Select an appropriate system (if existing) from the Connection drop-down list or define a new system.</p> <ul style="list-style-type: none"> • To define a new system, click New. • Select Hardware or Simulator Connection from the CodeWarrior Bareboard Debugging list. Click Next. • Specify a name and a description for the connection. • Select an appropriate target (if existing) from the Target drop-down list or define a new target. • To define a new target, click New on the Hardware or Simulator Connection dialog. • Select Hardware or Simulator Target from the CodeWarrior Bareboard Debugging list. Click Next. • Specify a name and a description for the target. • Select a target from the Target type drop-down list. On the Initialization tab, ensure there are no initialization files selected. • Click Finish to create the target and close the Hardware or Simulator Target dialog. • Select the type of connection you will use from the Connection type drop-down list. • Click Finish. • Select all the cores on which Linux is running (for example, core 0 for single-core or cores 0-7 for 8-core SMP).
Debugger Tab > Debugger options > Symbolics Tab	<p>Select the Cache Symbolics between sessions checkbox. The symbolics are loaded from the elf file to the debugger for the first session only. This shows a speed improvement for <code>vmlinux.elf</code> as the size is bigger than around 100 MB.</p>
Debugger Tab > Debugger options > OS Awareness Tab	<p>Select Linux from the Target OS drop-down list.</p>
Debugger Tab > Debugger options > OS Awareness Tab > Boot Parameters	<p>Disable all settings on the Boot Parameters tab.</p> <p style="text-align: center;">NOTE</p> <p style="text-align: center;">For details on the options available on the Boot Parameters tab, see Setting up RAM disk on page 294.</p>

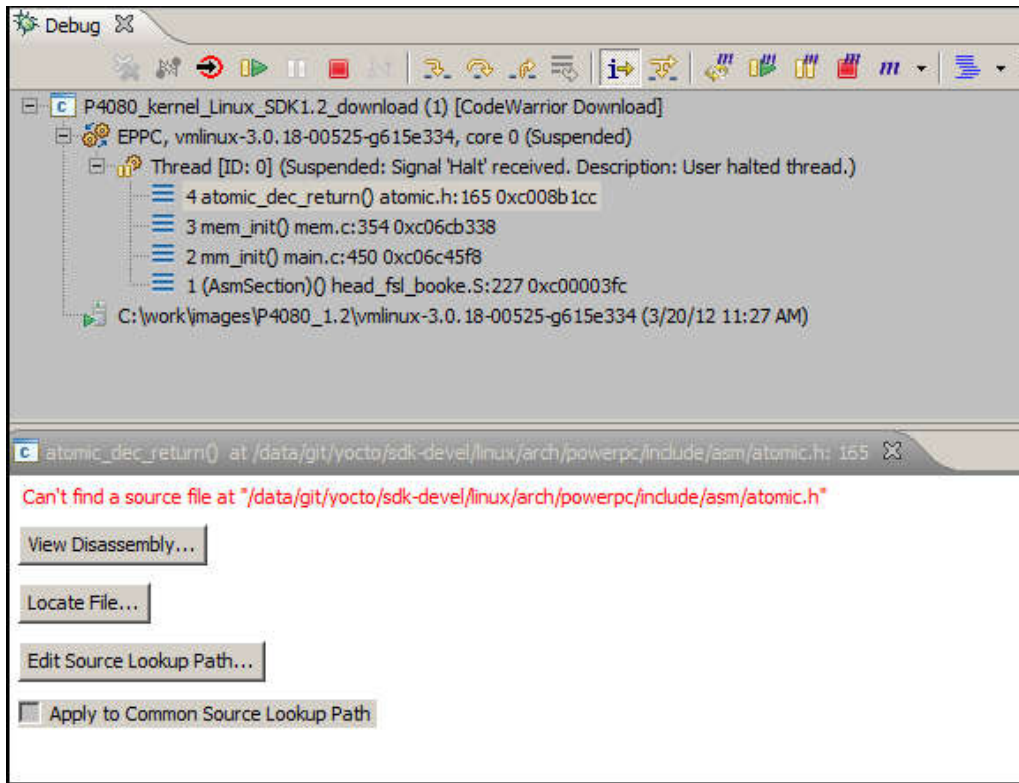
Table continues on the next page...

Table 133: Kernel Project Attach Launch Configuration Settings (continued)

Debug Window Component	Settings
Debugger Tab > Debugger options > OS Awareness Tab > Debug Tab	<p>Debug tab</p> <ul style="list-style-type: none"> Select the Enable Memory Translation checkbox <p>Physical Base Address is set to value CONFIG_KERNEL_START (0x0)</p> <p>Virtual Base Address is set to value CONFIG_KERNEL_START (0xc000 0000 for 32 bits, and 0xC000 0000 0000 0000 for 64bits).</p> <ul style="list-style-type: none"> Memory Size is the kernel space translation size. <p style="text-align: center;">NOTE</p> <p style="text-align: center;">The values shown above should be set as configured in the linux config file (.config). You can read the MMU registers to verify what you have configured and do a correction, if required.</p> <ul style="list-style-type: none"> Select Enable Threaded Debugging Support checkbox Select Enable Delayed Software Breakpoint Support If required, also select Update Background Threads on Stop. When enabled, the debugger reads the entire thread list when the target is suspended. This decreases the speed. If the option is disabled, the speed is increased but the Debug window might show non-existent threads, as the list is not refreshed.

3. Click the **Source** page to specify path mappings. Path mappings are not required if the debug host is similar to the compilation host. If the two hosts are separate, the .elf file contains the paths for the compilation host. Specifying the path mappings helps establish paths from compilation host to where the sources are available to be accessed by the debugger on the debugger host. If no path mapping is specified, when you perform a debug on the specified target, a source file missing message appears (shown in the figure below).

Figure 106: Debug View When No Path Mapping is Specified



You can specify the path mappings, either by adding a new path mapping on the **Source** tab or by clicking the appropriate buttons (**Locate File**, **Edit Source Lookup Path**) that appear when a source path mapping is not found.

