Debugging a Linux Symmetric Multi-Processing (SMP) Kernel using DS-5

Version 1.0

Non-Confidential

Issue 01
102608_0100_01_en

Debugging a Linux Symmetric Multi-Processing (SMP) Kernel using DS-5

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2020	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Contents

1. Overview

You can develop, port, and debug the Linux kernel for a platform using DS-5 Debugger. You can set breakpoints, view registers, view memory, single step at source level, and perform all other standard debugging functions. You can debug the Linux kernel before the MMU is enabled (with a physical memory map), and after the MMU is enabled (with a virtual memory map). DS-5 allows you to do all this and more, not just for single-core platforms, but for SMP platforms too.

In DS-5, you configure a debugging session to a target using the **Debug Configuration** dialog. The Linux kernel debug options available in the dialog are primarily designed for post-MMU debug. But with some extra controls, you can use it for pre-MMU debug also. This makes it possible to debug the Linux kernel at source-level, all the way from its entry point, through the pre-MMU stages. You can then continue debugging seamlessly through the MMU enable stage to the post-MMU debug stage with full kernel awareness. This is achieved without disconnecting, reconfiguring, and reconnecting to your target.

This tutorial takes you through:

- Creating a debug configuration, and loading the Linaro Linux 3.4 SMP kernel on a Cortex-A9x4 Fixed Virtual Platform (FVP) model supplied with DS-5.
- Debugging the kernel - pre-MMU stage.
- Debugging the kernel - post-MMU stage.
- Viewing the structure of the kernel.
- Multi-core features.

This tutorial also briefly looks at Logging features, Single-stepping individual cores or threads, and Using the command line to view cores, thread, and process information.

2. Prerequisites

In the following, you can find the prerequisites to this tutorial:

- **DS-5 installation and licensing**
  You need to have Arm DS-5 installed and the license to use it. If you do not have a license, see the Getting Started tutorial to see how to obtain a license.
- **Enable telnet on your Windows host**
  The FVP model uses a telnet client for its simulated serial ports. Telnet is disabled by default in newer versions of Windows. You need to enable it. Check the documentation for your Windows version for details about enabling the telnet client.
- **DS-5 Linux Distribution Example**
  Download and import the **DS-5 Linux Distribution Example** into DS-5. This archive file contains the Linux kernel built with debug information, the `vmlinux` symbol file, file system, and the full source code.

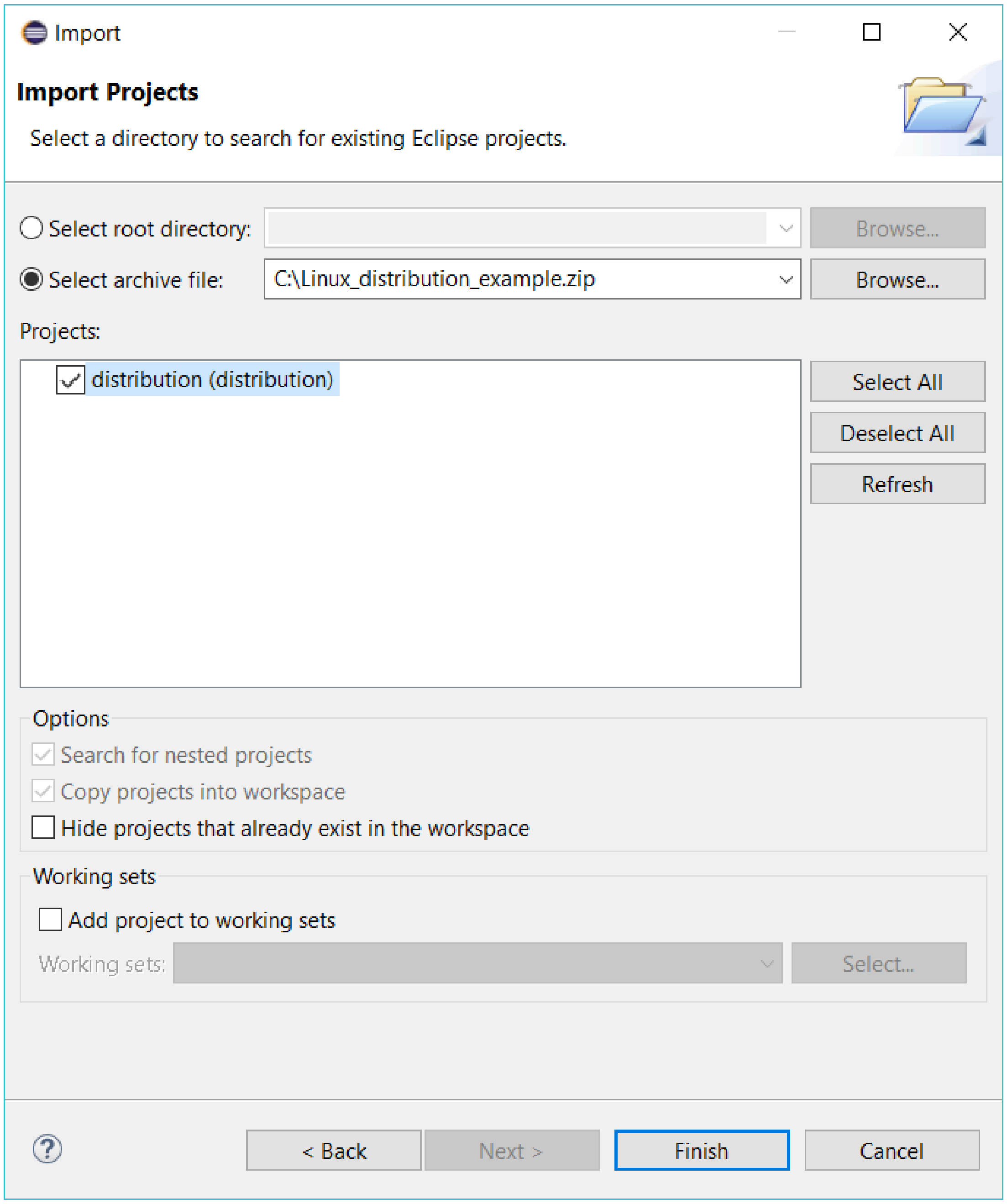
Note

If you are working on a Windows machine, you will encounter issues with file naming incompatibilities between Linux and Windows. These can be ignored for the purpose of this tutorial.

---

1. Go to the DS-5 Downloads page and download the `Linux_distribution_example.zip` file from the **DS-5 Extra Downloads** section.
2. In DS-5, select **File** > **Import** to display the **Import** dialog.
3. Under **General**, select **Existing Projects into Workspace** and click **Next**.
4. Select the **Select archive file** option, and **Browse** and select the file you downloaded from the Arm download center. The **distribution** project is available in the projects area.

Figure 2-1: Import projects window



Select root directory:

[✓ Select archive file:] C:\Linux_distribution_example.zip

Projects:

- [✓] distribution (distribution)

Options

- [✓] Search for nested projects
- [✓] Copy projects into workspace
- [ ] Hide projects that already exist in the workspace

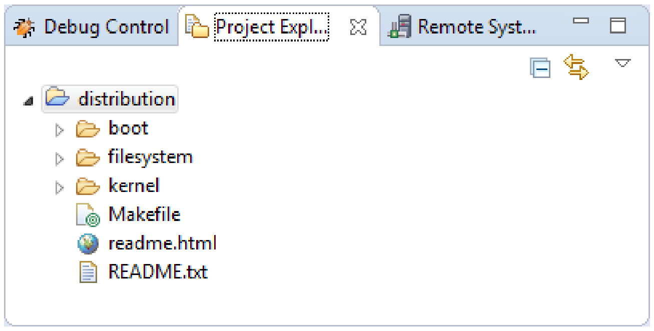
Working sets

- [ ] Add project to working sets

Working sets:

5. Click **Finish** to import the files into the project folder. You can view the imported files in the **Project Explorer**.

Figure 2-2: Project Explorer folder view



distribution
- boot
- filesystem
- kernel
- Makefile
- readme.html
- README.txt

- **Set up the sources**

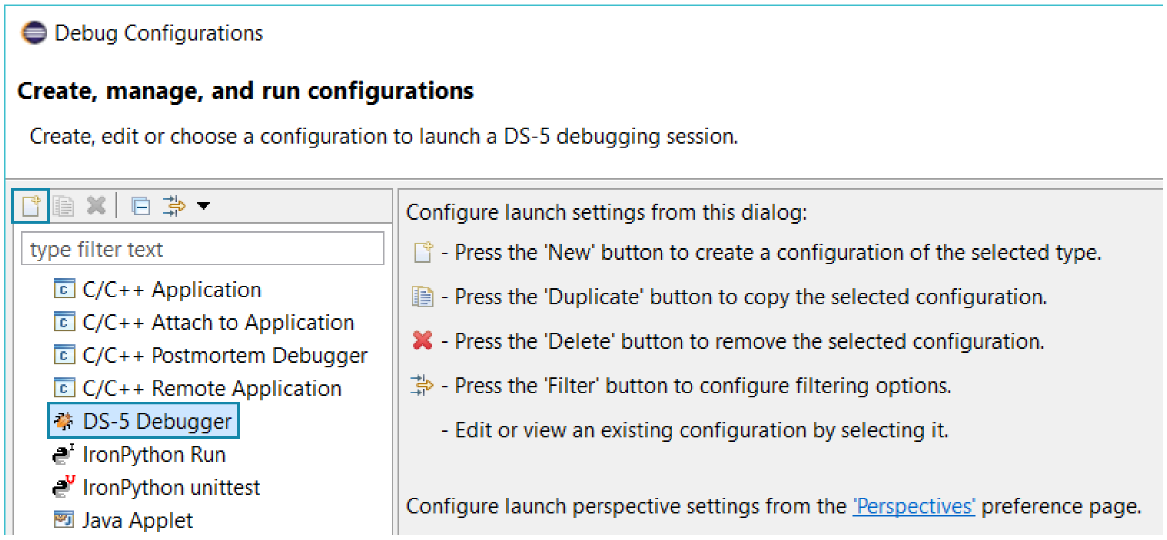
  The DS-5 Linux Distribution Example archive file also contains the required sources. You can find it under `<DS-5 Workspace>\distribution\kernel\linux-linaro-3.4-rc3-2012.04-0-patched\source.tar.bz2`. Unzip this file into a location on your workstation.

