Freescale Semiconductor
Application Note

Document Number: AN4980

Linux kernel and modules debug using CodeWarrior for QorIQ LS series – ARM V7 ISA

Contents

1. Introduction

This document describes the steps required for Linux kernel and modules debugging using the CodeWarrior for QorIQ LS series – ARM V7 ISA.

This document includes the following sections:

- Build the Linux sources.
- Perform Linux kernel and modules debug in CodeWarrior for QorIQ LS series – ARM V7 ISA.

2. Preliminary background

The following are the steps required to compile Linux for the LS1021AQDS board.

2.1. Downloads

Before Linux kernel and modules debug, download LS1021A SDK ISO.

2.2. Install the SDK

To install SDK on the host machine, perform these steps:

1. Mount the ISO on your machine

   ```
   $ sudo mount -o loop LS1021A-SDK-<version>-<target>-<yyyymmdd>-
   yocto.iso /mnt/cdrom
   ```

2. As a non-root user, install the SDK:

   ```
   $ cd /mnt/cdrom
   $ ./install
   ```

3. When prompted to input the install path, ensure that the current user has correct permissions for the install path.

NOTE	There are no uninstall scripts. To uninstall Yocto, you can remove the `<yocto_install_path>/LS1021A-SDK-<version>-<target>-<yyyymmdd>-yocto` folder manually.

2.3. Host Environment

Yocto requires some packages to be installed on host folder. The following steps are used for preparing Yocto:

1. `$ cd <yocto_install_path>`
2. `$ ./scripts/host-prepare.sh`
3. `$ source ./fsl-setup-poky -m <machine>`

NOTE	For example, for LS1021AQDS board, the above command will be: `$ source ./fsl-setup-pocky -m ls1021aqds -j 4 -t 4`, where `-j` is the number of jobs to spawn during the compilation stage and `-t` is the number of BitBake tasks that can be issued in parallel.

2.4. Builds

To build various packages, the following steps need to be performed:

```
1. $ cd <yocto_install_path>/
2. $ source ./build_<machine>_release/SOURCE_THIS
3. $ bitbake <package-recipe>
```

NOTE	u-boot, rcw, kernel, dtb, and rootfs images are can be found in: `build_<machine>_release/tmp/deploy/images/<machine>`.

2.5. Linux Kernel

In some cases, it is necessary to configure and rebuild the Linux Kernel. In our case, this is necessary for adding the debug symbols.

1. Do menuconfig

   ```
   $ bitbake –c menuconfig virtual/kernel
   ```

2. The kernel configuration window will be opened. Go to ***Kernel hacking > Compile-time checks and compiler*** *option* and select **Compile the kernel with debug info** checkbox.
3. Save the new configuration and rebuild the Linux Kernel

   ```
   $ bitbake –c virtual/kernel
   ```

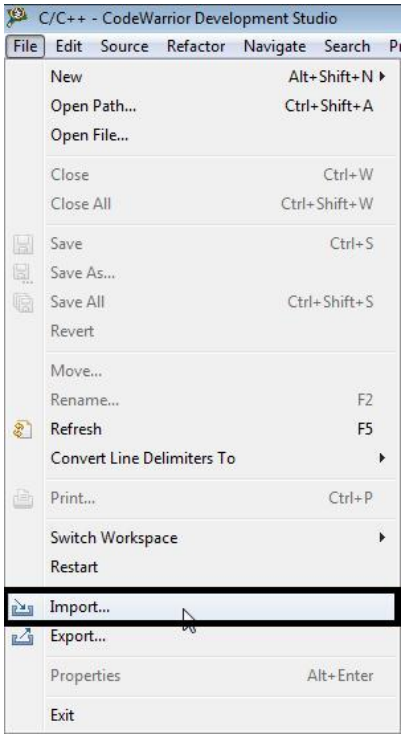
NOTE	**a.** vmlinux image with debug symbols can be found in folder: `build_<machine>_release/tmp/work/<machine>-fsl-linux-gnueabi/ linux-layerscape-sdk/3.12-r0/git/` **b. v**mlinux elf file will be imported in CodeWarrior for QorIQ LS series – ARM V7 ISA.

3. Create ARMv7 project

To create an ARMv7 project for Linux kernel debug, follow these steps:

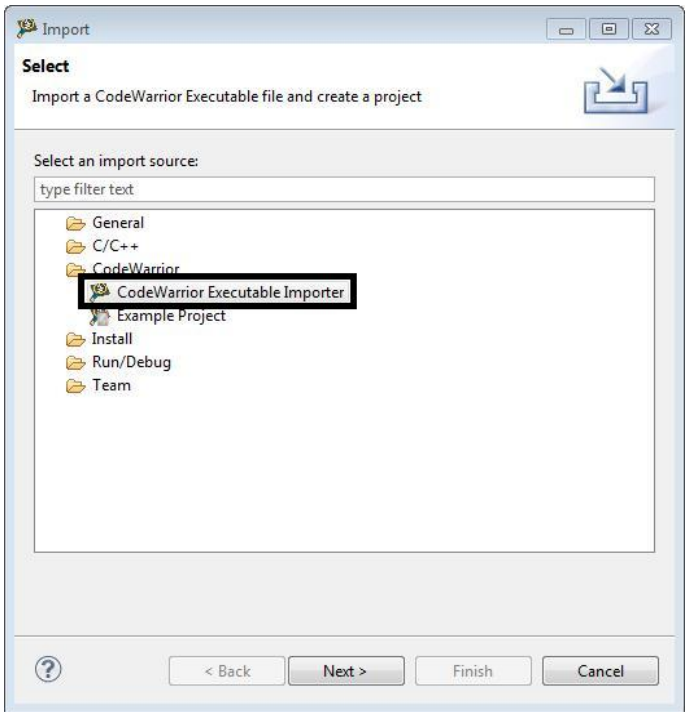
1. Start CodeWarrior for QorIQ LS series – ARM V7 ISA.
2. Select **File** > **Import** and import the vmlinux executable file generated during the Linux kernel compilation. For details, see Linux Kernel section.