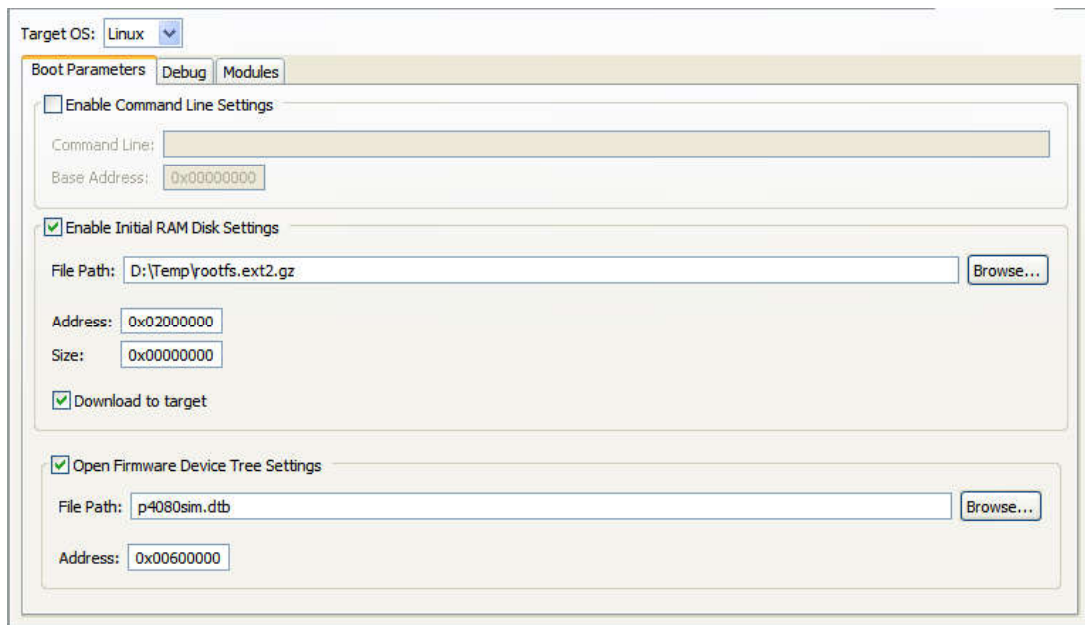
4. Click **Apply** to save the settings.
5. Click **Close**.

7.7.6.3 Setting up RAM disk

This section describes specifying RAM disk information that is used by the Linux kernel when it is booted.

You can specify RAM disk information in the **Boot Parameters** tab, which is present on the **OS Awareness** tab of the **Debugger** tab of the **Debug Configurations** dialog, as shown in the figure below. [Table 134. Kernel Project Download Launch Configuration Settings](#) on page 309 lists the instructions to set up the RAM disk.

Figure 107: Kernel Debug - OS Awareness Tab



Depending on the method you choose for passing parameters to the kernel during kernel initialization, the RAM disk information can be provided in any of the following ways:

- [Flattened Device Tree Initialization](#) on page 295
- [Regular Initialization](#) on page 296

7.7.6.3.1 Flattened Device Tree Initialization

In this method, the RAM disk is set up by specifying a device tree file that contains the initialization information.

To follow the Flattened device tree initialization method:

1. Open the **Debug Configurations** dialog.
2. Select the **Debugger** tab.
3. From the **Debugger options** panel, select the **OS Awareness** tab.
4. From the **Target OS** drop-down list, select **Linux**.
5. On the **Boot Parameters** tab, select the **Enable Initial RAM Disk Settings** checkbox.
The options in this group activate.
6. In the **File Path** field, type the path of the RAM disk.
Alternatively, click **Browse** to display a dialog that you can use to select this path.

NOTE

The RAM disk is created by the build tool and not by the kernel. It contains the initial file system. For details, see the SDK User Manual in `iso/help/documents/pdf`.

7. In the **Address** text box, enter `0x02000000`, or another appropriate base address where you want the RAM disk to be written.

NOTE

Ensure that the address you specify does not cause the RAM disk to overwrite the kernel. The kernel is loaded to 0x00000000. The address you specify should be greater than the size, in bytes, of the uncompressed Linux kernel with no debug symbols.

NOTE

If you use a DTB file, ensure to use the same addresses for RAM disk and initial RAM disk (`initrd`) start value from the chosen section. The kernel must find the RAM disk at the address specified in the `.dtb` file.

8. In the **Size** text box, enter the size of the RAM disk file. To copy all the contents of the RAM disk file, enter zero (0).
9. Select the **Download to target** checkbox to download the RAM disk file to the target board.
The debugger copies the initial RAM disk to the target board only if this checkbox is checked.

NOTE

Most embedded development boards do not just use a small initial RAM disk, but a large root file system. The Download to target option works in both the cases, but for large file systems it is better to deploy the file directly to the target in the flash memory and not have it downloaded by the debugger.

7.7.6.3.2 Regular Initialization

In this method, the RAM disk is set up by passing the parameters through the command-line settings using the **Boot Parameters** tab.

To follow the regular initialization method:

1. Open the **Debug Configurations** dialog.
2. Select the **Debugger** tab.
3. From the **Debugger options** panel, select the **OS Awareness** tab.
4. From the **Target OS** drop-down list, select **Linux**.
5. On the **Boot Parameters** tab, select the **Enable Command Line Settings** checkbox.
The options in this group activate.
6. Specify the RAM disk parameters for use in the Command Line field. For example:

- You can specify the following when the regular initialization of the kernel is used:

```
root=/dev/ram rw"
```

- Sample NFS parameters:

```
"root=/dev/nfs ip=10.171.77.26  
nfsaddr=10.171.77.26:10.171.77.21  
nfsroot=/tftpboot/10.171.77.26"  
"root=/dev/nfs rw  
nfsroot=10.171.77.21:/tftpboot/10.171.77.26  
ip=10.171.77.26:10.171.77.21:10.171.77.254:255.255.255.0:8280x:eth0:off"
```

where, 10.171.77.21 is the IP address of the NFS server and 10.171.77.26 is the IP address of the target platform.

"/tftpboot/10.171.77.26" is a directory on the host computer where the target platform file system is located.

"8280x" is the host name.

- Sample flash parameters: `root=/dev/mtdblock0` or `root=/dev/mtdblock2`
(depending on your configuration)

7.7.6.4 Using Open Firmware Device Tree Initialization method

You can use the Open Firmware Device Tree Initialization method as an alternate way of loading parameters to the kernel from a bootloader on Power Architecture processors.

Since downloading the kernel with the CodeWarrior IDE emulates bootloader behavior, the IDE provides this way of passing the parameters to the kernel.

The Open Firmware Device Tree initialization method involves the following general actions:

1. [Obtain a DTS file](#) on page 297
2. [Edit DTS file](#) on page 299
3. [Compile DTS file](#) on page 300
4. [Test DTB file](#) on page 300
5. [Modify a DTS file](#) on page 300

7.7.6.4.1 Obtain a DTS file

A device tree settings (.dts) file is a text file that contains the kernel setup information and parameters.

To obtain a device tree source file that can be used with CodeWarrior:

1. Configure a TFTP server on a Linux PC.
2. Copy the Linux images on the TFTP server PC in the specific directory. The following files are needed:
 - `ulimage`
 - `rootfs.ex2.gz.uboot` (if this is not present, check if the **Target Image Generation > Create a ramdisk that can be used by u-boot** option is enabled.
 - A device tree blob (DTB) obtained from the kernel sources. To convert this into a DTB, use the Device Tree Compiler (DTC) that is available in the BSP:

```
dtc -f -b 0 -S 0x3000 -R 8 -I dtb -O dts <target>.dtb > <target>.dts
```

NOTE

Standard DTS files are available along with Linux kernel source files in `<SDK_Linux_sources_root>/arch/powerpc/boot/dts`. For the exact location of where the kernel images are stored, see the SDK User Manual from `iso/help/documents/pdf`.