3. Creating a debug configuration

Start by creating a **Debug Configuration** to configure connections to the Fixed Virtual Platform (FVP) model. When you create the **Debug Configuration**, you set up the kernel to load on the FVP.

1. Launch DS-5.
2. From the main menu, select **Run** > **Debug Configurations** to open the **Debug Configurations** dialog.
3. In the **Debug Configurations** dialog:
   a. Select **DS-5 Debugger**.
   b. Click the **New launch configuration** button.

Figure 3-1: New launch configuration


Debug Configurations

Create, manage, and run configurations

Create, edit or choose a configuration to launch a DS-5 debugging session.

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
- DS-5 Debugger
- IronPython Run
- IronPython unittest
- Java Applet

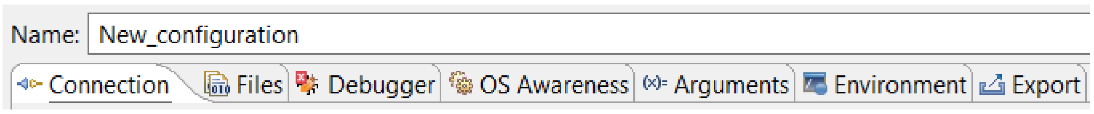
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

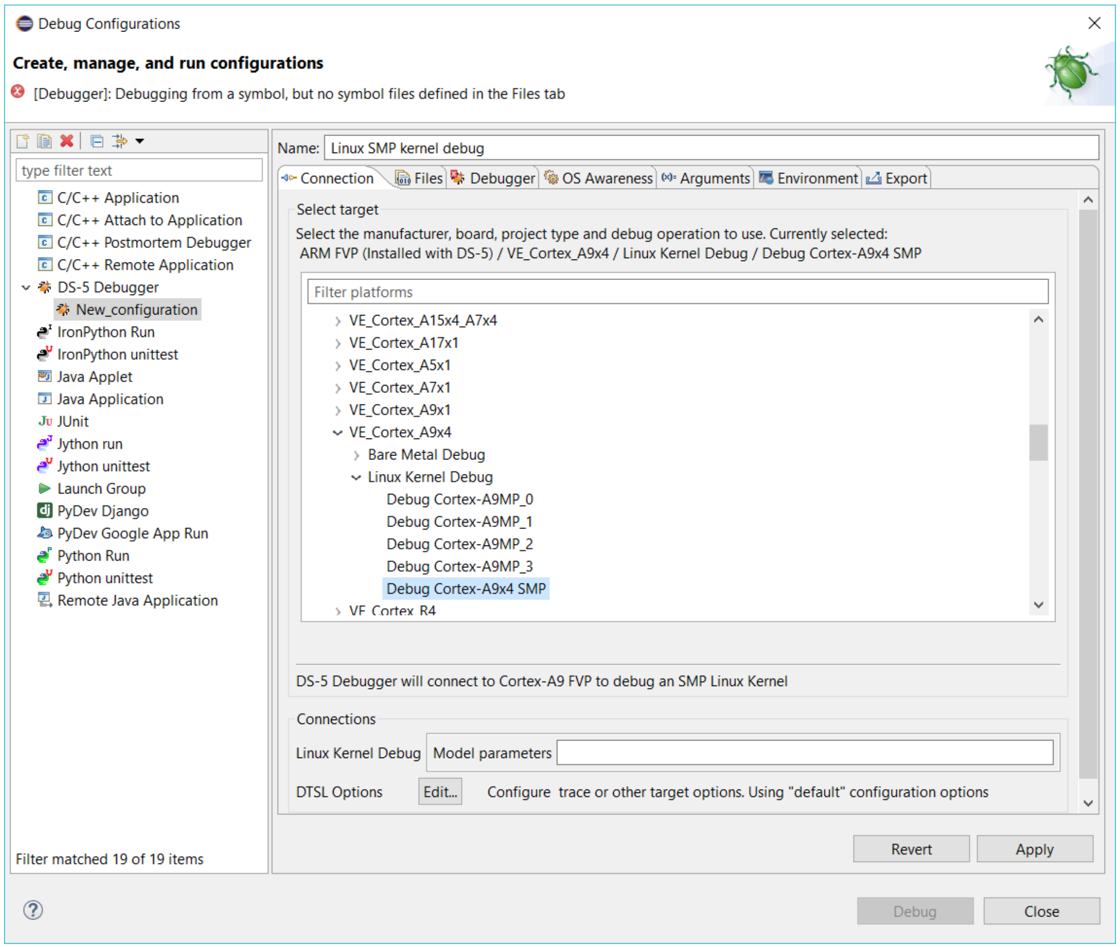
This creates a DS-5 debug configuration and displays the various tabs required to specify settings for loading your application on the target.

Figure 3-2: New configuration


Name: New_configuration

   c. Give a name to the debug configuration. For example, **Linux SMP kernel debug**.
   d. In the **Connection** tab:
      1. Under **Select target**, browse and select **Arm FVP (Installed with DS-5)** > **VE_Cortex-A9x4** > **Linux Kernel Debug** > **Debug Cortex-A9x4 SMP**.

Figure 3-3: Select target



[Debugger]: Debugging from a symbol, but no symbol files defined in the Files tab

Name: Linux SMP kernel debug

Select target

Select the manufacturer, board, project type and debug operation to use. Currently selected:
ARM FVP (Installed with DS-5) / VE_Cortex_A9x4 / Linux Kernel Debug / Debug Cortex-A9x4 SMP

DS-5 Debugger will connect to Cortex-A9 FVP to debug an SMP Linux Kernel

Connections

Linux Kernel Debug | Model parameters

DTSL Options | Edit... | Configure trace or other target options. Using "default" configuration options

2. Under **Connections** > **Linux Kernel Debug**, in the **Model parameters** field, enter the following as a single line:

Note

- The code below is listed in separate lines for reading clarity. It must be in a single line with single spaces between the commands when copied to the Model parameters field.
- Replace <DS-5 Install folder> with the path to your DS-5 installation, and <DS-5 Workspace> with the path to your workspace.

```
--data "<DS-5 Install folder>\arm\linux_distribution\kernel_ve@0x80008000"

--data "<DS-5 Install folder>\arm\linux_distribution\rtsm_ve-
cortex_a9x4.dtb@0x80f00000"

-C motherboard.mmc.p_mmc_file="<DS-5 Install folder>\arm\linux_distribution
\rootfs.image"
```

```
-C motherboard.vfs2.mount="<DS-5 Workspace>"

-C motherboard.terminal_3.start_telnet=false

-C motherboard.terminal_3.mode=raw

-C motherboard.pl011_uart3.untimed_fifos=true

-C motherboard.terminal_3.start_port=5003

-C motherboard.smsc_91c111.enabled=1

-C motherboard.hostbridge.userNetworking=1

-C motherboard.hostbridge.userNetPorts="8080=8080"
```

e. In the **Files** tab:

   1. Under **Target Configuration** > **Application on host to download** field, click **File System**.
   2. Browse to your DS-5 installation folder, and select `<DS-5 Install folder>\arm\linux_distribution\bootwrapper_ve.axf` to load the bootwrapper.

      What is a boot wrapper?

      The boot wrapper (`bootwrapper_ve.axf`) is a small program that must execute before starting the kernel.

      The boot wrapper is responsible for setting up the execution environment for the kernel, including:

      - Detecting the processor type.
      - Initializing the vector table.
      - Setting up the RAM.
      - Initializing a serial port.
      - Handling the device tree and kernel command line.
      - And finally, starting the kernel.
   3. Deselect the **Load symbols** option. The kernel debug information is loaded later.

f. In the **Debugger** tab, under **Run Control**, select **Debug from entry point**.

g. Click **Debug** to start debugging.

4. If the **Confirm Perspective Switch** dialog is displayed, click **Yes** to switch to the **DS-5 Debug Perspective**.

DS-5 connects to the model and displays the connection status in the **Debug Control** view and opens a window.

You have now created a debug configuration, loaded the files on the FVP, and started the debug connection. The next step is to start debugging the kernel.

4. Debugging the kernel - pre-MMU stage

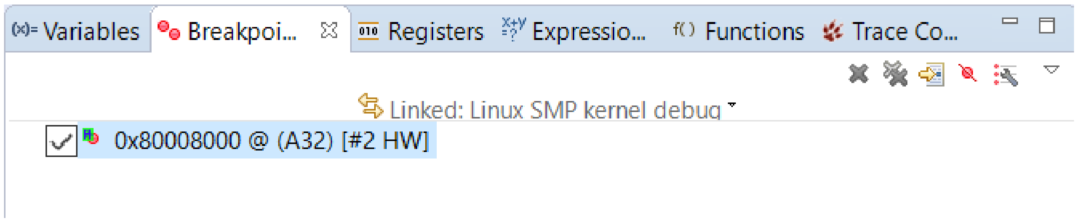
The following steps shows you how to debug the kernel in the pre-MMU stage:

Set a temporary hardware breakpoint on the entry point into the kernel

1. In the **Commands** view, enter thbreak `0x80008000` to set a temporary hardware breakpoint on the entry point into the kernel. `0x80008000` is the base address in RAM where this Linux kernel is loaded, and is also the entry point for the kernel. It is the same address as specified in the `--data "<DS-5 Install folder>\arm\linux_distribution\kernel_ve@0x80008000"` parameter to the FVP model. This address is also hard-coded within the boot wrapper as the address to jump to when the boot wrapper has completed its setup tasks.

   You can view the breakpoint in the Breakpoints view.

   Figure 4-1: Breakpoint view



0x80008000 @ (A32) [#2 HW]

2. In the **Debug Control** view, click **Continue** . to start the target. The code is executed and stops at the breakpoint.

   At this stage, you can:

   - either continue with the next step - **Disable OS awareness before loading the debug symbols**
   - or analyze the system registers in the **Registers** view:

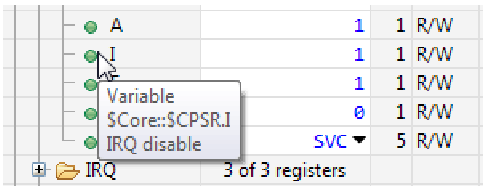
Analizing the system registers in the Registers view:

1. You can use the **Registers** view to check if the **Core** and **CP15** system registers are set, as recommended by the Booting Arm Linux reference page (For Armv8 targets use the AArch64 Linux reference page).