Figure 1. CodeWarrior File menu



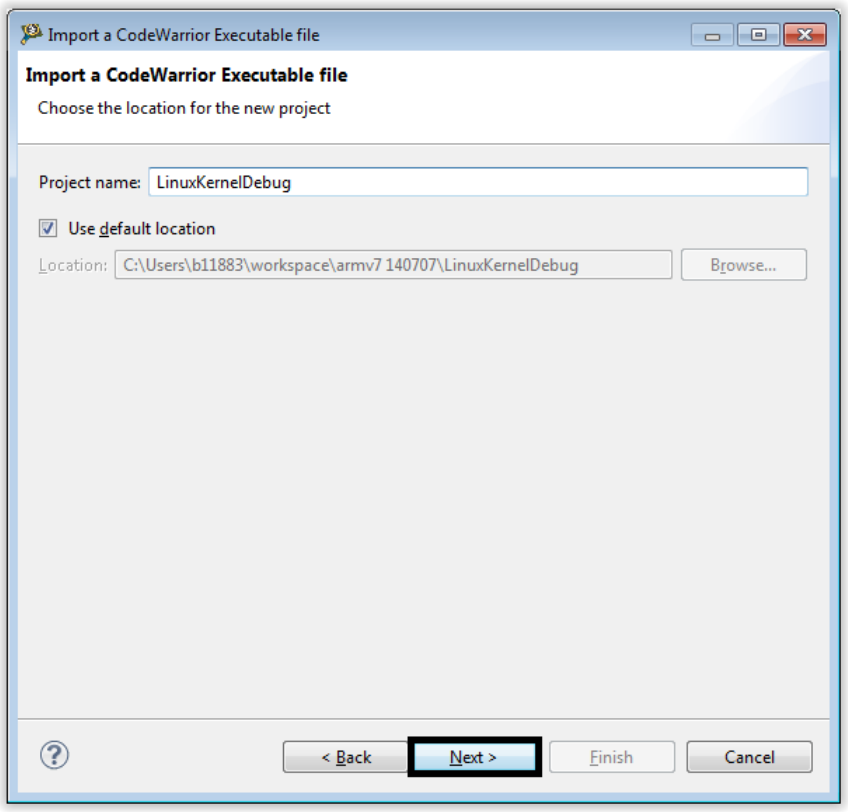
3. Choose the source to **Import** and click **Next**.

Figure 2. Import wizard



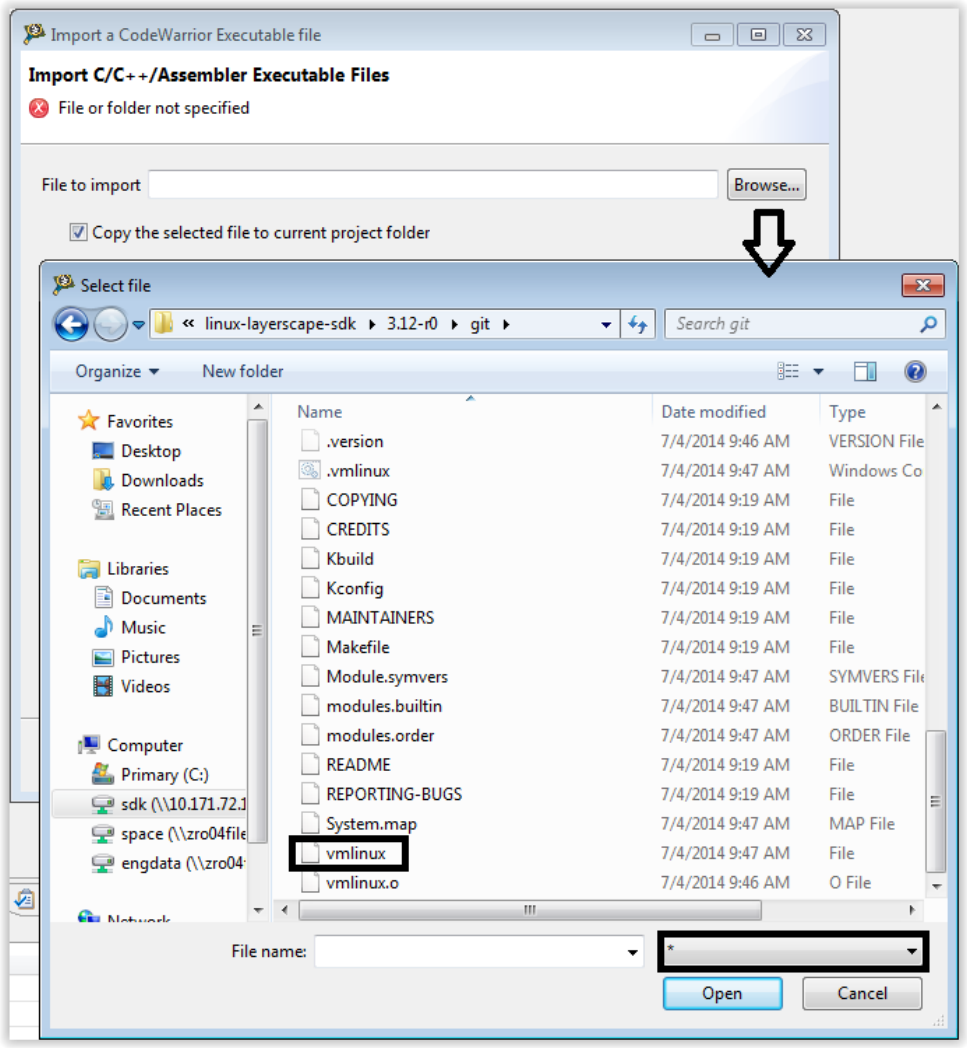
4. Specify **Project name** and **Location**, or use the default location and click **Next**.