3. Power on the target. Wait until the uboot prompt is displayed.
4. Ensure that networking is working on the target. You need to have a network cable plugged in and set several variables (`ipaddr`, `netmask`, `serverip`, `gatewayip`), including the IP address of the TFTP server. For example,

```
ipaddr=10.171.77.230  
netmask=255.255.255.0
```

```
serverip=10.171.77.192  
gatewayip=192.168.1.1
```

5. Check that network connectivity is working by pinging the TFTP server.

```
ping $serverip
```

6. On the uboot prompt, download the DTS and configure it for the current target. For example,

```
tftp 3000000 /tftpboot/<target>.dtb  
fdt addr 0x3000000  
fdt boardsetup  
fdt print
```

7. Copy the output of this command as a DTS file.
8. Modify the memreserve statement at the beginning of the DTS file. The first parameter is the start address of the memory reserved for the RAM disk. The second parameter is the size of the RAM disk and must be modified each time the RAM disk is repackaged as you might add additional packages to the RAM disk. For example,

```
/memreserve/ 0x20000000 0x453ecc;
```

9. Modify the chosen node in the DTS file. The linux,initrd-start argument must be the start address of the RAM disk, and the linux,initrd-end value must be the end address of the RAM disk. For example,

```
chosen {  
    linux,initrd-start = <0x2000000>;  
    linux,initrd-end = <0x2453ecc>;  
    linux,stdout-path = "/soc8572@ffe00000/serial@4500";  
};
```

10. Ensure that the frequencies of the target are correct. If the DTS was generated in U-Boot as described above, the frequencies should be correct. However, if you update an existing DTS file for a new board revision, the frequencies might have changed and they need to be corrected in the DTS file.

- a. At the U-Boot prompt, inspect the current configuration.

```
bdinfo  
...  
    intfreq = 1500 MHz  
    busfreq = 600 MHz  
...
```

- b. The intfreq value from the U-Boot output must be converted to a hexadecimal value and added to the clock-frequency value of the CPU node in the DTS file. The busfreq value must be placed in the same way in the bus-frequency parameter. For example,

```
cpus {  
    PowerPC,<target>@0 {  
        ...  
        timebase-frequency = <0x47865d2>;  
        bus-frequency = <0x23c34600>;  
        clock-frequency = <0x5967f477>;  
    };  
};
```


- c. The same busfreq value is the clock frequency for the serial ports and must be updated in the DTS file also:

```
serial0: serial@4500 {
    ...
    clock-frequency = <0x23c34600>;
};
```

NOTE

If you are using hardware for kernel debugging, see [Edit DTS file](#) on page 299.

7.7.6.4.2 Edit DTS file

You need to edit the settings (.dts) file with information relevant to the current target board and kernel.

If you have a DTS file specifically designed for your target board, you should modify only the RAM disk end address and reserved memory area, in case you are using a RAM disk.

A standard .dts text file has a number of nodes which are given no value (actually <0>) or are missing nodes (for example, the /chosen branch).

When the Linux kernel is started from U-Boot with bootm, U-Boot dynamically edits the .dtb file in RAM so as to fill in the missing values and add the /chosen branch, based on the U-Boot environment variables.

The CodeWarrior IDE does not fill in the missing values and branches when it downloads the .dtb file to RAM. You must manually create and compile a separate and complete .dts file.

The following steps detail the changes that must be applied to the .dts file so the kernel boots successfully when the CodeWarrior IDE loads the .dtb file into RAM with a Linux kernel and a initial RAM disk.

1. Update the bus-frequency and clock-frequency nodes from the value KRd=>bi_busfreq
2. Update the clock-frequency nodes from the value KRd=>bi_initfreq;
3. Update the following nodes from the value KRd=>bi_tbfreq:


```
/cpus/ PowerPC,8349@0/timebase-frequency
```
4. Create the following node from the size on disk of the file entered in LKBP=>Enable Initial RAM Disk=>File Path or from the address entered in LKBP=>Enable Initial RAM Disk=>Address:


```
/memreserve/
```
5. Create the following node from LKBP=>Command Line:


```
/chosen/bootargs
```
6. Create the node:


```
linux,stdout-path
```
7. Create the following node from the address entered in LKBP=>Enable Initial RAM Disk=>Address:


```
/chosen/linux,initrd-start
```
8. Create the following node from the size on disk of the file entered in LKBP=>Enable Initial RAM Disk=>File Path and from the address entered in LKBP=>Enable Initial RAM Disk=>Address:


```
/chosen/linux,initrd-end
```

7.7.6.4.3 Compile DTS file

You can compile the settings (.dts) file to a binary (.dtb) file, if you need the binary file to set up the kernel parameters for the board.

1. Ensure that you have the DTC device tree compiler on your host machine.

If the DTC device tree compiler is missing, get the latest DTC source archive from bitshrine.org. Extract the archive, run make, and put the binary somewhere reachable by your PATH.

```
wget dtc-20070307.tar.bz2
wget dtc-20070307.tar.bz2.md5
wget dtc-20070307.tar.gz
wget dtc-20070307.tar.gz.md5
```

2. Navigate to the folder containing DTS files.

NOTE

The location of the DTS file might differ based on the BSP version being used. For the correct location of the file, see the SDK User Manual in [iso/help/documents/pdf](#).

3. Compile the .dts device tree source file for the board:

```
$ cd arch/powerpc/boot/dts
$ dtc -I dts -O dtb -V 0x10 -b 0 <target>.dts > <target>.dtb
```

NOTE

You can use the created binary (.dtb) file in the CodeWarrior IDE (in the **Boot Parameters** tab); see [Configure an attach kernel debug scenario](#) on page 291 for details.

7.7.6.4.4 Test DTB file

You can test the binary (.dtb) file outside the CodeWarrior IDE.

The steps are as follows:

1. Load the uImage, rootfs.ext2.gz.uboot, and <target>.dtb file onto the board.
2. Boot the board and verify that Linux comes up fine.

```
$ bootm <kerneladdress> <ramdiskaddress> <dtbaddress>
```

NOTE

The target board must have U-Boot present in the flash at the reset address so that U-Boot can run and set board configurations.

7.7.6.4.5 Modify a DTS file

You may need to modify a DTS file if you are using a BSP version that is not supported by a CodeWarrior DTS file or custom board.

Follow these steps to modify the DTS file:

1. Obtain a DTS file.

NOTE

The location of the DTS file might differ based on the BSP version being used. For the correct location of the file, see the SDK User Manual in [iso/help/documents/pdf](#).