   For example, using the **Registers** view you can:

   - Check the Core is in SVC (supervisor) mode with both `IRQ` and `FIQ` interrupts disabled.

   Figure 4-2: IRQ and FIQ interrupts disabled

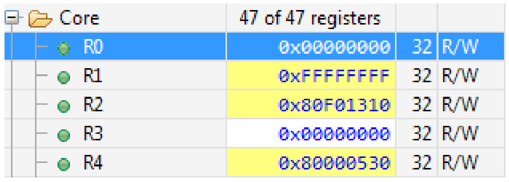


A	1	1	R/W
I	1	1	R/W
	1	1	R/W
	0	1	R/W
	SVC	5	R/W
IRQ	3 of 3 registers		

Variable
$Core::$CPSR.I
IRQ disable

- Check `R0` is `0`.

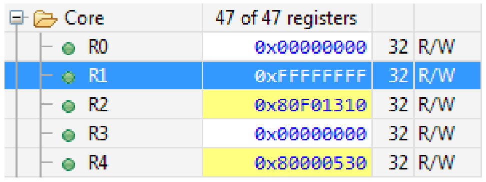
Figure 4-3: `R0` is `0`



Core	47 of 47 registers		
R0	0x00000000	32	R/W
R1	0xFFFFFFFF	32	R/W
R2	0x80F01310	32	R/W
R3	0x00000000	32	R/W
R4	0x80000530	32	R/W

- Check `R1` contains `0xFFFFFFFF` to indicate this is a “device tree” platform.

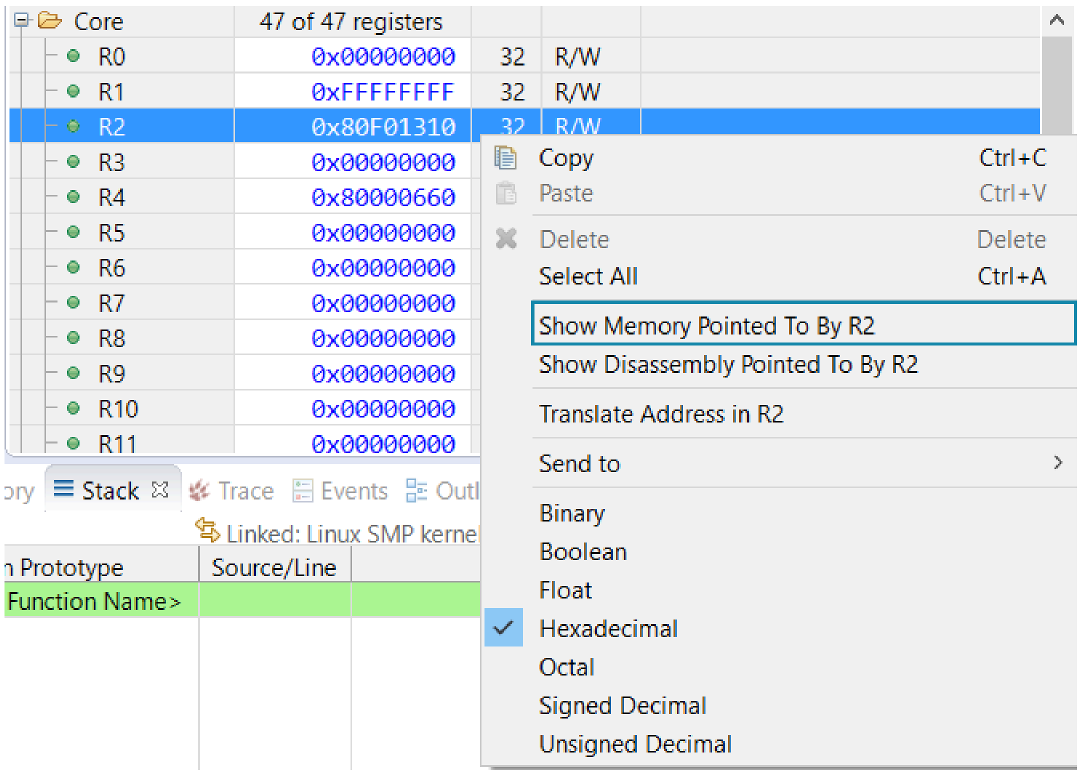
Figure 4-4: `R1` contains `0xFFFFFFFF`



Core	47 of 47 registers		
R0	0x00000000	32	R/W
R1	0xFFFFFFFF	32	R/W
R2	0x80F01310	32	R/W
R3	0x00000000	32	R/W
R4	0x80000530	32	R/W

- Check `R2` contains a pointer to the device tree. Right-mouse click `R2` and select **Show Memory Pointed To By R2**.

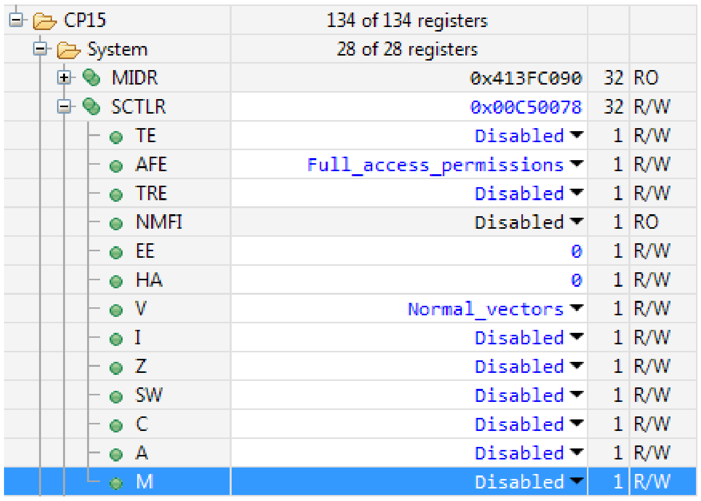
Figure 4-5: `R2` contains a pointer to the device tree



Core	47 of 47 registers		
R0	0x00000000	32	R/W
R1	0xFFFFFFFF	32	R/W
R2	0x80F01310	32	R/W
R3	0x00000000		
R4	0x80000660		
R5	0x00000000		
R6	0x00000000		
R7	0x00000000		
R8	0x00000000		
R9	0x00000000		
R10	0x00000000		
R11	0x00000000		

- Expand `CP15` > `System` > `SCTLR` > `M` and check that the MMU is off.

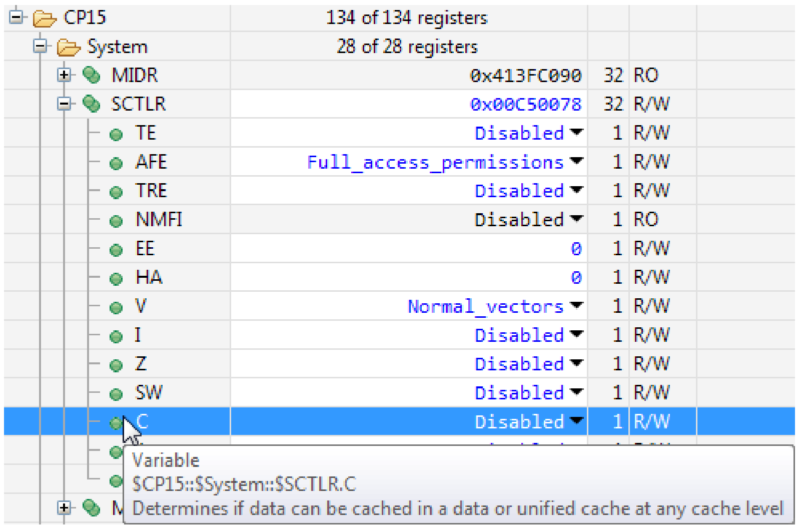
Figure 4-6: MMU is disabled



CP15	134 of 134 registers		
System	28 of 28 registers		
MIDR	0x413FC090	32	RO
SCTLR	0x00C50078	32	R/W
TE	Disabled	1	R/W
AFE	Full_access_permissions	1	R/W
TRE	Disabled	1	R/W
NMFI	Disabled	1	RO
EE	0	1	R/W
HA	0	1	R/W
V	Normal_vectors	1	R/W
I	Disabled	1	R/W
Z	Disabled	1	R/W
SW	Disabled	1	R/W
C	Disabled	1	R/W
A	Disabled	1	R/W
M	Disabled	1	R/W

- Expand `CP15` > `System` > `SCTLR` > `C` and check that the data cache is off.

Figure 4-7: Data cache is disabled

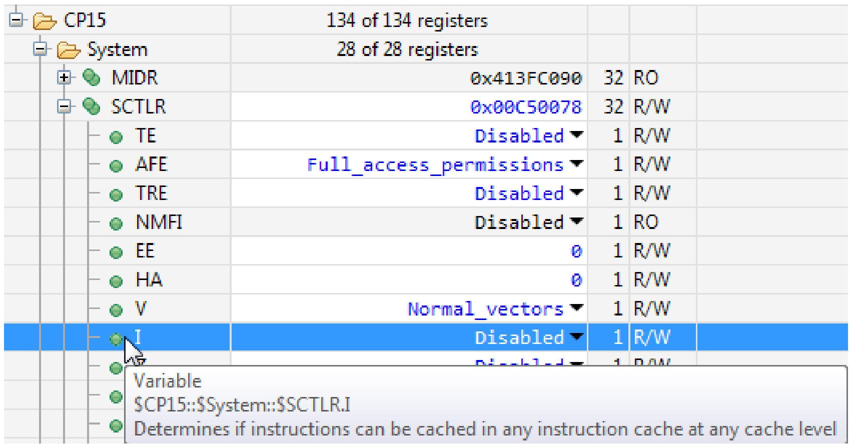


CP15	134 of 134 registers		
System	28 of 28 registers		
MIDR	0x413FC090	32	RO
SCTLR	0x00C50078	32	R/W
TE	Disabled	1	R/W
AFE	Full_access_permissions	1	R/W
TRE	Disabled	1	R/W
NMFI	Disabled	1	RO
EE	0	1	R/W
HA	0	1	R/W
V	Normal_vectors	1	R/W
I	Disabled	1	R/W
Z	Disabled	1	R/W
SW	Disabled	1	R/W
C	Disabled	1	R/W

Variable
$CP15::$System::$SCTLR.C
Determines if data can be cached in a data or unified cache at any cache level

- Expand `CP15` > `System` > `SCTLR` > `I` and check the status of the instruction cache. Depending on the status, this could be either on or off.