Figure 3. Importing executable file page



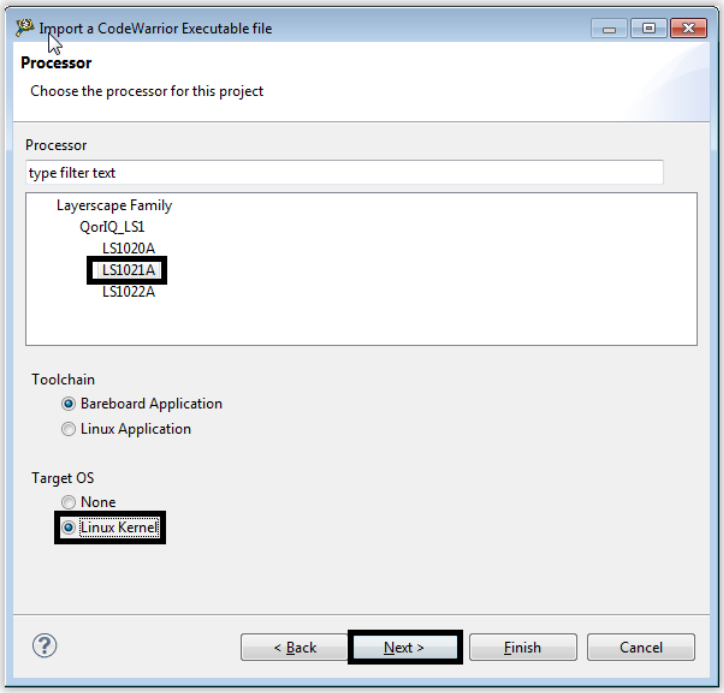
5. Browse to the vmlinux executable file and select O**pen**. By default, CodeWarrior looks for an `.elf` extension, so change the file type in the lower right corner of **Select File** dialog.

Figure 4. Select vmlinux executable file page



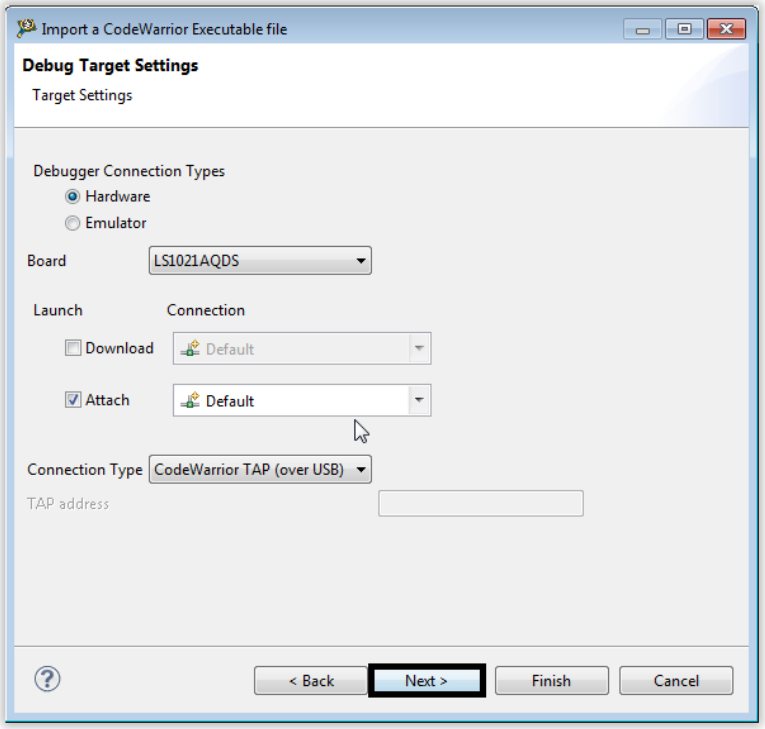
6. Select **Processor** type, **Linux Kernel** as Target OS and click **Next**.

Figure 5. Processor page



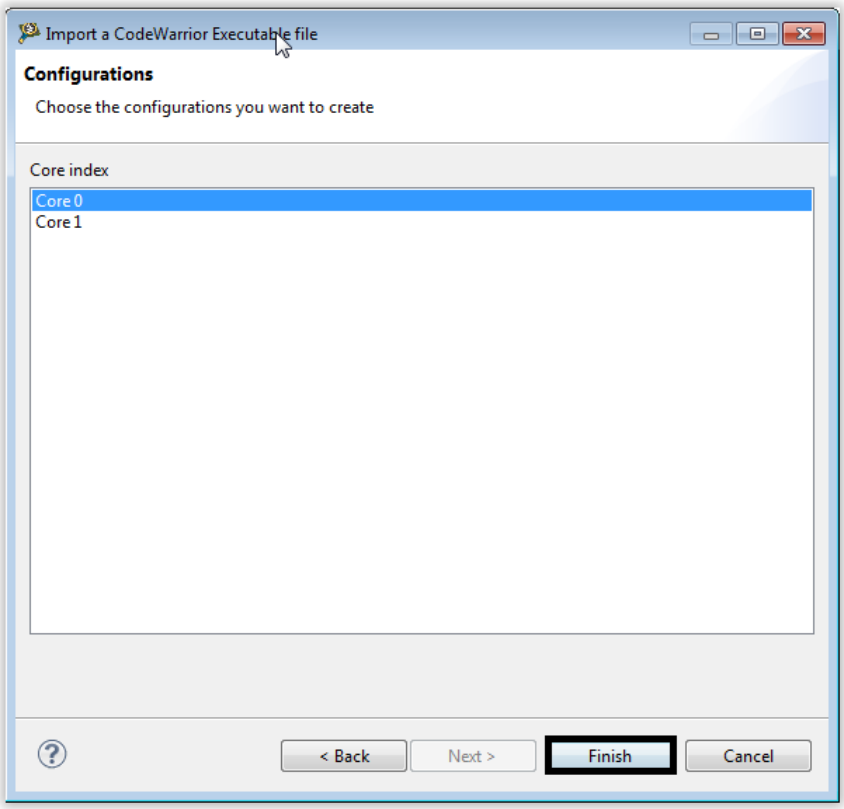
7. Select **Debugger Connection Types**, **Board**, **Launch**, **Connection Type** and click **Next**.