2. Modify this DTS file with the information provided by U-Boot. To do this:
 - a. Check the `/proc/device-tree/` directory for the required information after kernel boot from U-Boot.
Alternatively, you may:
 - b. Enable `ft_dump_blob` call from the `u-boot/common/cmd_bootm.c` file. By default this is disabled.
 - c. Build the U-Boot and write it on the target to have this enabled when booting the kernel.
 - d. After this, configure U-Boot as described in the BSP documentation to boot the kernel and save the boot log.
 - e. Check the device tree displayed during kernel boot and accordingly modify your DTS file.

7.7.7 Debugging the kernel to download the kernel, RAM disk, and device tree

This section describes how to debug the Linux kernel using CodeWarrior IDE to download the kernel, RAM disk, and device tree.

Perform the following steps:

1. Create a project for the Linux kernel image. See [Creating a CodeWarrior Project using the Linux Kernel Image](#) on page 289.
2. Configure the launch configuration for Linux kernel debug.
 - a. Select **Run > Debug Configurations**.
The **Debug Configurations** dialog appears.
 - b. From the left pane, in the CodeWarrior group, select the appropriate launch configuration.
 - c. On the **Main** page, in the **Connection** panel, select the appropriate system from the **Connection** drop-down list.
 - d. Click **Edit**.
The **Properties for <connection>** window appears.
 - e. Click **Edit** next to the **Target** drop-down list.
The **Properties for <Target>** dialog appears.
 - f. On the **Initialization** tab, select the checkboxes for all the cores in the **Run out of reset** column.
 - g. In the **Initialize target** column, select the checkbox for core 0.
 - h. Click the ellipses button in the **Initialize target script** column.
The **Target Initialization** dialog appears.
 - i. Click **File System** and select the target initialization file from the following path:

```
<CWInstallDir>\PA\PA_Support\Initialization_Files\<<Processor Family>  
\<target>_uboot_init_Linux.tcl
```

NOTE

The initialization file is automatically set when you select **Linux Kernel** as the **Target OS**, while creating a new Power Architecture project using the CodeWarrior Bareboard Project Wizard.

- j. Click **OK** to close the **Memory Configuration File** dialog.
- k. Click **OK** to close the **Properties for <Target>** dialog.

- l. Click **OK** to close the **Properties for <connection>** dialog.
- m. On the **Debug** tab of the **Debugger** tab, select an **Program execution** option, to stop the debug process at the program entry point or at a specified user function or address like `start_kernel`.
- n. On the **OS Awareness** tab of the **Debugger** tab, select **Linux** from the **Target OS** drop-down list.
- o. On the **Boot Parameters** tab of the **OS Awareness** tab:
 - i. Select the **Enable Initial RAM Disk Settings** checkbox.
The fields in that panel are enabled.
 - ii. In the **File Path** text box, enter the location of the BSP file, `rootfs.ext2.gz`.
 - iii. In the **Address** text box, enter the address where you want to add the RAM disk.
 - iv. In the **Size** text box enter 0 if you want the entire RAM disk to be downloaded.
 - v. Select the **Open Firmware Device Tree Settings** checkbox.
 - vi. In the **File Path** text box, enter the location of the device tree file.
 - vii. In the **Address** text box, enter the location in memory where you want to place the device tree.

NOTE

Ensure that the memory areas for kernel, RAM disk, and device tree do not overlap.

- p. Click **Apply** to save the settings you made to the launch configuration.

3. Click **Debug** to start debugging the kernel.

NOTE

If the kernel does not boot correctly, check the values entered in the **Boot Parameters** tab. Also ensure that you provided a valid device tree and RAM disk.

7.7.8 Debugging the kernel based on MMU initialization

This section describes how to debug the Linux kernel based on whether the MMU is disabled, being enabled, or enabled.

NOTE

You can debug the kernel on all stages from `0x0` till `start_kernel` and further, without the need of PIC changes, breakpoints at `start_kernel`, and multiple debug sessions.

Debugging the Linux kernel involves three stages with different views and functionality:

- [Debugging the Kernel before the MMU is Enabled](#) on page 302
- [Debugging the Kernel while the MMU is being Enabled](#) on page 304
- [Debugging the Kernel after the MMU is Enabled](#) on page 304

7.7.8.1 Debugging the Kernel before the MMU is Enabled

This procedure shows how to debug the kernel before the memory management unit (MMU) is initialized.

You can always debug assembly before virtual addresses are being used, without setting the alternate load address.

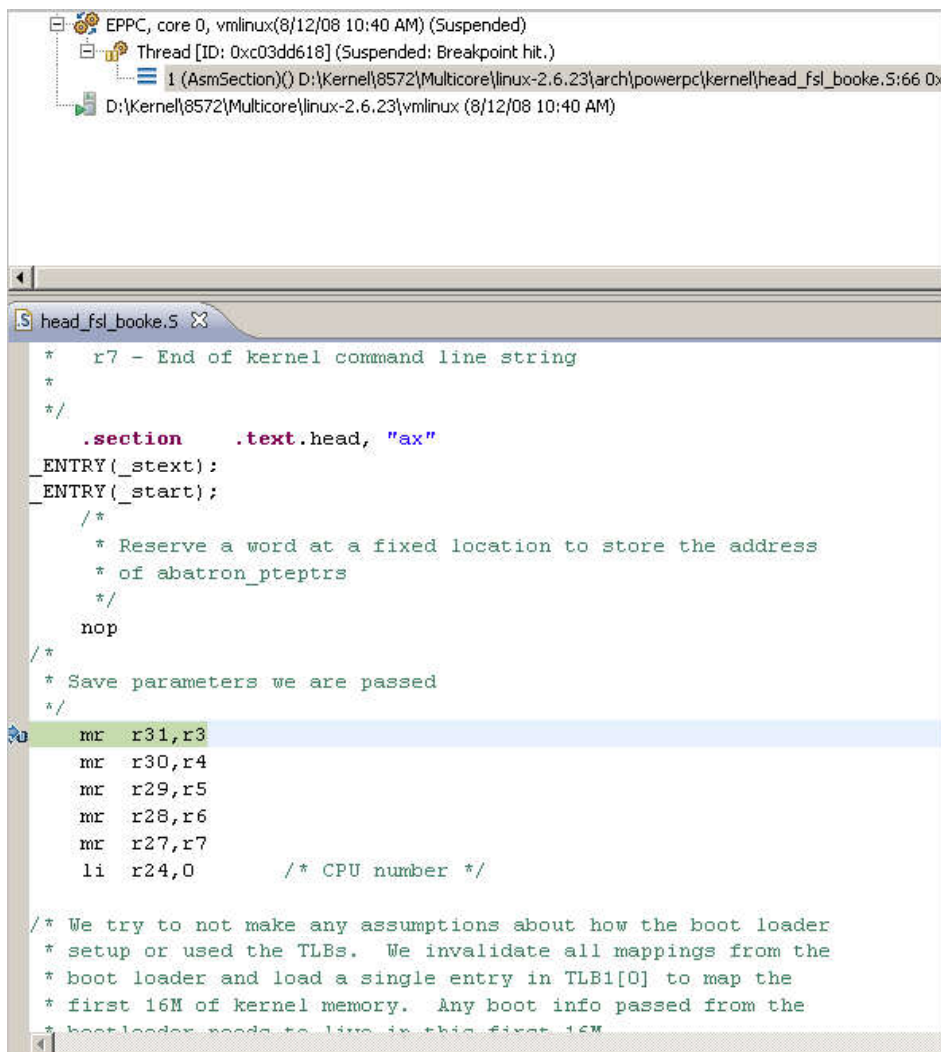
To debug the kernel before the MMU is enabled, follow these steps:

1. Select **Run > Debug Configurations** from the CodeWarrior menu bar to open the **Debug** Configurations dialog.
2. From the **Debugger** page, select the **PIC** tab.

3. Select the **Alternate Load Address** checkbox.
4. In the **Alternate Load Address** field, type the hexadecimal form of the memory address (for example, 0x00000000).
5. Click **Apply**. The CodeWarrior IDE saves your changes to the launch configuration.
6. Click **Debug**. The **Debug** perspective appears.
7. Set a breakpoint early in `head_fsl_booke.S`.

You can perform source level debug until the `rfi` instruction in `head_fsl_booke.S`.

Figure 108: Kernel Debug - Before MMU is Enabled



NOTE

You must stop the debug session and clear the **Alternate Load Address** checkbox in the **PIC** tab to debug after the `rfi` instruction in `head_fsl_booke.S`.

7.7.8.2 Debugging the Kernel while the MMU is being Enabled

This procedure shows how to debug the kernel while the memory management unit is being initialized.

To debug this section of code, ensure that the **Alternate Load Address** checkbox in the **PIC** tab is disabled.

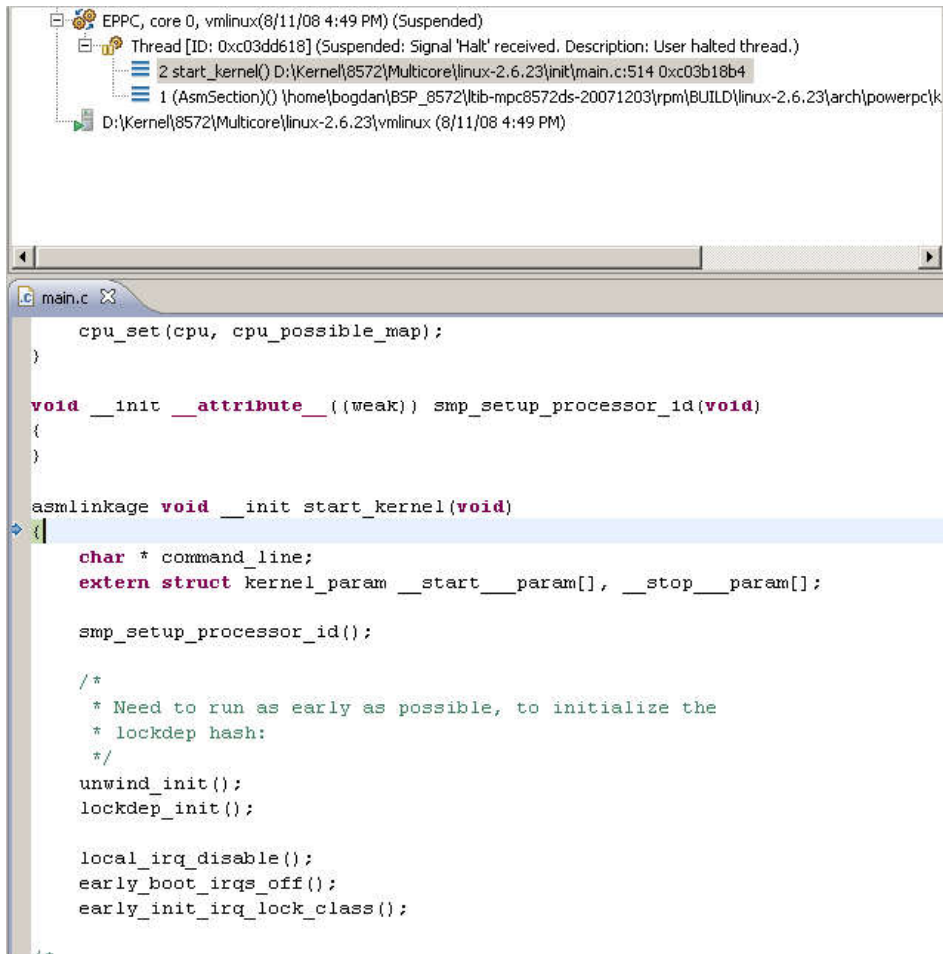
7.7.8.3 Debugging the Kernel after the MMU is Enabled

This procedure shows how to debug the kernel after the memory management unit is initialized.

To debug the kernel after the MMU is enabled, follow these steps:

1. Select **Run > Debug Configurations** from the CodeWarrior menu bar to open the **DebugConfigurations** dialog.
2. From the **Debugger** tab, select the **PIC** tab.
3. Clear the **Alternate Load Address** checkbox.
4. Click **Apply**.
5. Click **Debug** to start the debug session. The **Debug** perspective appears.
6. In the editor area, set a breakpoint at `start_kernel`, after the eventpoint, in `main.c`. This will stop the debug session at `start_kernel` function (shown in the figure below).

Figure 109: Kernel Debug - After MMU is Enabled



7. Click **Run**.

The debugger halts execution of the program at whatever breakpoints have been set in the project (if any breakpoints have been set).