Figure 4-8: Check the instructrion cache status



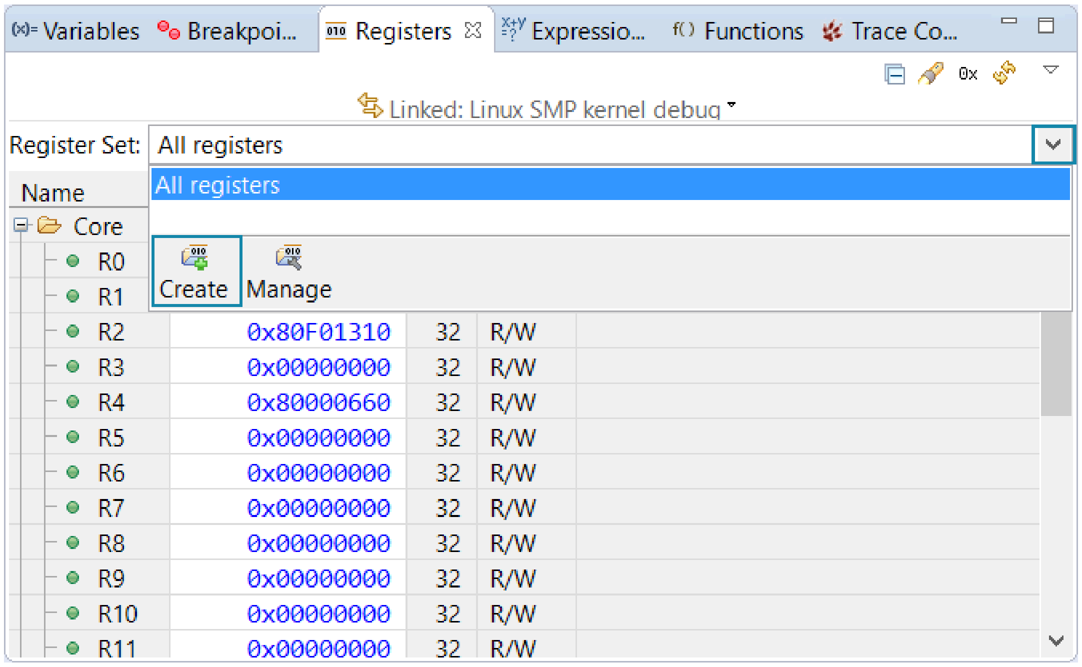
CP15	134 of 134 registers		
System	28 of 28 registers		
MIDR	0x413FC090	32	RO
SCTLR	0x00C50078	32	R/W
TE	Disabled	1	R/W
AFE	Full_access_permissions	1	R/W
TRE	Disabled	1	R/W
NMFI	Disabled	1	RO
EE	0	1	R/W
HA	0	1	R/W
V	Normal_vectors	1	R/W
I	Disabled	1	R/W

Variable
$CP15::$System::$SCTLR.I
Determines if instructions can be cached in any instruction cache at any cache level

2. You can set up the **Registers** view to filter the displayed additional information. For example, if you only want to view information present in the `CP15` group of registers, you have to create a new **Register set**:

- In the **Registers** view, use the drop down arrow to expand the viewed register sets and select **Create**.

Figure 4-9: Create registers



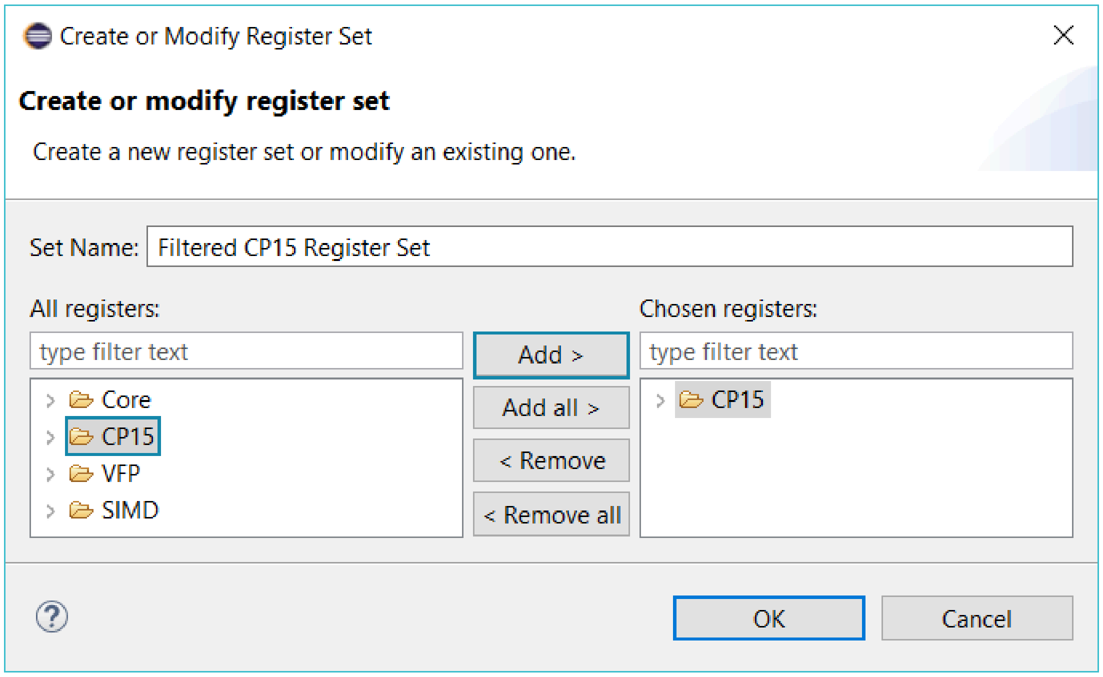
Register Set: All registers

All registers

Name			
Core			
R0			
R1			
R2	0x80F01310	32	R/W
R3	0x00000000	32	R/W
R4	0x80000660	32	R/W
R5	0x00000000	32	R/W
R6	0x00000000	32	R/W
R7	0x00000000	32	R/W
R8	0x00000000	32	R/W
R9	0x00000000	32	R/W
R10	0x00000000	32	R/W
R11	0x00000000	32	R/W

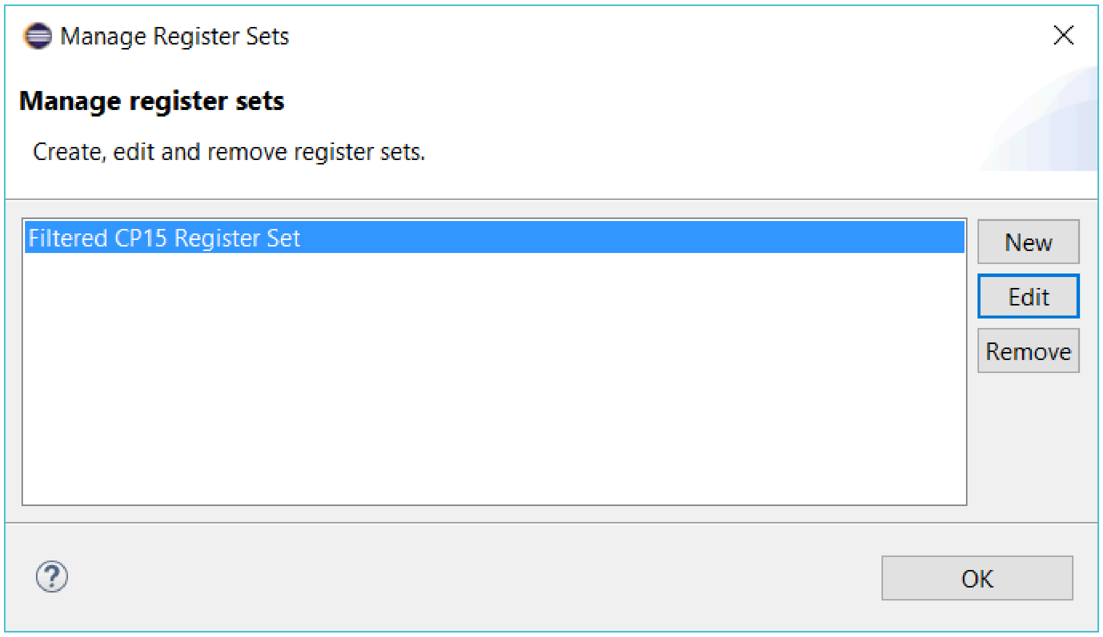
- In the **All registers** section, select `CP15` and click **Add >**. Your selected registers appear under **Chosen registers**. Give the register set a name such as **Filtered CP15 Register Set**. You can create multiple register groups if needed. To finish creating your new register sets, click **OK**.

Figure 4-10: Filtered CP15 Register Set


Create or Modify Register Set
Create or modify register set
Create a new register set or modify an existing one.
Set Name: Filtered CP15 Register Set
All registers:
type filter text
Core
CP15
VFP
SIMD
Add >
Add all >
< Remove
< Remove all
Chosen registers:
type filter text
CP15
OK
Cancel

- The **Registers** view displays the specific register group you selected.
- To switch between various register groups, click the All Registers drop down and select the group you want.
- To edit or remove existing register sets, in the Registers view, under the Register Set drop down, select Manage to open the Manage Register Set dialog.
  - To edit an existing register set, select a register set, and click Edit. Edit the register set in the Create or Modify Register Set dialog.
  - To remove a register set, select the register set you want to delete and click Remove.
  - Click OK to confirm your changes.

Figure 4-11: Manage register sets



Manage register sets

Create, edit and remove register sets.

Filtered CP15 Register Set

Disable OS awareness before loading the debug symbols

Since the MMU is currently off, DS-5 Debugger's OS support must be disabled before the debug symbols in the `vmlinux` file are loaded. This is to avoid the debugger from trying to read the kernel structures before they are set up (and possibly resulting in data aborts).

Enter `set os enabled off` in the **Command** view and click **Submit** or press **Enter**. This disables OS awareness.

Calculate the offset

The debug symbols in the `vmlinux` file have virtual addresses. So, when the `vmlinux` file is loaded, the debugger assumes that the operating system is up and running with the MMU enabled. But this also can be used to debug pre-MMU at source-level by applying an offset when loading the symbol file.