Figure 6. Target Settings page



8. Select the **Configurations** that you want to create and then click **Finish** to close the wizard.

Figure 7. Configurations page



4. Linux kernel debug support

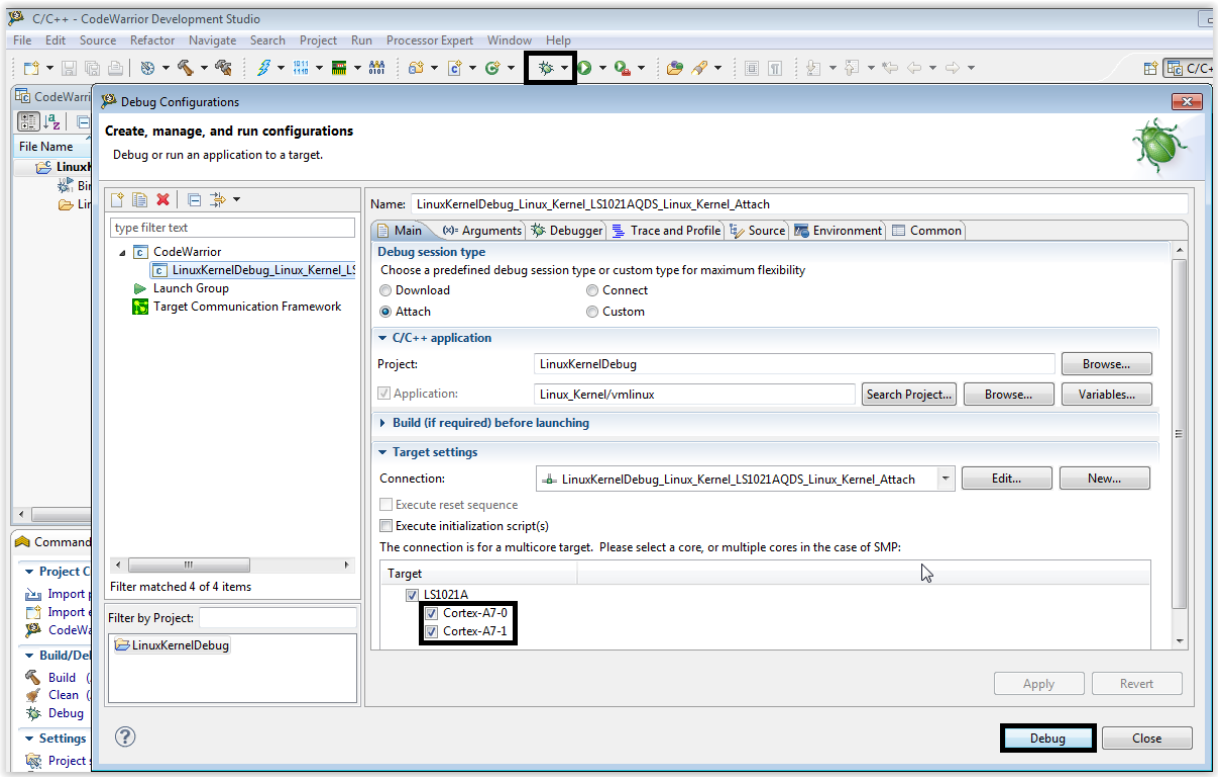
4.1. Debugger settings for Linux kernel debugging

The vmlinux executable file generated during the Linux kernel compilation should be imported as CodeWarrior project (for more information, see Create ARMv7 project section).

After the CodeWarrior project is created, perform these steps to start Linux kernel debug:

1. Select **Run** > **Debug configurations**, to open **Debug configurations** dialog and click **Debug**.

Figure 8. Debug Configurations dialog

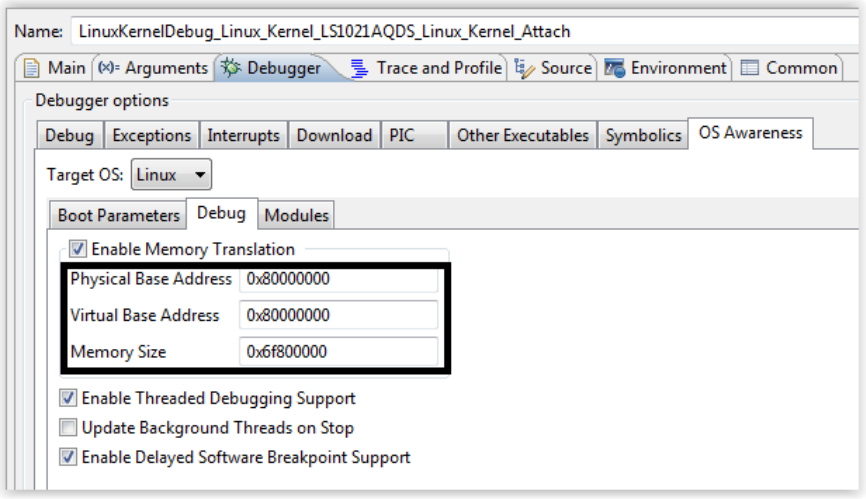


NOTE	Both cores should be selected. Also, make sure no initialization file is used.

2. Go to **Debugger** -> **OS Awareness**. Since an Attach Launch configuration is used to attach to a running Linux kernel, make sure that all the checkboxes are disabled.