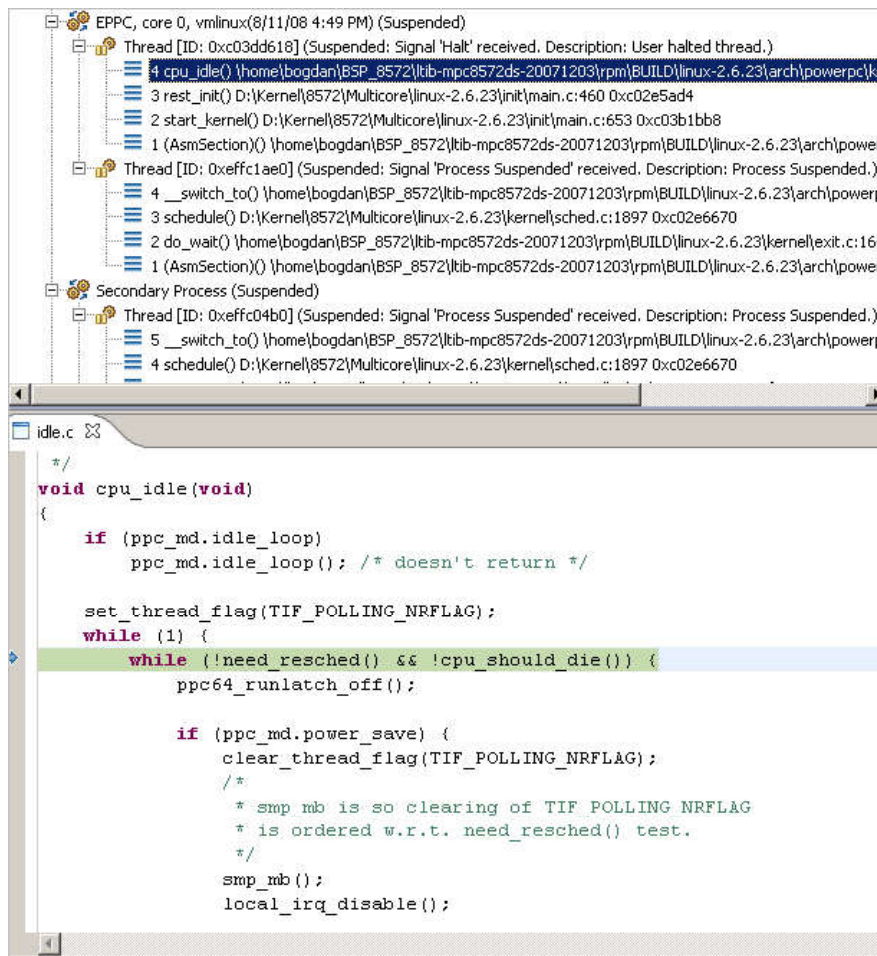
8. Run through the rest of the code until the kernel starts to boot.

When the kernel boots, boot status messages appear in the simulator window.

NOTE

You can click Terminate to halt running of the kernel and set breakpoint/watchpoints in the debug window, as shown in the figure below.

Figure 110: Kernel Stopped by User



9. Continue debugging.

10. When finished, you can either:

- a. Kill the process by selecting **Run > Terminate**.
- b. Leave the kernel running on the hardware.

7.7.9 Debugging the kernel by attaching to a running U-Boot

This section explains how to debug the Linux kernel by attaching it to a running U-Boot.

To debug the kernel by attaching to a running U-Boot, perform the following:

1. Create a project for the Linux kernel image. For more details, see [Creating a CodeWarrior Project using the Linux Kernel Image](#) on page 289.
2. Configure the launch configuration for Linux kernel debug. For more details, see [Configure an attach kernel debug scenario](#) on page 291.
3. Select **Run > Debug Configurations**. The **Debug Configurations** dialog appears.
4. From the left pane, expand the **CodeWarrior Attach** tree and select the appropriate launch configuration.
5. From the **Debugger** tab, select the **PIC** tab.
6. Clear the **Alternate Load Address** checkbox.
7. Click **Apply**.
8. Click **Debug** to start the debug session. The **Debug** perspective appears.
9. While the U-Boot is running, attach the target.

The debugger displays a warning, in the console, as the kernel is not being executed on the target.

NOTE

For multi-core processors, only `core0` is targeted in the **Debug** view. This is normal as the secondary cores are initialized in the Linux kernel after MMU initialization. CodeWarrior will automatically add other cores, in the **Debug** view, after the kernel initializes the secondary cores.

10. Set software or hardware breakpoints for any stage (before or after MMU initialization).

To set a software breakpoint for the entry point address (for example, `address 0x0`), issue the following command in the **Debugger Shell** view.

```
bp 0x0
```

11. Using the U-boot console, load the Linux kernel, DTB file, and RAM disk/rootfs from flash or from TFTP.
12. Debug the kernel.

The debugger halts execution of the program at whatever breakpoints have been set in the project. Typical stages involved in debugging the kernel are discussed below:

a. Debugging the kernel at the entry point

The CodeWarrior debugger will stop at the kernel entry point, if any software or hardware breakpoint has been set for entry point.

NOTE

For the debugger to stop at the kernel entry point, set a breakpoint before loading the kernel from the U-boot console.

At the entry point, the MMU is not initialized and therefore debugging before MMU initialization also applies in this stage.

b. Debugging the Kernel before the MMU is enabled

Being in early debug stage, the user should set the correct PIC value, to see the source correspondence, by issuing the `setpicloadaddr 0x0` command in the **Debugger Shell** view.

Before setting a breakpoint for the stage after MMU initialization (for example, breakpoint at `start_kernel`) the correct PIC should be set, by issuing the `setpicloadaddr reset` command in the **Debugger Shell** view. This is required to ensure that the new breakpoint is set with the correct PIC for the stage after MMU initialization.

The user can set breakpoints and run/step to navigate, before MMU initialization. The correct PIC should be set by issuing the `setpicloadaddr reset` command in the **Debugger Shell** view, before the debuggers enters the next stage.

c. Debugging the Kernel after the MMU is enabled

After the MMU is initialized, the PIC value must be reset y issuing the `setpicloadaddr reset` command in the **Debugger Shell** view. During the Linux Kernel booting, you can debug this stage directly, if no breakpoint has been set for the stage before MMU initialization. Alternatively, you can also debug this stage after run or step from the stage before initialization.

NOTE

In case of SMP, all the secondary cores are targeted and displayed in the **Debug** view.

13. When finished, you can either:

- a. Kill the process by selecting **Run > Terminate**.
- b. Leave the kernel running on the hardware.

7.8 Debugging Loadable Kernel Modules

This section explains how to use the CodeWarrior debugger to debug a loadable kernel module.

This section contains the following subsections:

- [Loadable Kernel Modules - An Introduction](#) on page 307
- [Creating a CodeWarrior Project from the Linux Kernel Image](#) on page 308
- [Configuring Symbolics Mappings of Modules](#) on page 310

7.8.1 Loadable Kernel Modules - An Introduction

The Linux kernel is a *monolithic kernel*, that is, it is a single, large program in which all the functional components of the kernel have access to all of its internal data structures and routines.

Alternatively, you may have a micro kernel structure where the functional components of the kernel are broken into pieces with a set communication mechanism between them. This makes adding new components to the kernel using the configuration process very difficult and time consuming. A more reliable and robust way to extend the kernel is to dynamically load and unload the components of the operating system using Linux *loadable kernel modules*.

A *loadable kernel module* is a binary file that you can dynamically link to the Linux kernel. You can also unlink and remove a loadable kernel module from the kernel when you no longer need it. Loadable kernel modules are used for device drivers or pseudo-device drivers, such as network drivers and file systems.

