



FTF | FREESCALE
TECHNOLOGY
FORUM 2015

Hands-On Workshop: Debug **Linux**[®] **Kernel** and **Linux Application** **Problems**

FTF-DES-F1203

Robert McGowan | Chief Architect
Catalin Udma
Razvan Ionescu

JUNE . 2015



External Use

Freescale, the Freescale logo, AllWin, C-S, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetic, MagniV, motorGT, PEG, PowerQUICC, Prosecc Expert, QorIQ, QorIQ Qonverge, Qorivos, ReadyPilot, SafeAssure, the SafeAssure logo, StarCore, Synchrify, Vortiga, Vybrid and Xilinx are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. AirMax, iSeek, iSeeStack, CoreNet, Flexis, LayerStack, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink and UMEMS are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © 2015 Freescale Semiconductor, Inc.



Layerscape LS2085A Software and Tools Enablement

CodeWarrior
Development Studio

**Software
Development Tools**



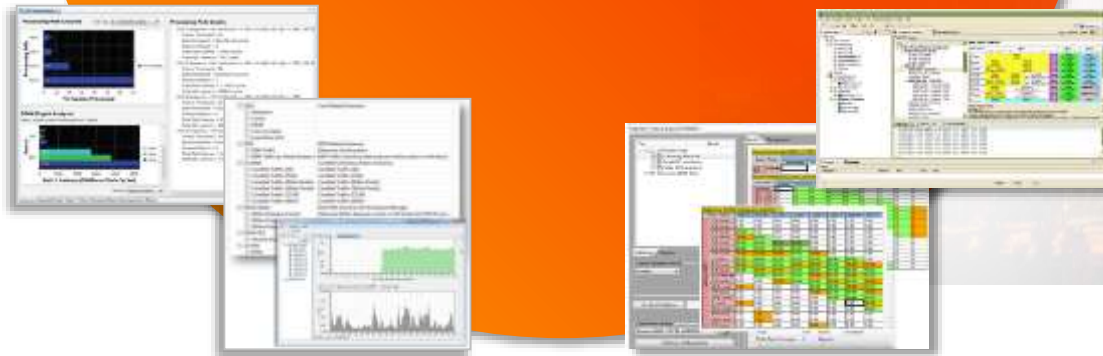
**QorIQ
Linux SDK**



**JTAG Run Control
and Trace Probes**

**Performance
Analysis
and Trace Tools**

**QorIQ SoC Platform
Configuration Tools**



Training and Hands-on Goals

- The following material has been developed so you ...
 - ... become familiar with the debugging U-boot and the Linux kernel
 - ... Learn about utilizing trace to aid in debug
 - ... Debug Linux applications
 - ...Trace a single Linux application

Agenda

- Lecture - Introduction / Overview
- Lecture - CodeWarrior
- Lecture - Board/Device Overview
- Activity - Create Bareboard Project
- Activity - CW Connection
- Lecture - SDK Prebuilt U-boot Image
- Lecture - U-Boot Debug
- Activity - U-Boot Debug
- Lecture - U-Boot Trace



Agenda

- Lecture - Linux Kernel Debug
- Activity - Linux Kernel Debug
- Lecture - Kernel Tracing
- Activity - Trace Compass
- Lecture - Linux Application Debug
- Activity - Linux Application Debug
- Activity - Application Trace
- Activity - Debug Print
- Lecture - Summary / Q/A