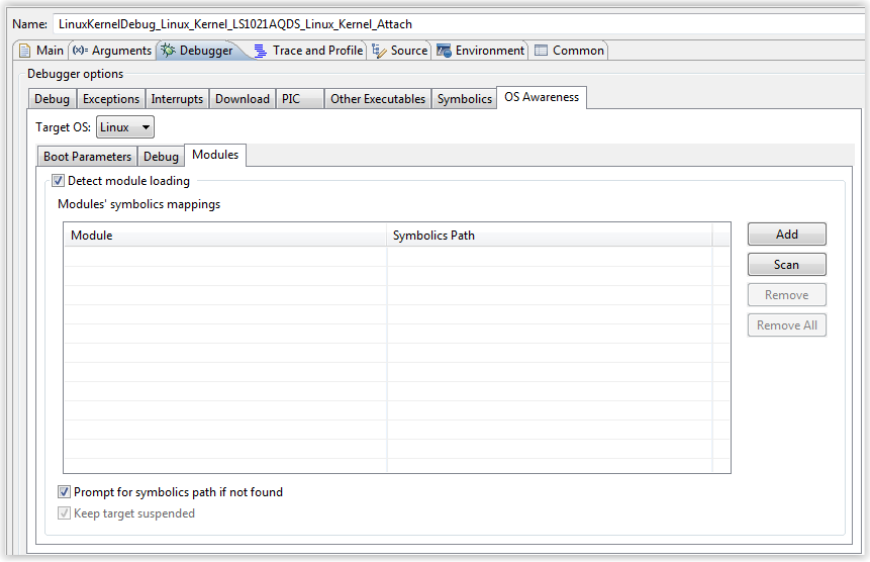
3. In the **Debug** tab, you need to select the **Enable Memory Translation** checkbox and use the settings as described in the figure below for early kernel debug capabilities. Also, select the **Enable Threaded Debugging Support** and **Enable Delayed Software Breakpoint Support** checkboxes.

Figure 9. OS Awareness – Debug tab



4. In the **Modules** tab, you need to select the **Detect module loading** and **Prompt for symbolics path if not found** checkboxes. These options are necessary for automatic insertion/removal detection of kernel modules.

Figure 10. OS Awareness – Module tab



5. Linux kernel and module debug

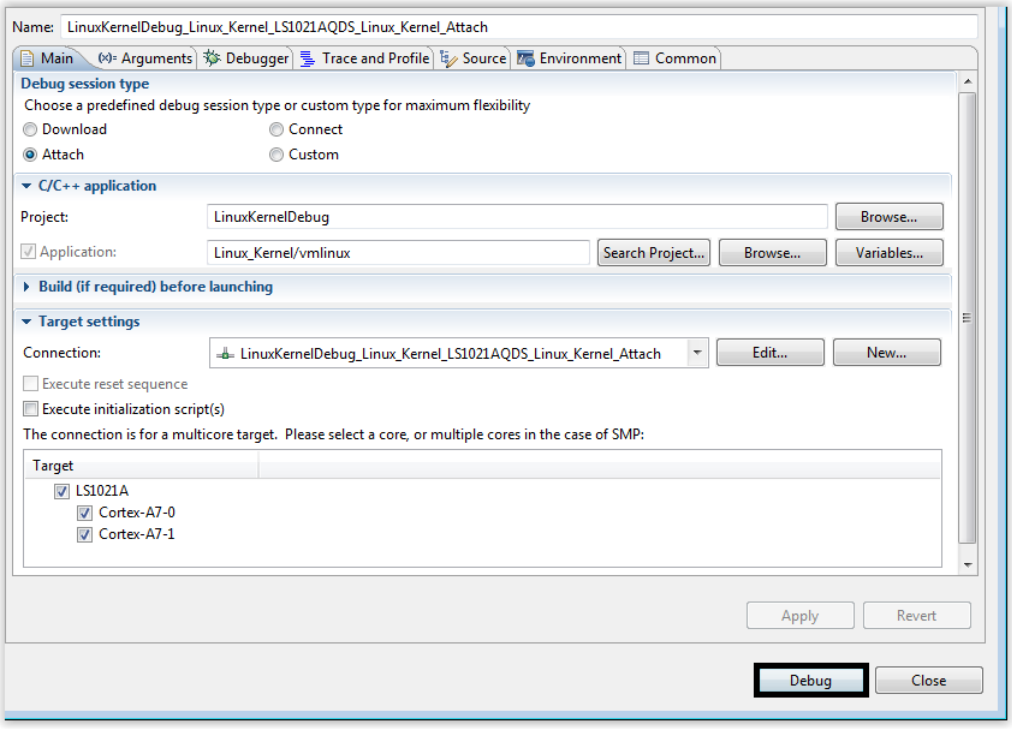
5.1. Linux kernel debugging

1. Power on the board and stop to U-Boot console.

Figure 11. Target stopped at u-boot prompt

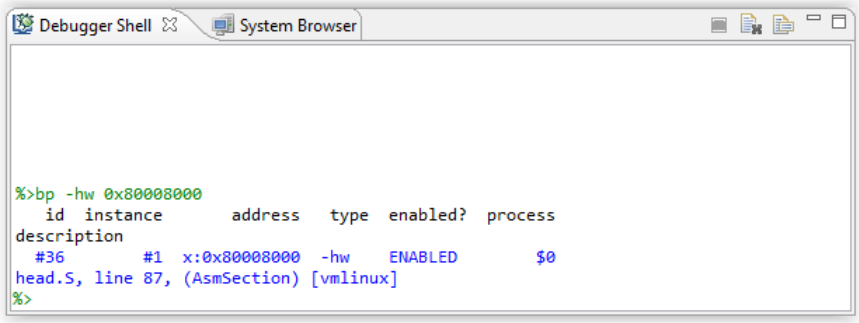
2. Attach to U-Boot using Attach launch.