To calculate the offset, calculate the difference between the physical (P) and virtual (V) addresses of the code. For example, if the kernel is linked at virtual address `0xC0008000` and is loaded at physical address `0x80008000`, the offset is `-0x40000000`, which is `0x80008000 - 0xC0008000` (P-V).

For the DS-5 Linux Distribution example used in this tutorial, the kernel is linked at virtual address `0x80008000` and loaded at physical address `0x80008000`, so the offset is `0x0`.

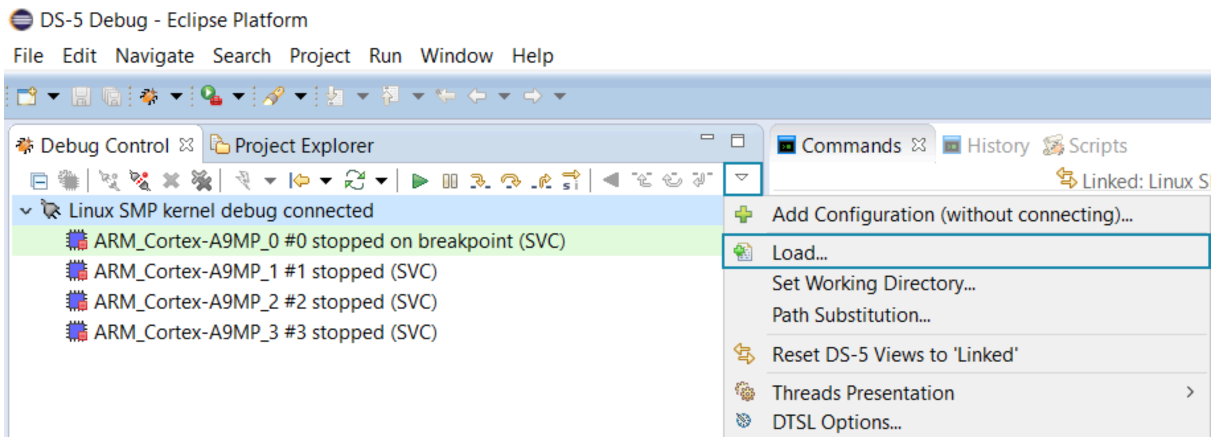
Load the vmlinux file

Now, load the `vmlinux` file with the kernel debug symbols required for this tutorial. The file is available in the Linux distribution example that you downloaded and imported into your workspace at the start of this tutorial.

Loading the `vmlinux` file enables you to perform source-level debugging of the Linux kernel. To load the file using the user interface:

1. In the **Debug Control** view, select the drop-down menu and select [Load].

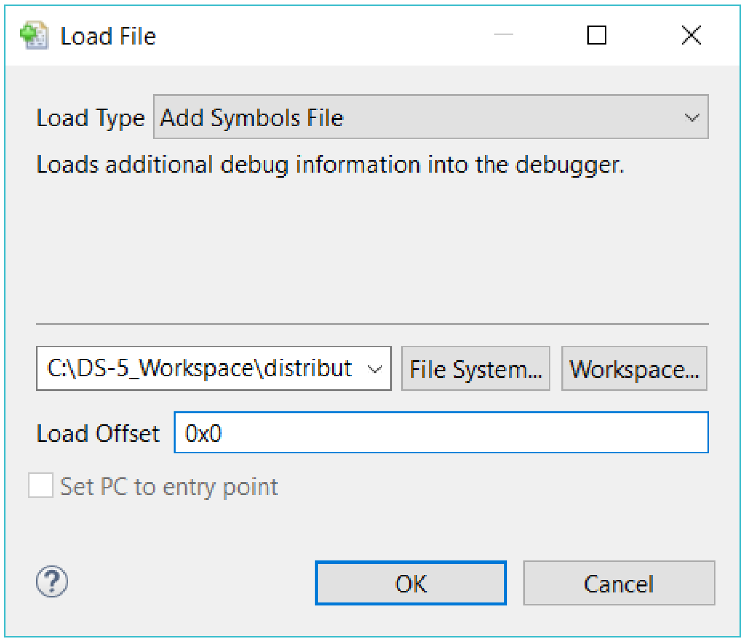
Figure 4-12: Debug control view



Linux SMP kernel debug connected
ARM_Cortex-A9MP_0 #0 stopped on breakpoint (SVC)
ARM_Cortex-A9MP_1 #1 stopped (SVC)
ARM_Cortex-A9MP_2 #2 stopped (SVC)
ARM_Cortex-A9MP_3 #3 stopped (SVC)

2. In the **Load File** dialog:
   a. **Load Type** - Select **Add Symbols File**
   b. **Workspace** - Click and select the vmlinux file. You can find this under `<DS-5 Workspace>\distribution\kernel\linux-linaro-3.4-rc3-2012.04-0-patched\built\VE_V7\vmlinux`
   c. [Load Offset] - Enter `0x0`.

Figure 4-13: Load file window



Load Type: Add Symbols File
Loads additional debug information into the debugger.
C:\DS-5_Workspace\distribut
Load Offset: 0x0
Set PC to entry point

   d. Click **OK**.

The `head.s` file opens in the **Editor** view.

You might need to single step once to open head.S in the editor. Also, if you get any **Source Not Found** alerts, you have to set the substitution path to the sources you set up as part of Prerequisites. The necessary substitution path to the `head.s` source file should be `<extraction_directory>/arch/arm/kernel`.

The **Disassembly** view shows the symbol `stext`, at the entry point for the kernel.

Loading the vmlinux file using the command line

You can also load the vmlinux file using the command-line. Use the `add-symbol-file` command to add the symbol file.

For this tutorial, enter: `add-symbol-file "<Your Project>\distribution\kernel\linux-linaro-3.4-rc3-2012.04-0-patched\built\VE_V7\vmlinux" 0x0` in the **Commands** view.

Replace `<Your Project>` with the name of the project you created as part of Prerequisites.

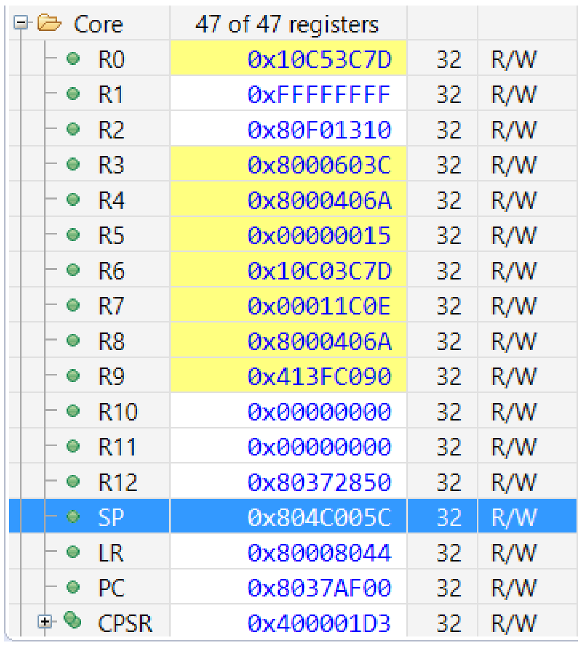
You can now, at source level, set breakpoints and watchpoints, view registers, view memory, single step, and other usual debug operations at this pre-MMU stage.

5. Debugging the kernel - post-MMU stage

Now that you have debugged the pre-MMU stage, we can move on to the post-MMU debugging stage.

1. In the **Commands** view, enter `thbreak __turn_mmu_on` to set a temporary hardware breakpoint to see when the MMU is turned on.
2. Click **Continue** ▶. (or F8 on your keyboard).
3. When the breakpoint at `__turn_mmu_on` is reached, note the value of `SP` (Stack Pointer) under the `Core` register group in the **Registers** view. It contains the virtual address of `__mmap_switched` and is the place that the code jumps to after the MMU is enabled. The value in this case is `0x804C005C`.

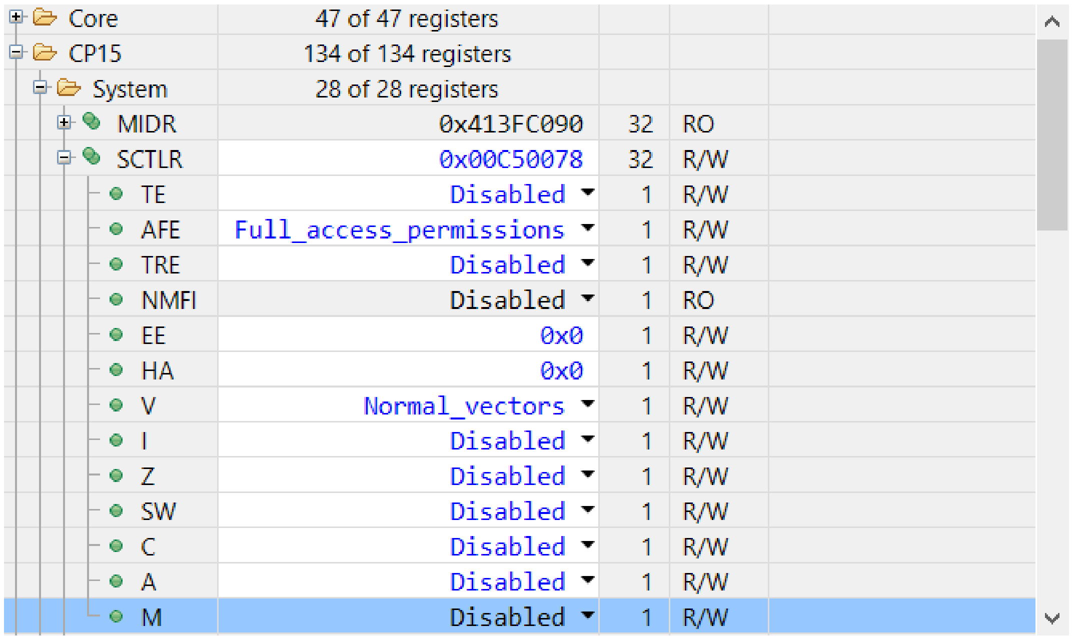
Figure 5-1: Stack Point value



Core	47 of 47 registers		
R0	0x10C53C7D	32	R/W
R1	0xFFFFFFFF	32	R/W
R2	0x80F01310	32	R/W
R3	0x8000603C	32	R/W
R4	0x8000406A	32	R/W
R5	0x00000015	32	R/W
R6	0x10C03C7D	32	R/W
R7	0x00011C0E	32	R/W
R8	0x8000406A	32	R/W
R9	0x413FC090	32	R/W
R10	0x00000000	32	R/W
R11	0x00000000	32	R/W
R12	0x80372850	32	R/W
SP	0x804C005C	32	R/W
LR	0x80008044	32	R/W
PC	0x8037AF00	32	R/W
CPSR	0x400001D3	32	R/W

You can also view the status of the MMU. In the **Registers** view, expand **CP15** > **SCTLR** > **M**. You can see that it is disabled at this point.