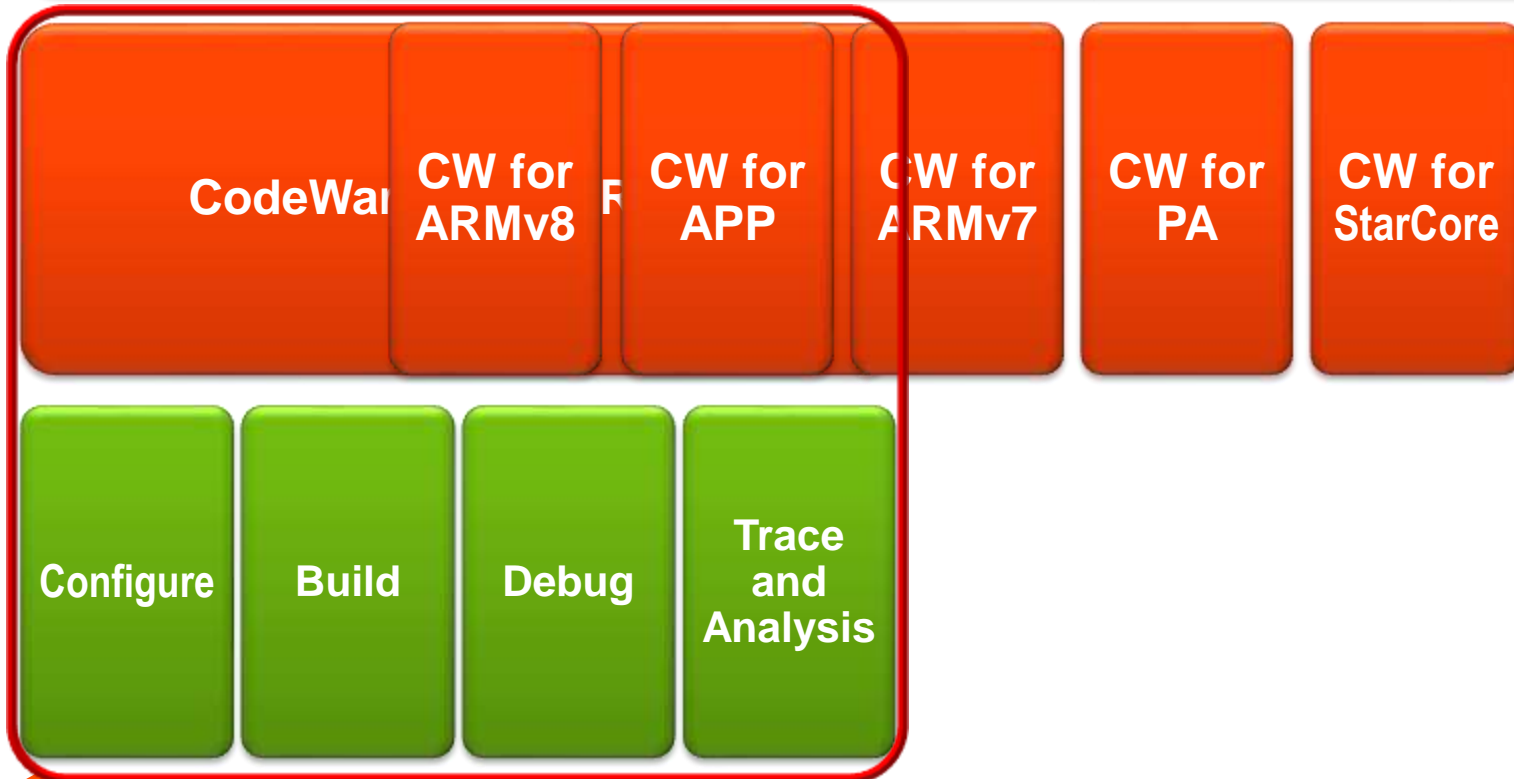
Layerscape Tools

CodeWarrior for ARMv8 ISA
CW-TAP



CodeWarrior Family

QorIQ Tools



CodeWarrior Family

QorIQ Tools

CodeWarrior for ARMv8

CW for
APP

CW for
ARMv7

CW for
PA

CW for
StarCore

Configure

Build

Debug

Trace
and
Analysis



CodeWarrior Development Studio

A Complete Development Environment Under Eclipse



- **Eclipse IDE**

- Configuration Wizards
- Plug-In Architecture
- 3rd party community



- **Build Tools**

- C/C++ Compiler

- **Initialization Tools**

- SOC platform initialization and configuration



- **Run Control**

- CW-TAP



- **Debugger**

- Multicore aware
- Cross-triggering
 - Run/Stop of targets simultaneously
- Access to all on-chip resources
- Linux awareness

- **Software Analysis - Trace & Profile**

- Leverages chip capabilities
 - Profiling Unit
 - In system trace buffering
- Trace / Code / Performance Viewer
- Offline trace visibility



CodeWarrior Aids Debug Through Multiple Phases

- SoC and board bring-up
 - Single-core and multi-core (AMP) bare-metal debugger
 - Device introspection: core and SoC registers, memory
 - U-boot
- Linux OS development
 - SMP aware kernel debug
 - Device driver development and debug
 - Aligned with Freescale SDK & Linaro GNU toolchain

CodeWarrior Aids Debug Through Multiple Phases (2)

- Linux application development
 - GNU debugger compatible + extensions for Linux application debug
 - Linux target information: System Browser Linux kernel module development and debug
 - Aligned with Freescale SDK & Linaro GNU toolchain
 - Target debug agent
- Performance Analysis
 - Core performance metrics & scenarios
 - SOC performance metrics & scenarios
 - Profiling from trace

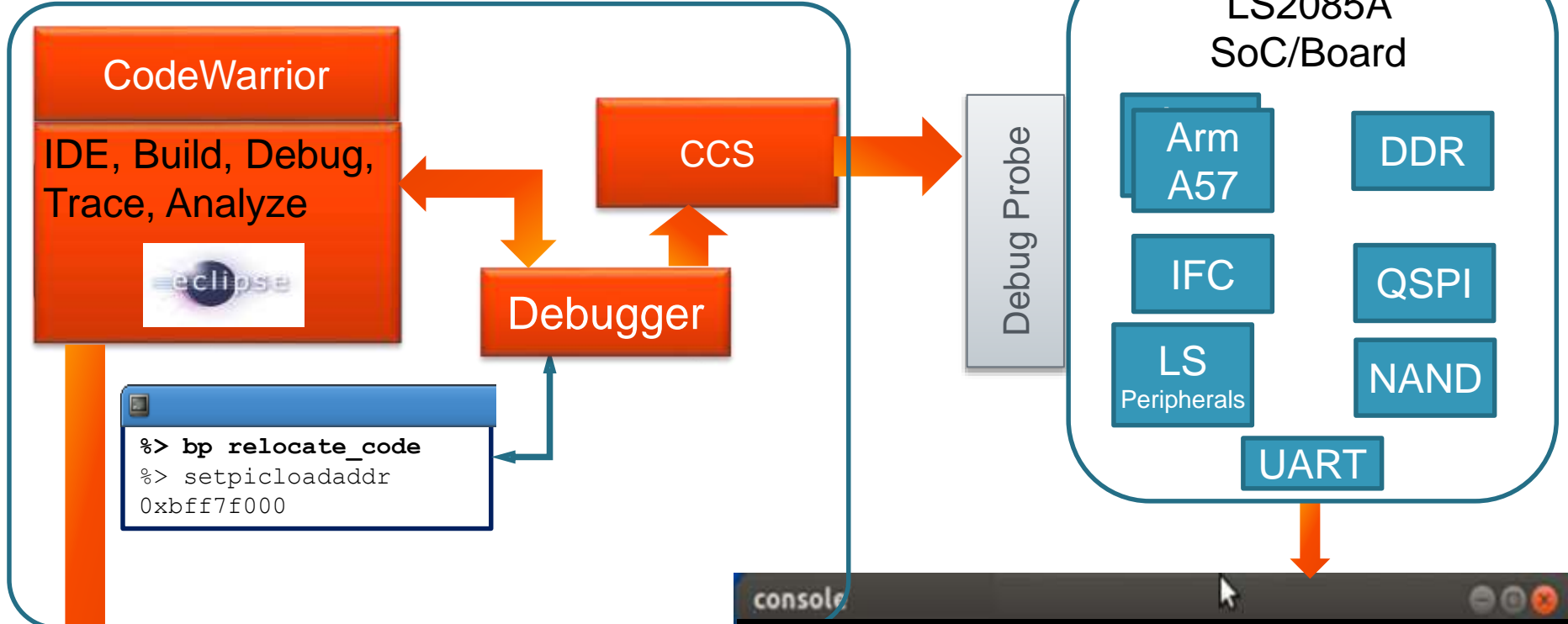
CodeWarrior Aids Debug Through Multiple Phases (3)