Figure 12. The Attach Launch configuration



3. Set a breakpoint at kernel entry point, using **Debugger Shell** command: `bp -hw 0x80008000`

Figure 13. Setting breakpoint at entry point in Debugger Shell



4. Start kernel from U-Boot console.

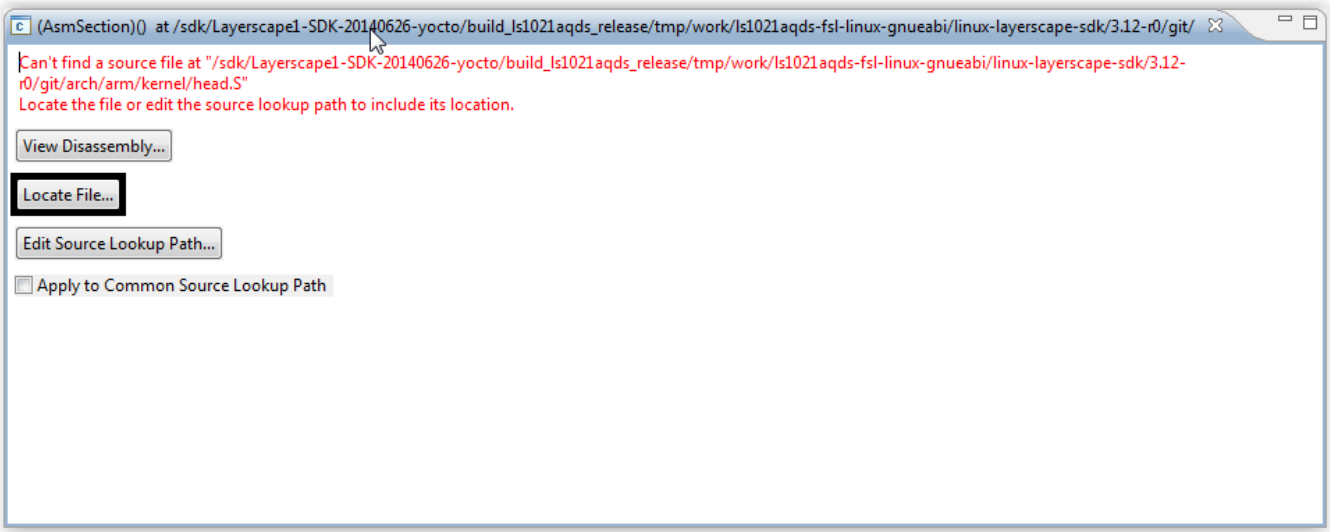
Figure 14. U-boot log - preparing the images for starting the Linux kernel

```
## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Linux-3.12.0+
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3053688 Bytes = 2.9 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 88000000 ...
   Image Name:   fsl-image-core-ls1021aqds-201406
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
   Data Size:    19170910 Bytes = 18.3 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 8f000000
   Booting using the fdt blob at 0x8f000000
   Loading Kernel Image ... OK
   Loading Ramdisk to cedb7000, end cffff65e ... OK
   Loading Device Tree to cedae000, end cedb6a91 ... OK

Starting kernel ...
```

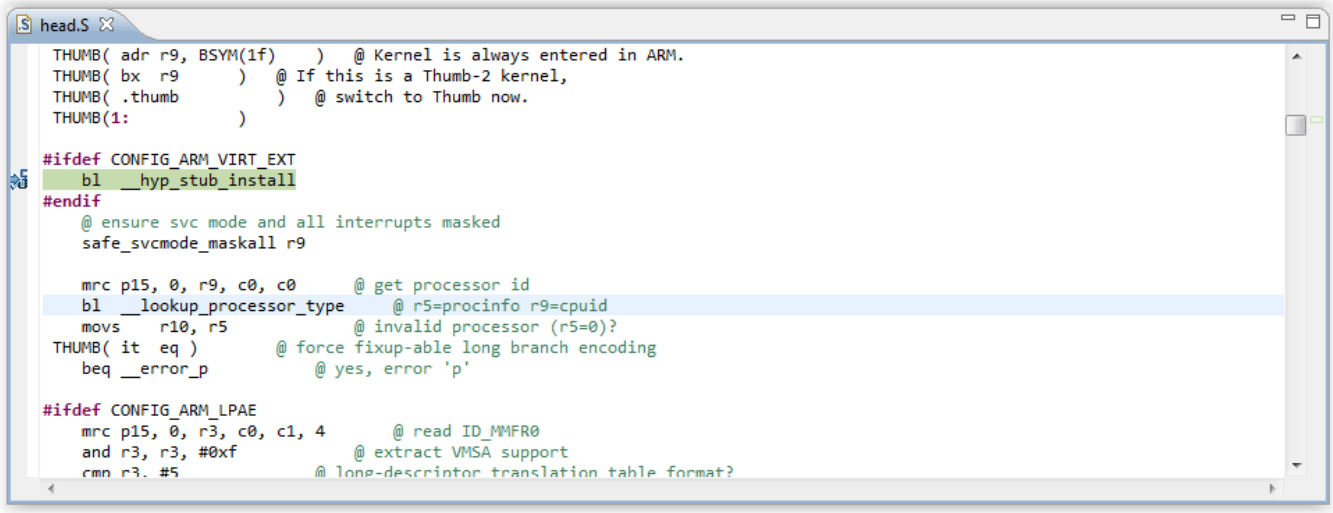
5. The breakpoint set at step 5 will be hit and CodeWarrior will ask for the location of the Linux kernel sources in order to make a path mapping between the original location of the sources and the new location (for example, the Linux kernel sources were copied from a Linux machine on a Windows machine).

Figure 15. Source file not found when target is stopped at kernel entry point



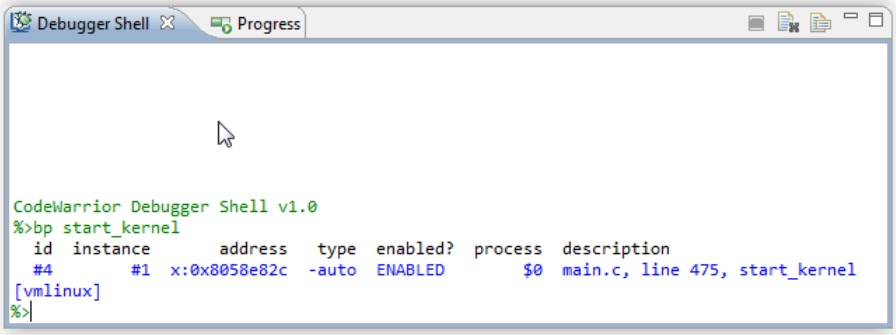
6. After locating the missing file, CodeWarrior will display in Source view the actual source file.