Figure 5-2: MMU status

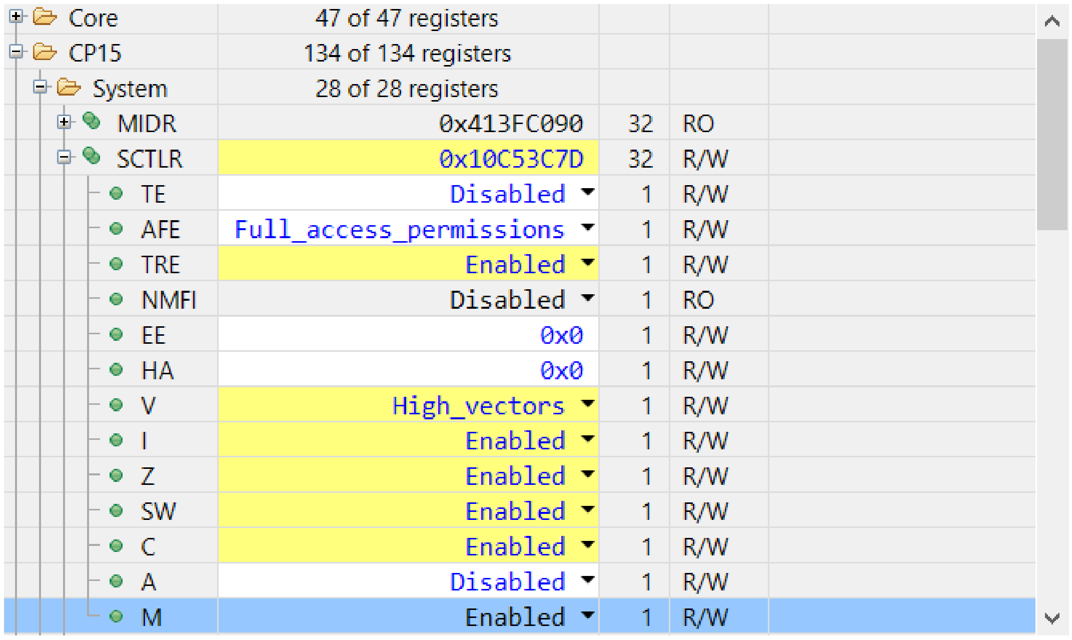


Core	47 of 47 registers		
CP15	134 of 134 registers		
System	28 of 28 registers		
MIDR	0x413FC090	32	RO
SCTLR	0x00C50078	32	R/W
TE	Disabled	1	R/W
AFE	Full_access_permissions	1	R/W
TRE	Disabled	1	R/W
NMFI	Disabled	1	RO
EE	0x0	1	R/W
HA	0x0	1	R/W
V	Normal_vectors	1	R/W
I	Disabled	1	R/W
Z	Disabled	1	R/W
SW	Disabled	1	R/W
C	Disabled	1	R/W
A	Disabled	1	R/W
M	Disabled	1	R/W

4. At the **Command** prompt, enter `thbreak *$SP` to place a temporary hardware breakpoint on the virtual address of `__mmap_switched`.

5. Click **Continue** to run the code. When the breakpoint at `__mmap_switched` is hit, the MMU is on. You can view the updated status of the MMU in the **Registers** view.

Figure 5-3: Updated status of the MMU



Core	47 of 47 registers		
CP15	134 of 134 registers		
System	28 of 28 registers		
MIDR	0x413FC090	32	RO
SCTLR	0x10C53C7D	32	R/W
TE	Disabled	1	R/W
AFE	Full_access_permissions	1	R/W
TRE	Enabled	1	R/W
NMFI	Disabled	1	RO
EE	0x0	1	R/W
HA	0x0	1	R/W
V	High_vectors	1	R/W
I	Enabled	1	R/W
Z	Enabled	1	R/W
SW	Enabled	1	R/W
C	Enabled	1	R/W
A	Disabled	1	R/W
M	Enabled	1	R/W

6. Enter `file` at the command prompt to discard currently loaded symbols.

This is done because the old set of symbols no longer apply, now that the MMU is on.

7. Now, reload the symbol file without the offset applied, so that post-MMU debugging will be done using unadjusted virtual addresses.

   At the **Command** prompt, enter: `add-symbol-file "<DS-5 Workspace>\distribution\kernel\linux-linaro-3.4-rc3-2012.04-0-patched\built\VE_V7\vmlinux"` in the **Commands** view.

   Replace `<DS-5 Workspace>` with the location of your workspace.

   The source code for`__mmap_switched` in `head-common.S` appears in the **Editor**.

You can now set breakpoints, view registers, view memory, single step, and other usual debug operations at this post-MMU stage, all at source level.

6. Viewing the structure of the kernel

The following steps shows you how to view the structure of the kernel:

1. After all the architecture-specific setup is done, the main C code entry into the kernel is in the `start_kernel()` function. This is available in the source at: `<DS-5 Workspace>\distribution\kernel\linux-linaro-3.4-rc3-2012.04-0-patched\source\init\main.c`.

   Enter `thbreak start_kernel` at the **Command** prompt to set a temporary hardware breakpoint on it.

2. Click **Continue** to run the code to reach the breakpoint.

   This displays the `main.c` source in the **Editor**.

   Again, if you get any **Source Not Found** alerts, you have to set the substitution path to the sources you set up as part of Prerequisites. The necessary substitution path to the `main.c` source file should be `<extraction_directory>/init/`.

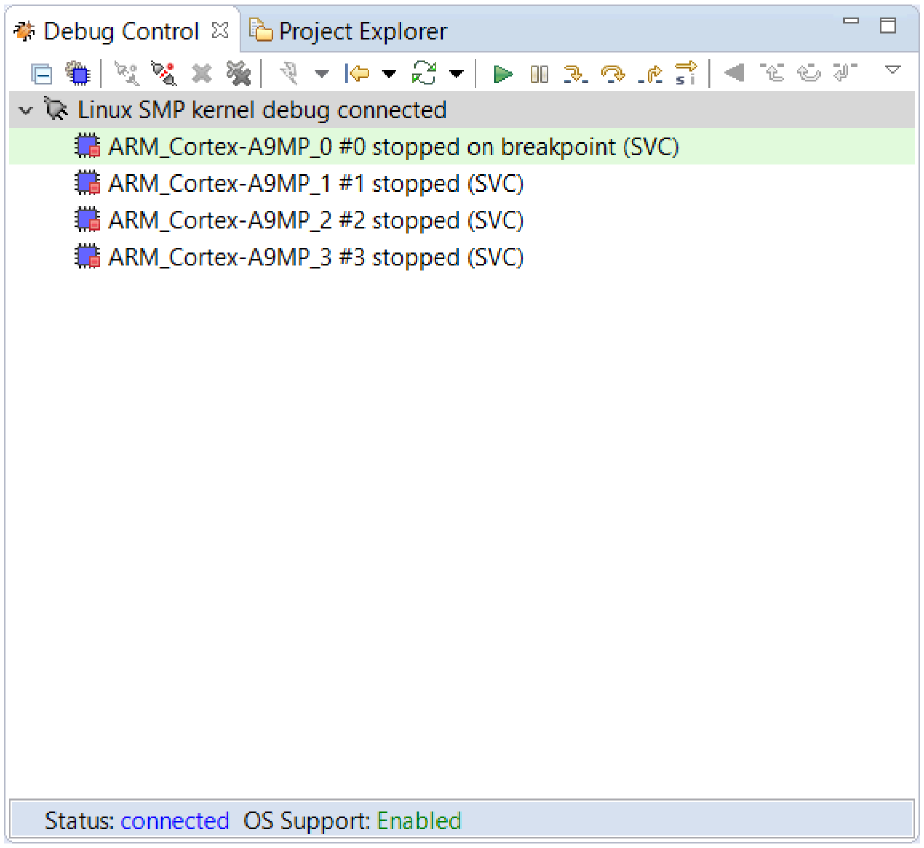
3. Enter `set os enabled` on to enable OS support in the DS-5 Debugger.

   - When OS support is enabled, the debugger also displays the Linux kernel support details. For example, you should see something similar to: `Enabled Linux kernel support for version "Linux 3.4.0-rc3-ve-v7-ds5 #1 SMP Mon Oct 31 00:17:24 GMT 2016 arm"`.

   - You can click in the **Debug Control** view to switch display to show threads. Similarly, you can click to show cores.