- Non-intrusive debug through trace
 - Core and SoC trace sources: configuration, extraction, visibility
 - Post-mortem debugging: offline trace
 - Debug-print
 - Linux aware trace
 - Linux kernel trace
 - Code Coverage

Modes of Debug

- Debug over JTAG
- Debug over Ethernet

CodeWarrior : U-boot debug



```
%> bp relocate_code
%> setpicloadaddr
0xbff7f000
```

- CW project: import u-boot elf
- Run Control: run/suspend/step
- Breakpoints
- Registers View: GPR + SoC registers

```
console
U-Boot 2014.01-gb330cec (Apr 18 2014 - 17:38:22)
CPU: Freescale LayerScape LS2085A, Version: 1.0,
(0x87080310)
Clock Configuration:
CPU0(ARMv8):800 MHz,
Bus:400 MHz, DDR:400 MHz,
Board: LS2085AQDS
I2C: ready
relocate start
```

CodeWarrior JTAG Probe

- CodeWarrior TAP
 - Provides connection between CodeWarrior and target device
 - Target run control
 - Serial port pass through
 - Interface to host
 - Locally over USB
 - Remotely over Ethernet
 - Buy separately (below \$500)
- Reminder : no USB-TAP support



CodeWarrior TAP

- JTAG debugging and CodeWarrior run control requires :

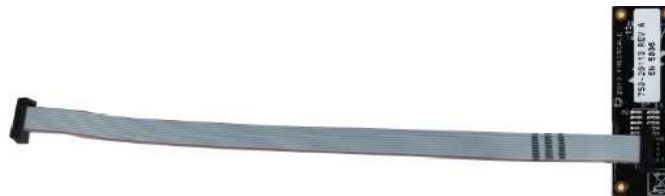
Base Unit : (Part # [CWH-CTP-BASE-HE](#))

www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW_TAP

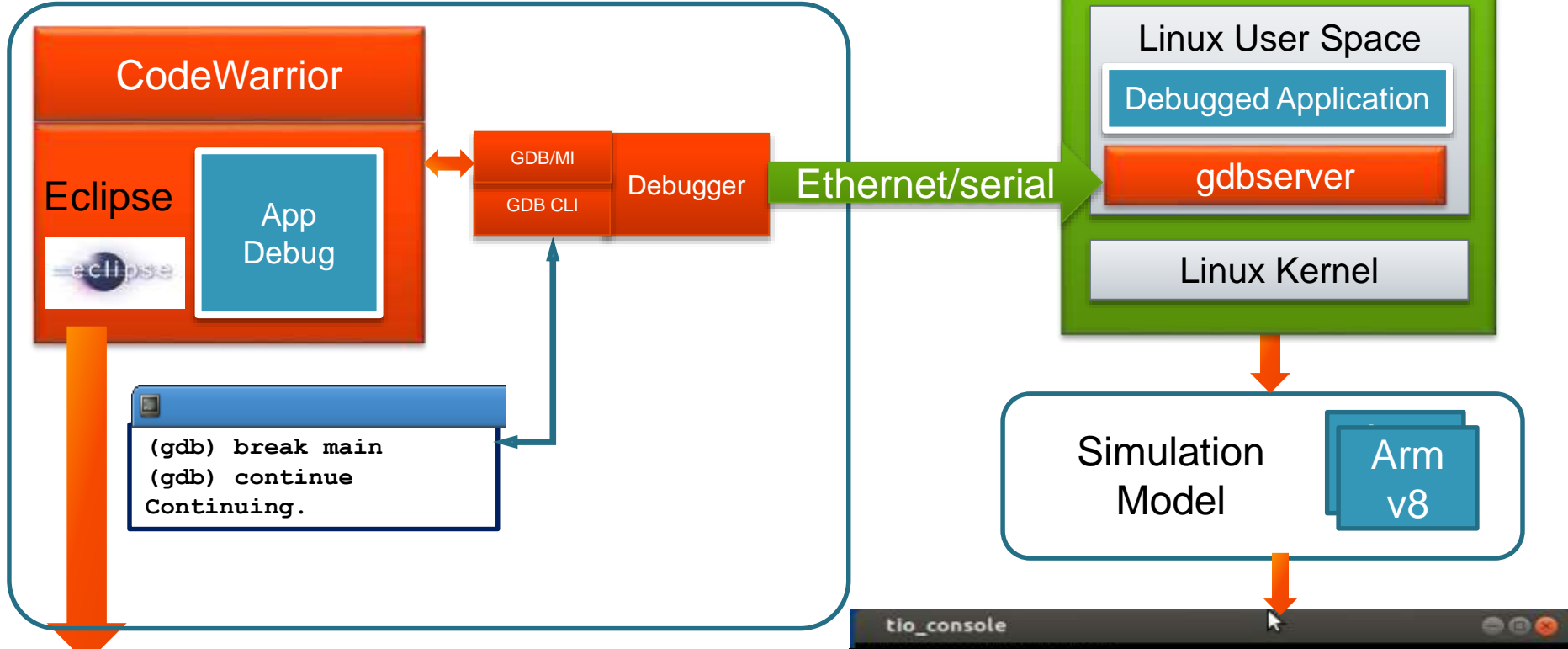


... and *Probe Tip* : (Part # [CWH-CTP-CTX10-YE](#))