Figure 16. Target stopped at entry point, after path mapping was performed



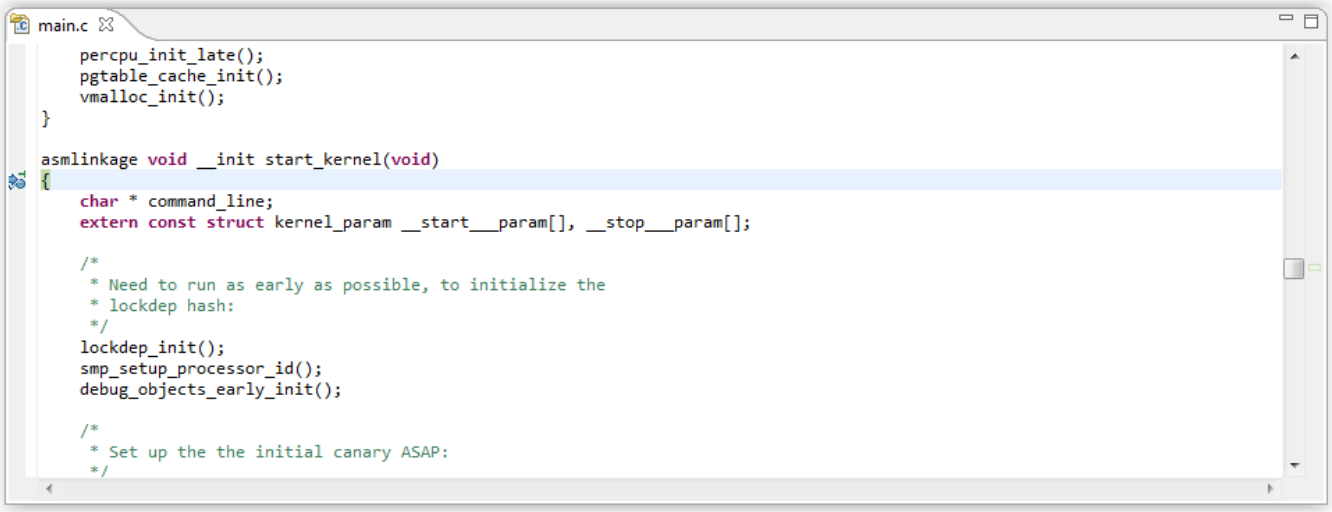
7. To start kernel debug from `start_kernel` symbol, set a breakpoint at `start_kernel`, using **Debugger Shell** command: `bp start_kernel`.

Figure 17. Setting a breakpoint from Debugger Shell at 'start_kernel' method



8. **Resume** using F8 or use **Debugger Shell** command: `go`. The breakpoint will be hit and kernel debugging can be performed from `start_kernel`.

Figure 18. Target stopped at 'start_kernel' method

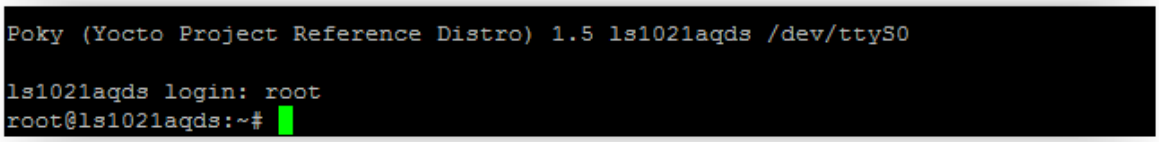


9. At this point, you are able to perform full Linux kernel debug using run control (step/run/suspend), set/remove breakpoints, read/write memory/registers/variables, etc.

5.2. Linux kernel module debugging

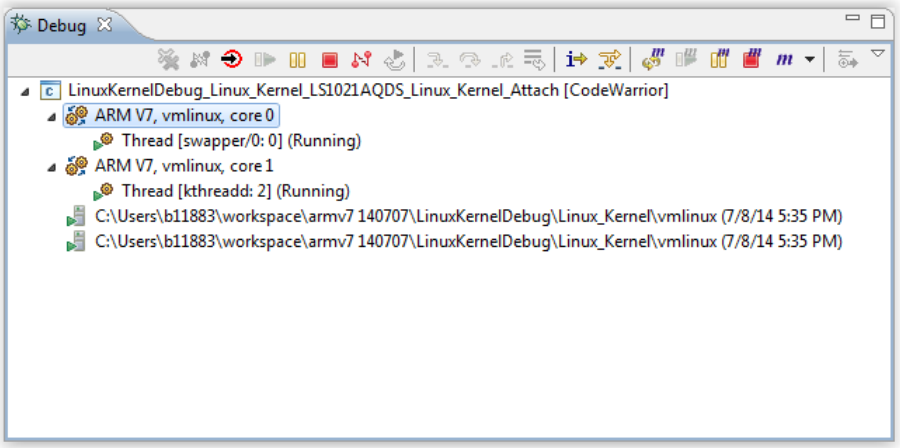
1. Login to Linux.

Figure 19. Linux prompt after logging in



2. Debugger is already attached to the target. If not, Attach to Linux using the Attach Launch configuration.

Figure 20. CodeWarrior attached to two running cores



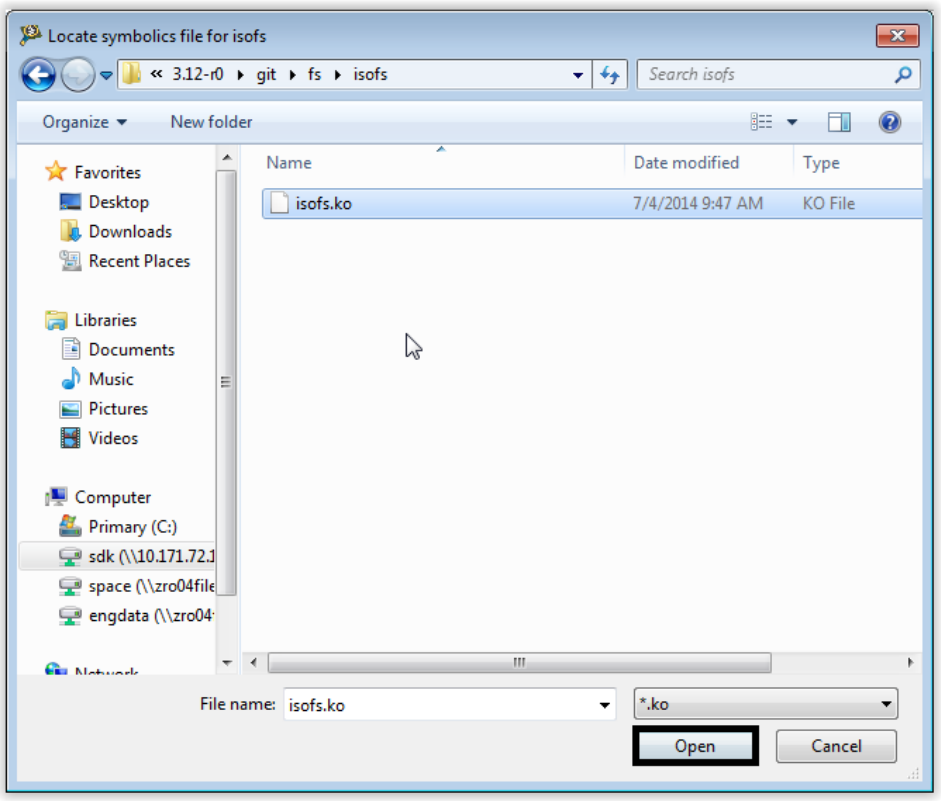
3. Insert a module into Linux.

```
root@ls1021aqds:~# modprobe isofs
```