     In this example, you can view four cores:

Figure 6-1: Core view

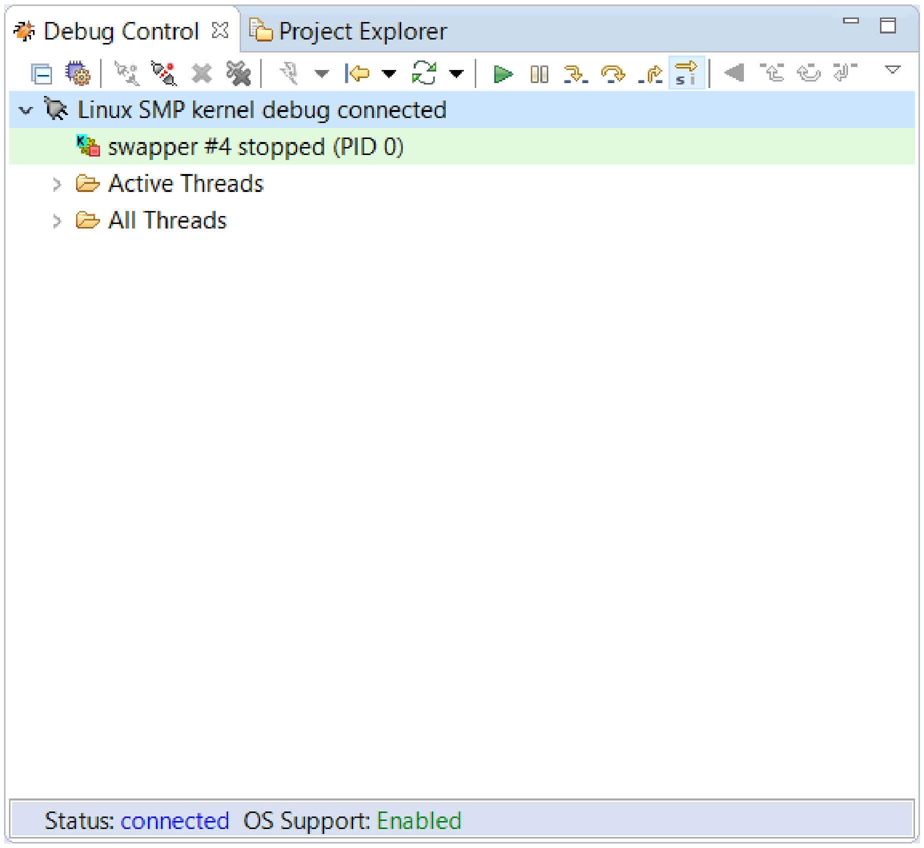


Linux SMP kernel debug connected
ARM_Cortex-A9MP_0 #0 stopped on breakpoint (SVC)
ARM_Cortex-A9MP_1 #1 stopped (SVC)
ARM_Cortex-A9MP_2 #2 stopped (SVC)
ARM_Cortex-A9MP_3 #3 stopped (SVC)

Status: connected OS Support: Enabled

Similarly, when viewing threads, you can view **Active Threads** and **All Threads**:

Figure 6-2: Active and All Threads



Linux SMP kernel debug connected
swapper #4 stopped (PID 0)
Active Threads
All Threads

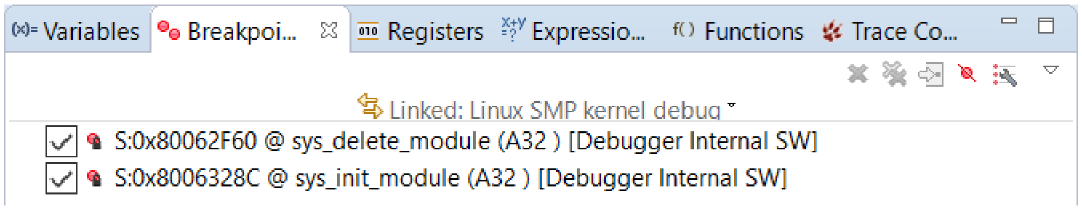
Status: connected OS Support: Enabled

- When OS awareness is enabled and kernel symbols are loaded from the `vmlinux` file, DS-5 Debugger tries to access some locations in the kernel. For example, it tries to read `init_nsproxy.uts_ns->name` to get the kernel name and version. It also tries to set breakpoints automatically on `sys_init_module()` and `sys_delete_module()` to trap when kernel modules are inserted (`insmod`) and removed (`rmmod`).

  The debugger handles internal breakpoints automatically, you cannot enable or disable them, or control them in any way. When internal breakpoints are hit, the debugger automatically performs the corresponding action, and then restarts program execution by itself.

  You can see these breakpoints appearing in the **Breakpoints** view.

Figure 6-3: Breakpoints view



S:0x80062F60 @ sys_delete_module (A32) [Debugger Internal SW]
S:0x8006328C @ sys_init_module (A32) [Debugger Internal SW]

4. To confirm the kernel symbols match the kernel image, enter `info os-version` in **Commands** view.

   Your operating system version should match the Linux kernel version. For example,`Operating system version: Linux 3.4.0-rc3-ve-v7-ds5 #1 SMP Mon Oct 31 00:17:24 GMT 2016 arm`.

   This is similar to entering `output init_nsproxy.uts_ns->name`, which should display an output similar to: `{sysname = "Linux", nodename = "(none)", release = "3.4.0-rc3-ve-v7-ds5", version = "#1 SMP Mon Oct 31 00:17:24 GMT 2016", machine = "arm", domainname = "(none)"}`.

   This might take a few moments to display because DS-5 Debugger has to process lots of debug symbols.

7. Multi-core features

So far, `CPU_0` performed all the tasks. The next steps explore using secondary cores for debugging.

While `CPU_0` is performing its tasks, the secondary cores are held in a standby state. Conceptually, you can think of the secondary cores being in a holding pen and being released one at a time, under the control of the primary core.

The secondary processors are released from their holding pen when the `platform_smp_prepare_cpus()` function is executed.

1. To observe when the function is executed, set a breakpoint on `platform_smp_prepare_cpus()` function.

   At the **Command** prompt, enter `thbreak platform_smp_prepare_cpus` to set a temporary hardware breakpoint.

2. Click **Continue** ▶ to run the code to reach the breakpoint.

   The last call in this function writes the address of secondary startup into the system-wide flags register. This write releases the secondary CPUs, which then branch to the address `(&versatile_secondary_startup = 0x804C7480C)`.

3. In the **Debug Control** view, notice that `Core_0` is in C code with its MMU on, and has started some threads, but `Core_1` is still in a holding pen (`WFE/LDR/CMP/BNE`) assembler loop with its MMU off.

   Set a breakpoint on the `versatile_secondary_startup()` function. Enter `thbreak versatile_secondary_startup`.

4. Click **Continue** ▶ to run to the breakpoint.

5. Expand `Core_1` in the [Debug Control] view. You can see that it has entered `versatile_secondary_startup()`.

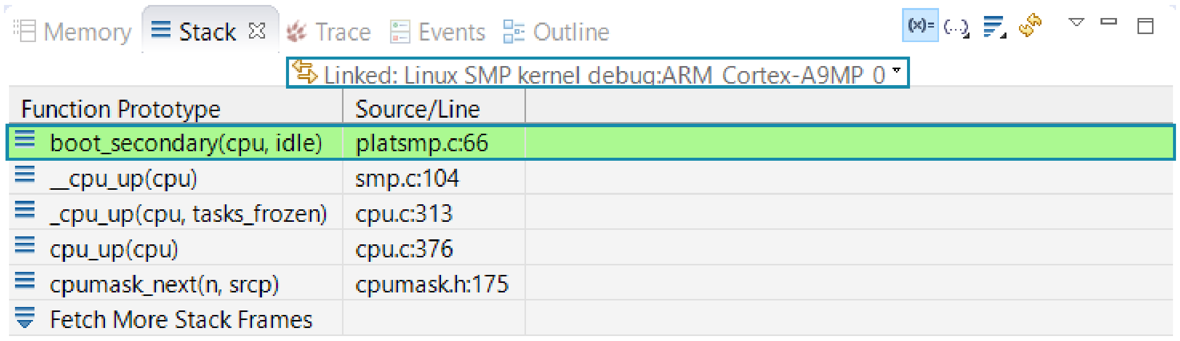
There are two levels of holding pens during an SMP kernel start-up. The first one holds all secondary cores until `Core_0` reaches a known good state. Once `Core_0` is in a known good state, the secondary cores are all released together. They then enter into a second-level holding pen. They are then released, one at a time by `Core_0`.

All cores have their MMUs enabled at this point, so you can explore this second-level holding pen using symbols with virtual addresses by setting breakpoints.

- Enter `thbreak boot_secondary` at the **Command** prompt to set a breakpoint on `boot_secondary` function. This is executed by `Core_0`.
- Enter `thbreak platform_secondary_init` at the **Command** prompt to set a breakpoint on the `platform_secondary_init` function. This is executed by `Core_1`.

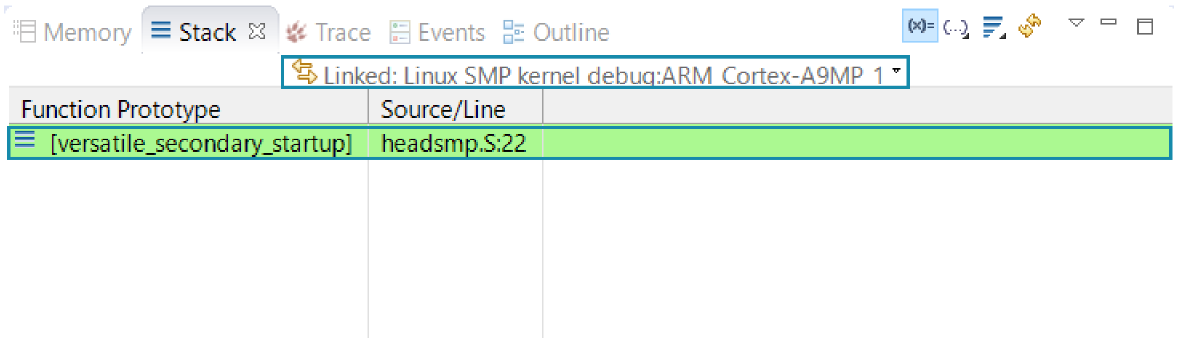
Run to each breakpoint. The `platsmp.c` file opens up in the **Editor**. As you hit each breakpoint, click on `Core_0` and switch to the **Stack** window, then `Core_1` and switch to the **Stack** window, to see functions under the respective cores.