www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CWH-CTP-CTX10-YE



Linux Application Debug



CodeWarrior

Eclipse

App
Debug

GDB/MI
GDB CLI

Debugger

Ethernet/serial

Remote Linux System

Linux User Space

Debugged Application

gdbserver

Linux Kernel

Simulation
Model

Arm
v8

- CW project: project wizard
- Eclipse DSF – automatic download and launch application
- Run Control: run/suspend/step
- Breakpoints
- Registers View: GPR registers

```
tio_console
root@freescale $ uname -a
Linux freescale 3.12.0+ #1 Wed Feb 26
09:45:41 IST 2014 aarch64 GNU/Linux
root@freescale $
root@freescale $
root@freescale $ ./myLinuxApplication
running Linux Application
```



Introducing the LS2085A RDB



LS2085A RDB Top View

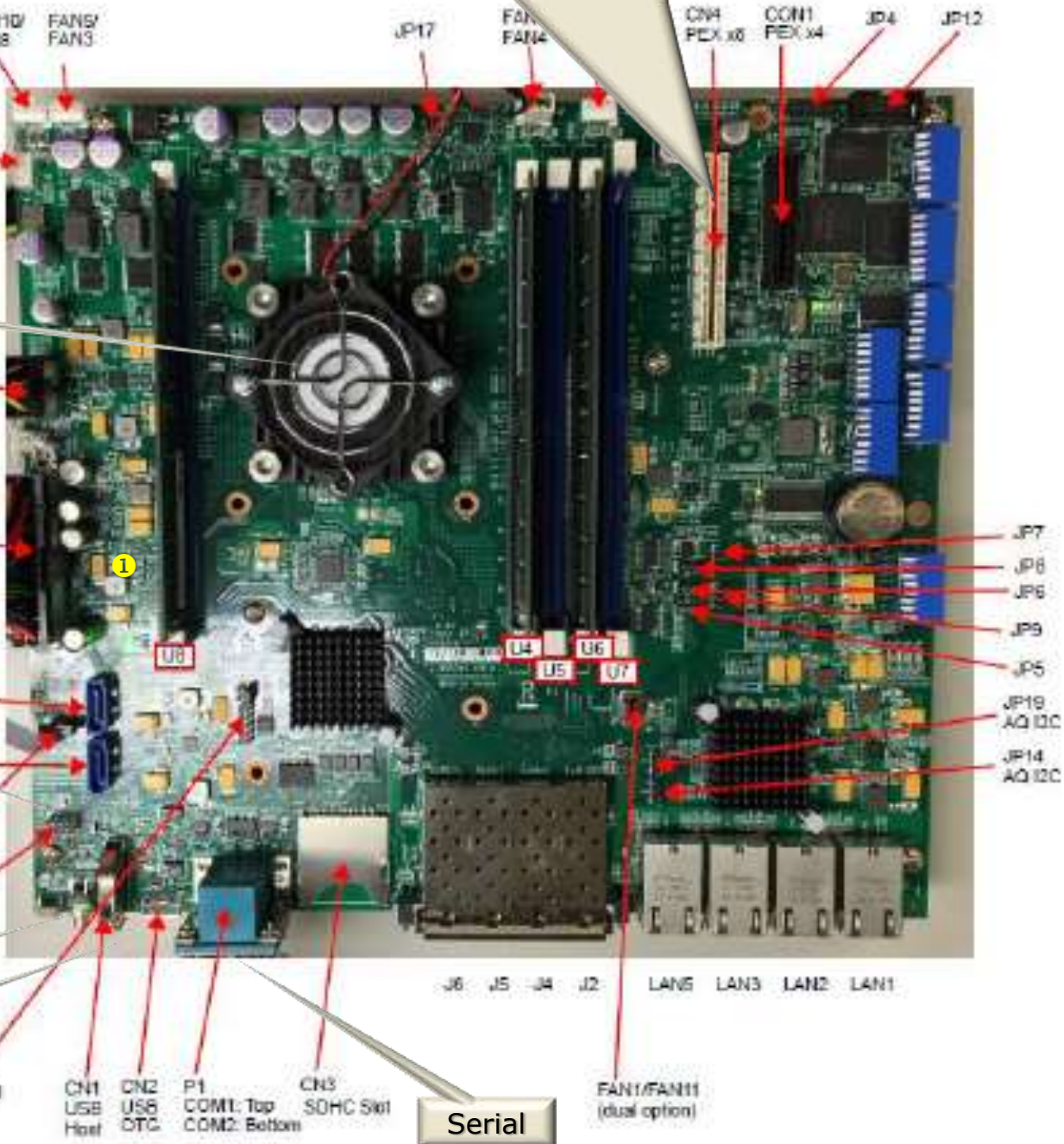
Aux Ethernet is plugged into PCIe

JTAG

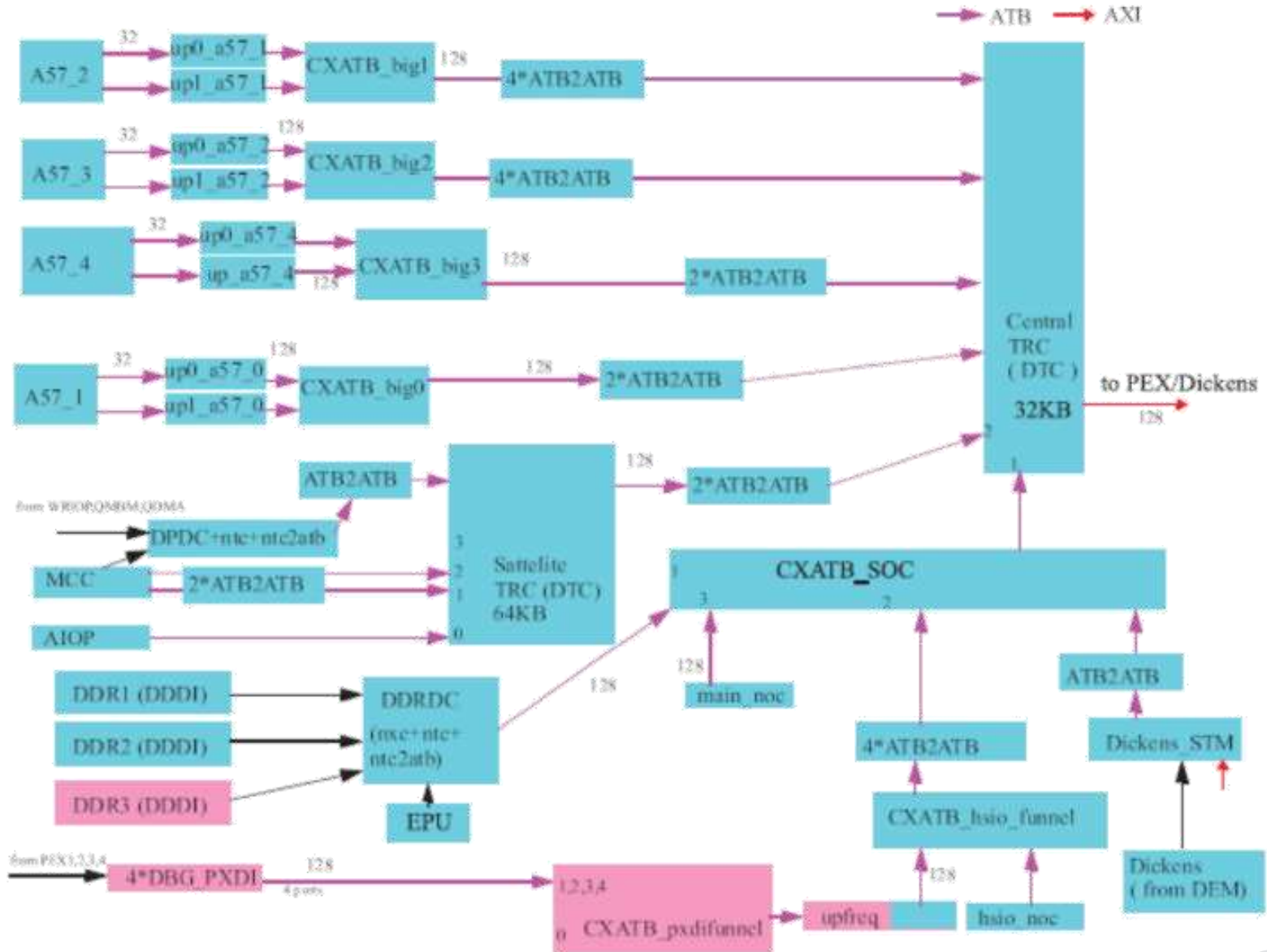
RESET

Power

Serial



QorIQ LS2085A Debug Block Diagram



Debug Features

- Run-Control debug features in cores
 - Cross-triggering between cores
- Trace
 - Program trace (ETM)
 - System trace (STM)
 - Stored in internal memory or DDR
 - No external export via TPIU or Aurora
- EPU Performance Monitor

Preparing the Environment

What has been done for you



Connections



PCIe Ethernet Card is here,
1G/100Mbit copper.

100-240 VAC, 47-63 Hz

Aux Ethernet