When a kernel module is loaded, it becomes a part of the kernel and has the same rights and responsibilities as regular kernel code.

Debugging a loadable kernel module consists of several general actions, performed in the following order:

1. Create a CodeWarrior Linux kernel project for the loadable kernel module to be debugged. See [Creating a CodeWarrior Project from the Linux Kernel Image](#) on page 308
2. Add the modules and configure their symbolics mapping. See [Configuring Symbolics Mappings of Modules](#) on page 310

7.8.2 Creating a CodeWarrior Project from the Linux Kernel Image

The steps in this section show how to create a CodeWarrior project from a Linux kernel image that contains symbolic debugging information.

NOTE

The following procedure assumes that you have made an archive of the Linux kernel image and transferred it to the Windows machine. For kernel modules debugging, ensure that you build the kernel with loadable module support and also make an archive for the `rootfs` directory, which contains the modules for transferring to Windows.

1. Launch CodeWarrior IDE.
2. Select **File > Import**. The Import wizard appears.
3. Expand the **CodeWarrior** group and select **CodeWarrior Executable Importer**.
4. Click **Next**.

The **Import a CodeWarrior Executable file** page appears.

5. Specify a name for the project, to be imported, in the **Project name** text box.
6. If you do not want to create your project in the default workspace:
 - a. Clear the **Use default location** checkbox.
 - b. Click **Browse** and select the desired location from the **Browse For Folder** dialog.
 - c. In the **Location** text box, append the location with the name of the directory in which you want to create your project.

NOTE

An existing directory cannot be specified for the project location.

7. Click **Next**.

The **Import C/C++/Assembler Executable Files** page appears.
8. Click **Browse** next to the **Executable** field.
9. Select the `vmlinux.elf` file.
10. Click **Open**.
11. From the **Processor** list, expand the processor family and select the required processor.
12. Select **Bareboard Application** from the **Toolchain** group.
13. Select **Linux Kernel** from the **Target OS** list.
14. Click **Next**.

The **Debug Target Settings** page appears.

15. From the **Debugger Connection Types** list, select the required connection type.
16. Specify the settings, such as board, launch configuration, connection type, and TAP address if you are using Ethernet or Gigabit TAP.
17. Click **Next**.

The **Configuration** page appears.

18. From the **Core index** list, select the required core.
19. Click **Finish**.

The wizard creates a project according to your specifications. You can access the project from the CodeWarrior Projects view on the Workbench.

20. Configure the launch configuration for linux kernel debug.

a. Select **Run > Debug Configurations**.

The Debug Configurations dialog appears.

21. Enter the launch configuration settings in the **Debug Configurations** dialog. The table below lists the launch configuration settings.

Table 134: Kernel Project Download Launch Configuration Settings

Debug Window Component	Settings
Main Tab	Select an appropriate system (if existing) from the Connection drop-down list or define a new system. <ul style="list-style-type: none"> • To define a new system, click New. • Select Hardware or Simulator Connection from the CodeWarrior Bareboard Debugging list. Click Next. • Specify a name and a description for the connection. • Select an appropriate target (if existing) from the Target drop-down list or define a new target. • To define a new target, click New on the Hardware or Simulator Connection dialog. • Select Hardware or Simulator Target from the CodeWarrior Bareboard Debugging list. Click Next. • Specify a name and a description for the target. • Select a processor from the Target type drop-down list. On the Initialization tab, ensure that there are no initialization files selected. • Click Finish to create the target and close the Hardware or Simulator Target dialog. • Select the type of connection you will use from the Connection type drop-down list. • Click Finish. • Select all the cores on which Linux is running (for example, core 0 for single-core or cores 0-7 for 8-core SMP).
Debugger Tab > Debugger options > Symbolics Tab	Select the Cache Symbolics between sessions checkbox. The symbolics are loaded from the elf file to the debugger for the first session only. This shows a speed improvement for <code>vmlinux.elf</code> as the size is bigger than around 100 MB.
Debugger Tab > Debugger options > OS Awareness Tab	Select Linux from the Target OS drop-down list.

Table continues on the next page...

Table 134: Kernel Project Download Launch Configuration Settings (continued)

Debug Window Component	Settings
Debugger Tab > Debugger options > OS Awareness Tab > Boot Parameters Tab	Select the Enable Initial RAM Disk Settings checkbox <ul style="list-style-type: none"> • File Path: Path of the RAM disk that you transferred from the Linux machine • Address: The address specified in Linux, initrd-start from the dts file Select the Download to target checkbox Select the Open Firmware Device Tree Settings checkbox <ul style="list-style-type: none"> • File Path: Path to the <target>.dtb file • Address: 0x00600000
Debugger Tab > Debugger options > OS Awareness Tab > Debug Tab	<ul style="list-style-type: none"> • Select the Enable Memory Translation checkbox <p>Physical Base Address is set to value CONFIG_KERNEL_START (0x0)</p> <p>Virtual Base Address is set to value CONFIG_KERNEL_START (0xc000 0000 for 32 bits, and 0xC000 0000 0000 0000 for 64bits).</p> <ul style="list-style-type: none"> • Memory Size is the kernel space translation size. <p style="text-align: center;">NOTE</p> <p style="text-align: center;">The values shown above should be set as configured in the linux config file (.config). You can read the MMU registers to verify what you have configured and do a correction, if required.</p> <p>Select the Enable Threaded Debugging Support checkbox</p> <p>Select the Enable Delayed Software Breakpoint Support checkbox</p>
Debugger Tab > Debugger options > OS Awareness Tab > Modules Tab	<ul style="list-style-type: none"> • Select the Detect module loading checkbox • Click Add to insert the kernel module file. See Configuring Symbolics Mappings of Modules on page 310 • Select the Prompt for symbolics path if not found checkbox

22.Click the **Source** page to add source mappings for `rootfs` and `linux-<version>`.