Figure 7-1: `Core_0` stack window



Function Prototype	Source/Line
boot_secondary(cpu, idle)	platsmp.c:66
__cpu_up(cpu)	smp.c:104
_cpu_up(cpu, tasks_frozen)	cpu.c:313
cpu_up(cpu)	cpu.c:376
cpumask_next(n, srcp)	cpumask.h:175
Fetch More Stack Frames	

Figure 7-2: `Core_1` stack window



Function Prototype	Source/Line
[versatile_secondary_startup]	headsmp.S:22

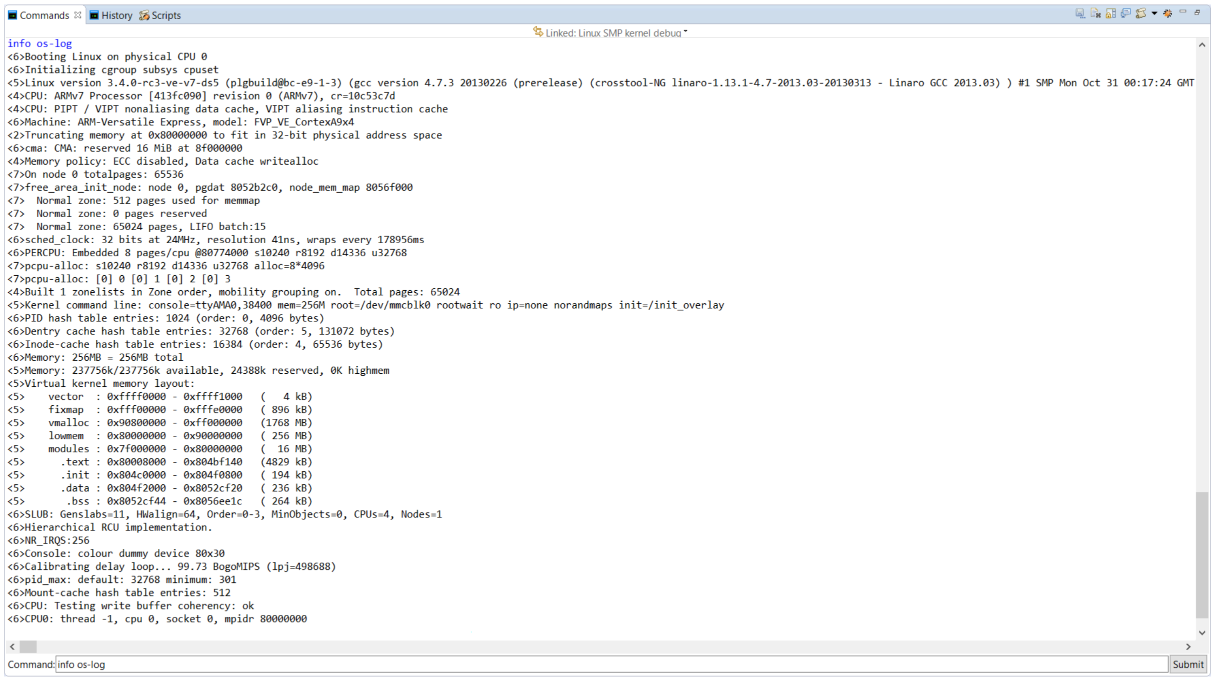
8. Logging features

It would be very useful to display early `printk` outputs in DS-5 Debugger's command window during kernel bring-up. Ideally, before the console is enabled, so no serial output is present.

For example, set a breakpoint on `console_init()` in `main.c`. Click **Continue** to run to the breakpoint. No log messages are output to the console because it is not enabled yet. But you can view the entire log for activities so far.

- Enter `info os-log` in the **Command** prompt to view the log.

Figure 8-1: `info os-log`



```
info os-log
<6>Booting Linux on physical CPU 0
<6>Initializing cgroup subsys cpuset
<5>Linux version 3.4.0-rc3-ve-v7-ds5 (plgbuild@bc-e9-1-3) (gcc version 4.7.3 20130226 (prerelease) (crosstool-NG linaro-1.13.1-4.7-2013.03-20130313 - Linaro GCC 2013.03) ) #1 SMP Mon Oct 31 00:17:24 GMT
<4>CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=10c53c7d
<4>CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
<6>Machine: ARM-Versatile Express, model: FVP_VE_CortexA9x4
<2>Truncating memory at 0x80000000 to fit in 32-bit physical address space
<6>cma: CMA: reserved 16 MiB at 8f000000
<4>Memory policy: ECC disabled, Data cache writealloc
<7>On node 0 totalpages: 65536
<7>free_area_init_node: node 0, pgdat 8052b2c0, node_mem_map 8056f000
<7>  Normal zone: 512 pages used for memmap
<7>  Normal zone: 0 pages reserved
<7>  Normal zone: 65024 pages, LIFO batch:15
<6>sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every 178956ms
<6>PERCPU: Embedded 8 pages/cpu @80774000 s10240 r8192 d14336 u32768
<7>pcpu-alloc: s10240 r8192 d14336 u32768 alloc=8*4096
<7>pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
<4>Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 65024
<5>Kernel command line: console=ttyAMA0,38400 mem=256M root=/dev/mmcblk0 rootwait ro ip=none norandmaps init=/init_overlay
<6>PID hash table entries: 1024 (order: 0, 4096 bytes)
<6>Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
<6>Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
<6>Memory: 256MB = 256MB total
<5>Memory: 237756k/237756k available, 24388k reserved, 0K highmem
<5>Virtual kernel memory layout:
<5>    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
<5>    fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)
<5>    vmalloc : 0x90800000 - 0xff000000   (1768 MB)
<5>    lowmem  : 0x80000000 - 0x90000000   ( 256 MB)
<5>    modules : 0x7f000000 - 0x80000000   (  16 MB)
<5>      .text : 0x80008000 - 0x804bf140   (4829 kB)
<5>      .init : 0x804c0000 - 0x804f0800   ( 194 kB)
<5>      .data : 0x804f2000 - 0x8052cf20   ( 236 kB)
<5>       .bss : 0x8052cf44 - 0x8056ee1c   ( 264 kB)
<6>SLUB: Genslabs=11, HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
<6>Hierarchical RCU implementation.
<6>NR_IRQS:256
<6>Console: colour dummy device 80x30
<6>Calibrating delay loop... 99.73 BogoMIPS (lpj=498688)
<6>pid_max: default: 32768 minimum: 301
<6>Mount-cache hash table entries: 512
<6>CPU: Testing write buffer coherency: ok
<6>CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
```

```
Command: info os-log
```

- If you want to view the log output line by line as the code is processed, enter `set os log-capture` on at the **Command** prompt.

9. Single-stepping individual cores or threads

You can single step through a single core or thread/process. Select either the core or the thread/process in the **Debug Control** view, then press **F5** on your keyboard.

When you single step through a process, it might be migrated to another core. If a breakpoint is set on a process, the debugger can track the migration of process-specific breakpoints to the other core.

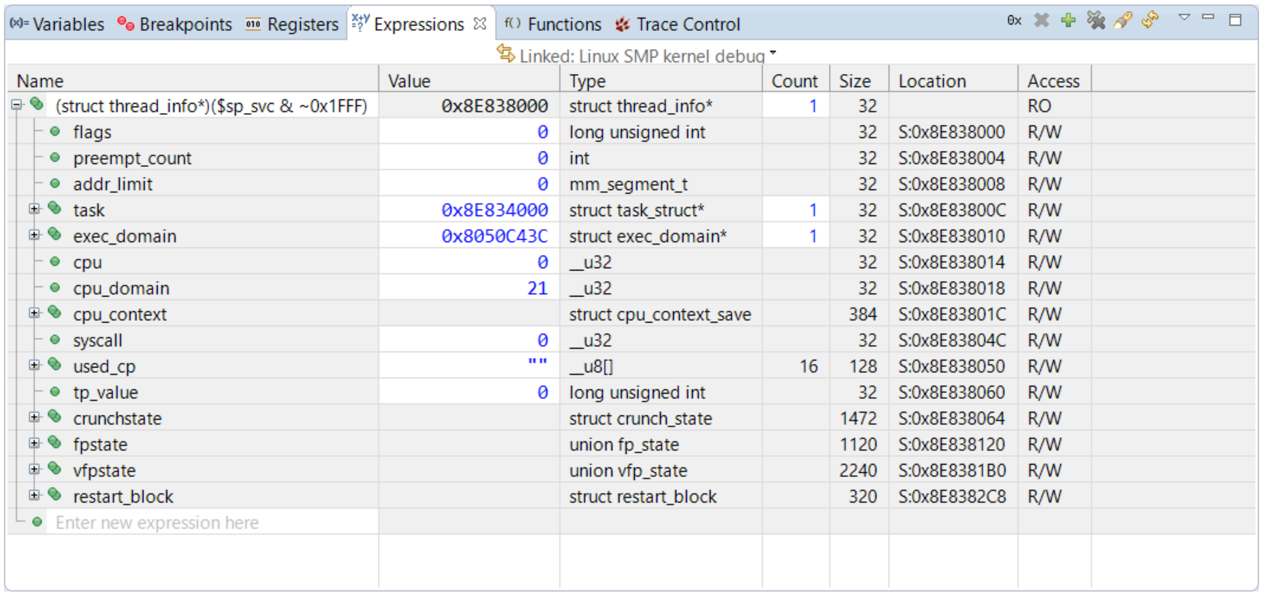
10. Using expressions to view data

You can use the **Expressions** view to create and work with expressions in DS-5. For example, to view the kernel's `thread_info` structure, in the **Expressions** view, enter:

```
(struct thread_info*)($sp_svc & ~0x1FFF).
```

When the view refreshes, expand the tree structure to explore its contents.

Figure 10-1: Expressions view



Name	Value	Type	Count	Size	Location	Access
(struct thread_info*)($sp_svc & ~0x1FFF)	0x8E838000	struct thread_info*	1	32		RO
flags	0	long unsigned int		32	S:0x8E838000	R/W
preempt_count	0	int		32	S:0x8E838004	R/W
addr_limit	0	mm_segment_t		32	S:0x8E838008	R/W
task	0x8E834000	struct task_struct*	1	32	S:0x8E83800C	R/W
exec_domain	0x8050C43C	struct exec_domain*	1	32	S:0x8E838010	R/W
cpu	0	__u32		32	S:0x8E838014	R/W
cpu_domain	21	__u32		32	S:0x8E838018	R/W
cpu_context		struct cpu_context_save		384	S:0x8E83801C	R/W
syscall	0	__u32		32	S:0x8E83804C	R/W
used_cp	""	__u8[]	16	128	S:0x8E838050	R/W
tp_value	0	long unsigned int		32	S:0x8E838060	R/W
crunchstate		struct crunch_state		1472	S:0x8E838064	R/W
fpstate		union fp_state		1120	S:0x8E838120	R/W
vfpstate		union vfp_state		2240	S:0x8E8381B0	R/W
restart_block		struct restart_block		320	S:0x8E8382C8	R/W
Enter new expression here						

11. Using the command line to view cores, thread, and process information

As you have seen, the **Debug Control** view is very useful to display details of individual cores, threads, and processes. You can also view the same information in the **Commands** view, by using individual commands.

- info cores - View information about the running processors.
- info threads - View information about the running threads.
- info processes - View information about the running processes.