Cross-over
Ethernet

JTAG (take off lid)



Use this serial port for the
console, 115200-8-1-N.

Connect all of these Ethernet ports to the same
port on the other RDB. All are 10G only.

USB<->Serial



Items That Have Been Setup For You

- Host OS
 - Best to use Linux on the host when developing Linux on the target
 - Multiple Linux OS supported
 - 64-bit Linux required
 - [Used Mint 17.1 for class](#)
- CodeWarrior for Networked Applications v2015.05
 - CodeWarrior for Layerscape ARMv8 ISA
- Freescale Linux SDK for LS2085A RDB
 - Installed from ISOs – could also obtain from GIT
 - Layerscape2-SDK-AARCH64-IMAGE-20150515-yocto
 - Layerscape2-SDK-SOURCE-20150515-yocto
 - Did not use CACHE
 - Added Freescale extensions for tracing support

Items That Have Been Setup For You

- Install on host
 - Yocto
 - Minicom / cutecom
 - 115200-8-N-1
 - Tftp server (not used in class)
 - telnet / putty (not used in class)
- Read RDB Quickstart Guide!
- Bitbake the SDK
- Install on target
 - Flash U-boot

Class Information

- Linux Login
 - User: class
 - Password: codewarrior
- SDK is installed in ~/SDK
 - Need to use full path in tool: /home/class/SDK
- On desktop
 - Launcher to Codewarrior – looks like rocket
 - shortcut to cutecom
 - Menu has link to terminal
 - Use for launch minicom
- No password on target Linux