4. CodeWarrior will automatically detect any `insmod/modprobe/rmmod` operation. A pop-up window will be triggered for locating the module debug symbols.

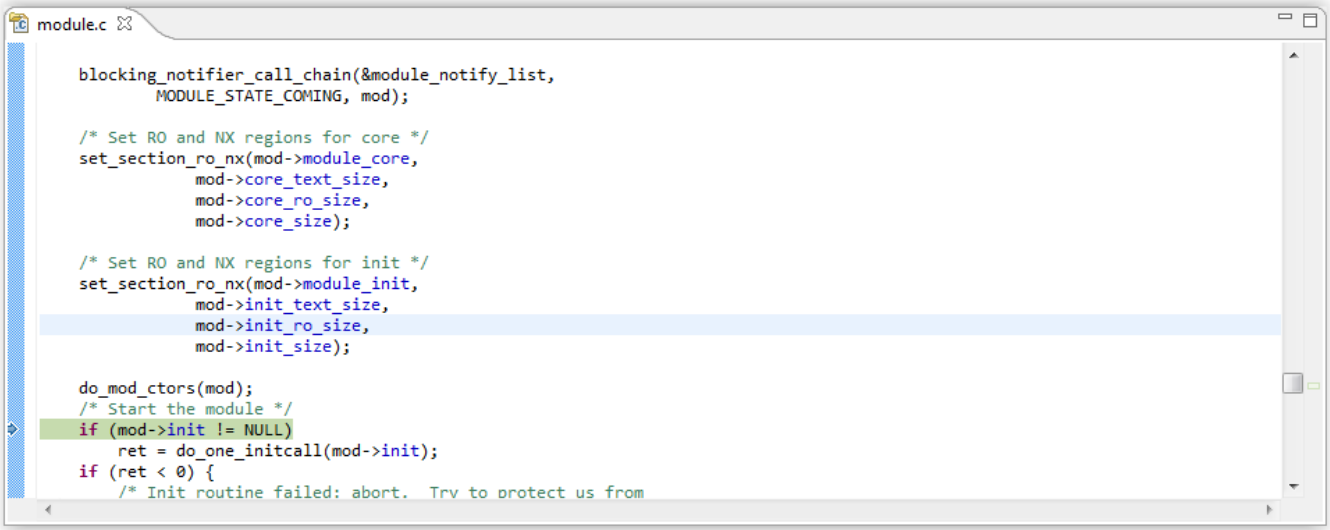
NOTE	In order to detect insertion/removal of kernel modules, CodeWarrior needs to be configured accordingly in the Debug configuration, **Debugger tab -> OS Awareness tab -> Modules** tab.

Figure 21. Prompt for 'isofs' symbolics file



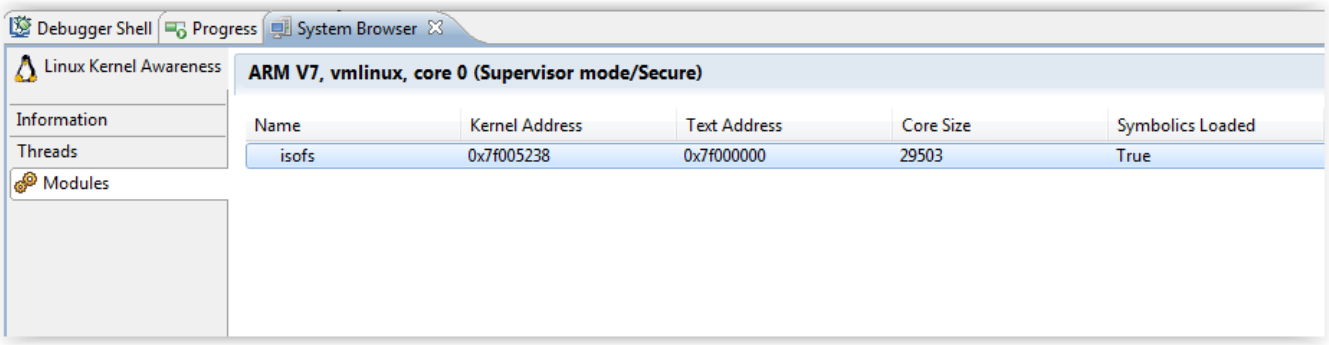
NOTE It is mandatory that the kernel image running on the target is the same with the vmlinux image in debugger, in order to have the kernel modules insertion/removal detection enabled.

Figure 22. Target stopped in do_init_module after detection that an insmod/modprobe was performed



5. **System Browser** can be used to see information about Kernel version, modules and threads running on each core.

Figure 23. Kernel modules list displayed in System Browser



6. For module debug, module's sources should be opened in CodeWarrior. Debugging (step, run, or breakpoint) can be done for the inserted module's.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

Document Number: AN4980

28 July 2014