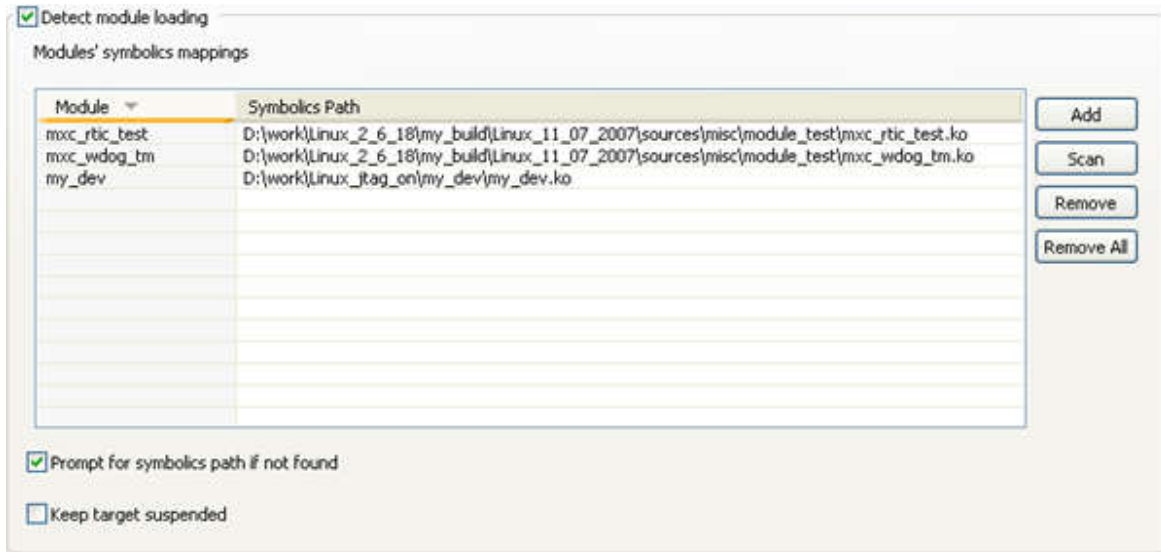
23.Click **Apply** to save the settings.

7.8.3 Configuring Symbolics Mappings of Modules

You can add modules to the Linux kernel project and configure the symbolics mappings of the modules using the **Modules** tab of the **Debug Configurations** dialog.

The figure below shows the **Modules** tab of the **Debug Configurations** dialog.

Figure 111: Kernel Module Debug - Modules Tab



The table below describes the various options available on the **Modules** tab.

Table 135: Kernel Module Project Launch Configuration - Modules Tab Settings

Option	Description
Detect module loading	Enables the debugger to detect module load events and insert an eventpoint in the kernel. Disabling this setting delays the module loading. This is useful in scenarios where multiple modules are loaded to the kernel and not all of them need to be debugged. You can enable this setting again in the Modules dialog. The dialog is available during the Debug session from the System Browser View toolbar > Module tab.
Add	Adds a module name along with the corresponding symbolic path This option displays a dialog in the following scenarios: <ul style="list-style-type: none"> • The file that you have selected is not a valid compiled kernel module • If the selected module already exists in the list with the same path
Scan	Automatically searches for module files and populates the kernel module list.
Remove	Removes the selected items. This button will be enabled only if a row is selected.
Remove All	Removes all items. This button will be enabled only if the kernel list contains any entries.

Table continues on the next page...

Table 135: Kernel Module Project Launch Configuration - Modules Tab Settings (continued)

Option	Description
Prompt for symbolics path if not found	Prompts to locate the symbolics file if a mapping for it is not available in the settings A Browse dialog appears that allows you to browse for a module file containing symbolics. The debugger will add the specified symbolics to the modules' symbolics mapping.
Keep target suspended	Keeps the target suspended after the debugger loads the symbolics file for a module. This option is useful if you want to debug the module's initialization code. It allows you to set breakpoints in the module's initialization code before running it. <p style="text-align: center;">NOTE</p> This option is automatically enabled when activating the Prompt for symbolics path if not found option.

NOTE

Breakpoints are resolved each time a symbolics file is loaded and the debugger uses the modules unload events for symbolics disposal and breakpoints cleanup.

7.9 Debugging Hypervisor Guest Applications

This section shows you how to debug hypervisor guest applications.

This section explains:

- [Hypervisor - An Introduction](#) on page 312
- [Prerequisites for Debugging a Guest Application](#) on page 313
- [Adding CodeWarrior HyperTRK Debug Stub Support in Hypervisor for Linux Kernel Debugging](#) on page 313
- [Preparing Connection to P4080DS Target](#) on page 314
- [Debugging AMP/SMP Guest Linux Kernels Running Under Hypervisor](#) on page 315
- [Debugging Hypervisor During the Boot and Initialization Process](#) on page 322

7.9.1 Hypervisor - An Introduction

The embedded hypervisor is a layer of software that enables the efficient and secure partitioning of a multi-core system.

A system's CPUs, memory, and I/O devices can be divided into groupings or partitions. Each partition is capable of executing a guest operating system.

Key features of the hypervisor software architecture are summarized below-

- Partitioning: Support for partitioning of CPUs, memory, and I/O devices:
 - CPUs: Each partition is assigned one or more CPU cores in the system.

- **Memory:** Each partition has a private memory region that is only accessible to the partition that is assigned the memory. In addition, shared memory regions can be created and shared among multiple partitions.
- **I/O devices:** P4080 I/O devices may be assigned directly to a partition (Direct I/O), making the device a private resource of the partition, and providing optimal performance.
- **Protection and Isolation:** The hypervisor provides complete isolation of partitions, so that one partition cannot access the private resources of another. The P4080 PAMU (an iommu) is used by Topaz to ensure device-to-memory accesses are constrained to allowed memory regions only.
- **Sharing:** Mechanisms are provided to selectively enable partitions to share certain hardware resources (such as memory)
- **Virtualization:** Support for mechanisms that enable the sharing of certain devices among partitions such as the system interrupt controller
- **Performance:** The hypervisor software uses the features of the Freescale Embedded Hypervisor APU to provide security and isolation with very low overhead. Guest operating systems take external interrupts directly without hypervisor involvement providing very low interrupt latency.
- **Ease of migration:** The hypervisor uses a combination full emulation and para-virtualization to maintain high performance and requiring minimal guest OS changes when migrating code from an e500mc CPU to the hypervisor.

7.9.2 Prerequisites for Debugging a Guest Application

The P4080 software bundle is the prerequisite for debugging a hypervisor guest application using the CodeWarrior IDE.

The software bundle used in the current example is *P4080 Beta 2.0.2 SW Bundle*.

7.9.3 Adding CodeWarrior HyperTRK Debug Stub Support in Hypervisor for Linux Kernel Debugging

This section explains how to add CodeWarrior HyperTRK debug stub support in the hypervisor for guest LWE or Linux kernel debugging.

To add CodeWarrior HyperTRK debug stub support:

1. Download the appropriate P4080 software bundle image (the BSP in `.iso` format) to a Linux computer.
2. Mount the `.iso` image file using this command: `mount -o loop BSP-Image-Name.iso /mnt/iso`
3. Install the BSP image file according to the instructions given in the BSP documentation.
4. Add CodeWarrior HyperTRK debug support to the hypervisor image (`hv.uImage`)

You can enable the HyperTRK debug support directly in the BSP. Alternatively, you can modify and build the HyperTRK manually, and then enable it in the hypervisor.

Perform the steps given in the subsections below:

- [Enabling HyperTRK Debug Support Directly in Build Tool](#) on page 314
- [Applying New HyperTRK Patches from CodeWarrior Install Layout](#) on page 314
- [Modifying and Building HyperTRK Manually](#) on page 314