RDB-LS2085A

SDK EAR4.0 Prebuilt u-boot Image



U-Boot Startup Messages

- Reset the RDB-LS2085A, interrupt the countdown
- Review the u-boot output in the console window :

```
U-Boot 2014.01Layerscape-SDK-V1.3+g50d6848 (Oct 24 2014 - 19:22:06)
CPU: Freescale LayerScape LS1021E, Version: 1.0, (0x87081110)
Clock Configuration:
  CPU0(ARMv8):1000 MHz,
  Bus:300 MHz, DDR:800 MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
  00000000: 0608000a 00000000 00000000 00000000
  00000010: 20000000 00407900 e0025a00 21046000
  00000020: 00000000 00000000 00000000 00038000
  00000030: 00000000 881b7340 00000000 00000000
Board: LS2085ARDB
CPLD: V2.3
PCBA: V3.0
VBank: 0
I2C: ready
DRAM: 1 GiB (DDR3, 32-bit, CL=10.5, ECC off)
Using SERDES1 Protocol: 32 (0x20)
```

U-Boot Startup Messages

```
Flash: 128 MiB
MMC:   FSL_SDHC: 0
EEPROM: NXID v1
Firmware 'Microcode version 0.0.0 for T1040 r1.0' for 1040 V1.0
QE: uploading microcode 'Microcode for T1040 r1.0'
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0300 1 slots 1 ports ? Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc
scanning bus for devices...
Found 0 device(s).
Net:   eTSEC1 is in sgmii mode.
eTSEC2 is in sgmii mode.
eTSEC1 [PRIME], eTSEC2, eTSEC3
Hit any key to stop autoboot: 0
```

U-Boot - Network Setup

- MAC address environment variables ethaddr, eth1addr and eth2addr are set by values stored in EEPROM.

To manually change the EEPROM MAC addresses :

```
=> mac id
=> mac 0 00:e0:0c:bc:e5:60
=> mac 1 00:e0:0c:bc:e5:61
=> mac 2 00:e0:0c:bc:e5:62
=> mac save
Programming passed.
```

- Establishing network connectivity in your network environment :

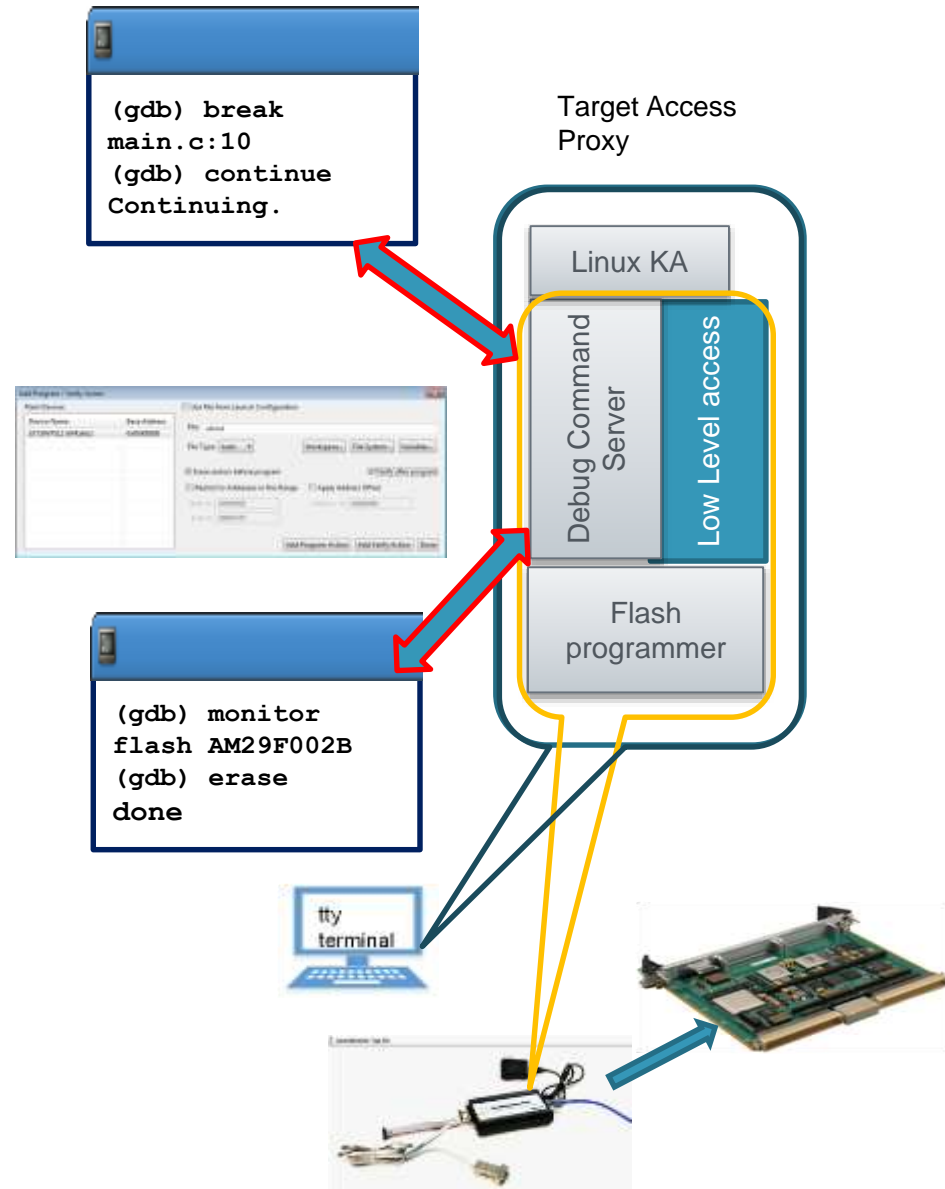
```
=> setenv ipaddr <RDB-LS2085A IP>      → IP address of RDB-LS2085A
=> setenv serverip <host IP>           → IP address of TFTP server
=> setenv ethaddr '00:e0:0c:bc:e5:60'
=> ping $serverip
Speed: 1000, full duplex
Using eTSEC3 device
host 192.168.1.101 is alive
=> saveenv                             → preserves envvars across boots
```

Bare-Metal Debug



Bare-Metal Debug

- Target interface to real hardware / simulator
- Lightweight debugger engine accessible from both GUI and command line
- Compatible with the GNU debugger front-end
- Standard set of memory/register access commands + monitor extensions
- Simultaneous connectivity with multiple clients
- Single- and Multi-core support



Activity

Create a CodeWarrior Bareboard Project



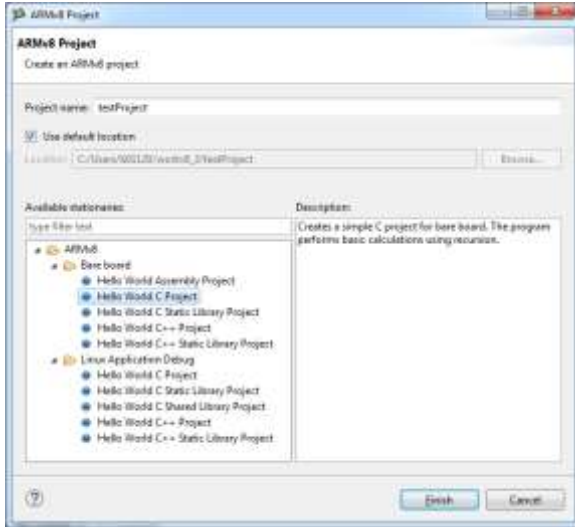
Activity - CodeWarrior Bareboard Debugging

Bareboard Debugging with *CodeWarrior TAP* Connection



- Activity Summary:

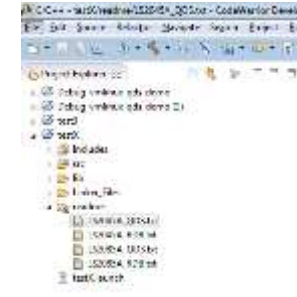
1. Launch CodeWarrior-ARMv8 from the desktop
2. Go to Workbench
3. *File* → *New* → *ARMv8 Stationery*
4. Enter a *Project Name*
5. Choose :
 - Bare board project type (ASM, C, C++)
6. Press *Finish*



Create bareboard project – Activity (cont.)

The project created can be used on any LS2 SoC & board supported:

- For any information about board hardware setup or project options please check the readme files attached in the project under section **readme**.
- You will find out about:
 - **Boot** options switch configurations
 - Default **RCW**
 - **Memory Map** with accessible areas
 - **SMP** debug configuration
 - **Console I/O** or **UART I/O** selection
 - Other useful info



```
LS2085A_QD1st.c
# Memory map and initialization
-----
0x0000_0000_0000 - 0x0000_0000_FFFF IPE CCSA - Boot ROM
0x0000_0100_0000 - 0x0000_0FFF_FFFF 240MB CCSA
0x0000_3000_0000 - 0x0000_3FFF_FFFF 120MB IFC NOR
0x0000_0000_0000 - 0x0000_FFFF_FFFF 2048MB GPP SRAM Region #1 (2GB)
0x0005_2000_0000 - 0x0005_2000_FFFF 64KB QD1S
0x0007_0000_0000 - 0x0007_3FFF_FFFF 180MB DCSA
0x000B_0000_0000 - 0x000B_FFFF_FFFF 6144MB GPP SRAM Region #2 (5GB)
-----
# NOR Flash
-----
Please consult the Flash Programmer Release Notes for more details on flash programming.

The NOR flash address range on LS2085A QD1 is 0x30000000 - 0x37FFFFFF.
Each flash sector is 128KB and there are 1624 sectors for a total of 128MB of NOR flash space.
The flash space is further divided into 8 virtual banks, by using CF6_LBMAP.
-----
# CodeWarrior Stationary Project
-----
The ARMv8 Stationary C Project can be configured to generate a baremetal application suitable for SMP or SMP mode. Please consult the "#define SMP" in start.S for more information.

The default project configuration is suitable for both single-core (running on "core0") and multicore SMP ("Use all cores") applications.

SMP implies shared memory model. All the cores execute the same application image, the memory is shared between cores, only the stack is established in different ranges for each core. The SMP support is limited by the lack of multicore-enabled runtime available in the baremetal toolchain. There is no support for SMP in the C++ stationary project.

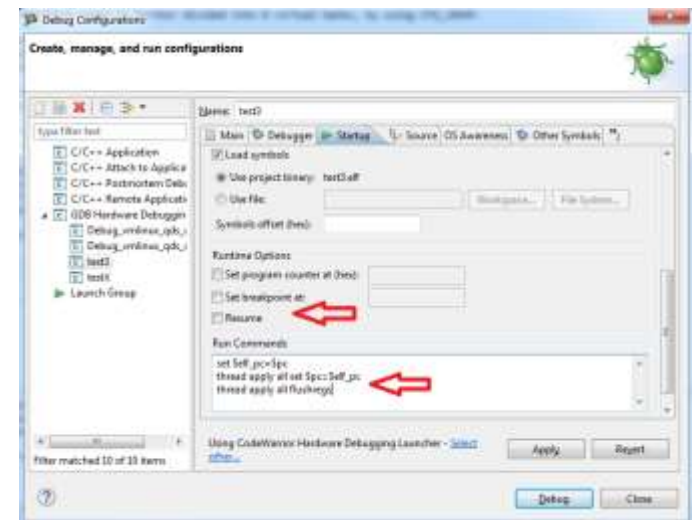
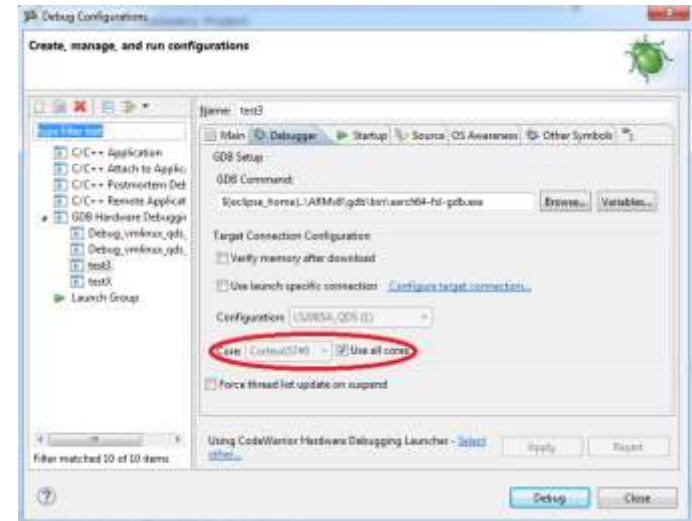
Please follow the below steps in order to debug the example application in SMP mode:
- Open the corresponding Debug Configuration
```



Create bareboard project – Activity (cont.)

Activating **SMP**:

- Go to Debug Configuration
- Select *Debugger* tab
- Make sure "**Use all cores**" is checked
- Go to *Startup* tab
- Uncheck "**Set breakpoint at**" and "**Resume**" in the *Runtime Options* section
- Enter the following commands in the *Run Commands* section:
 - `set $self_pc=$pc`
 - `thread apply all set $pc=$self_pc`
 - `thread apply all flushregs`

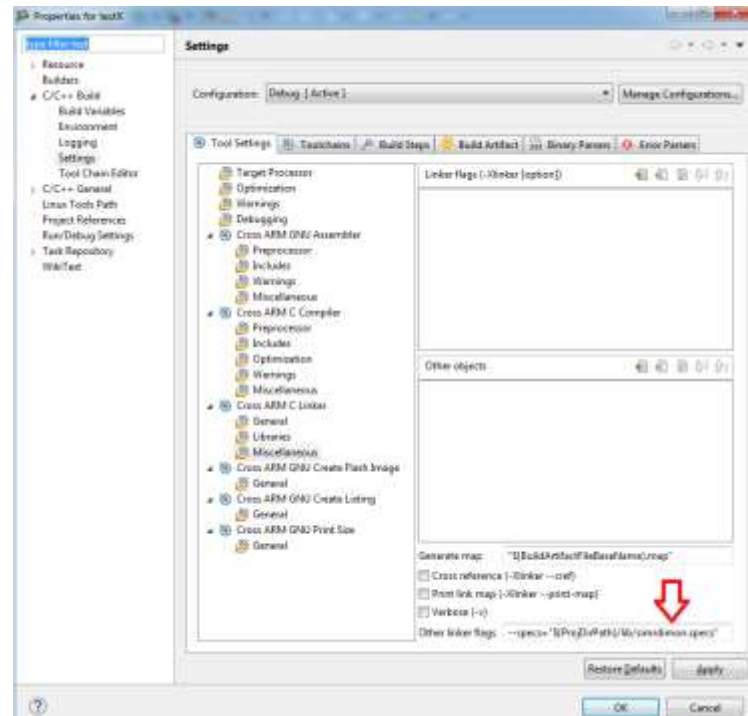


Create bareboard project – Activity (cont.)

The default **I/O mode** is debugger console (in other words the *simrdimon* library is being used). You can change to **UART** I/O if needed by modifying the project build settings: navigate to

C/C++ Build -> Settings -> Cross ARM C (or C++) Linker -> Miscellaneous

- In *Other linker flags* section
- Replace *simrdimon.specs* with *uart.specs*
- Re-build the project



Activity

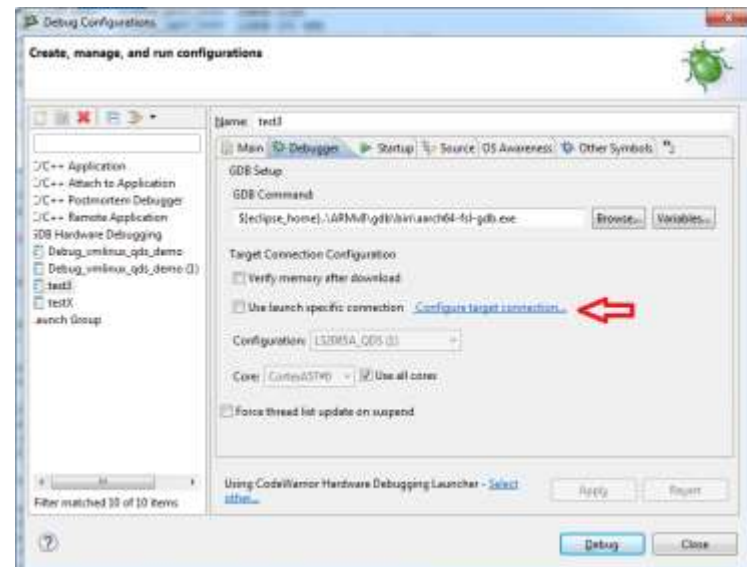
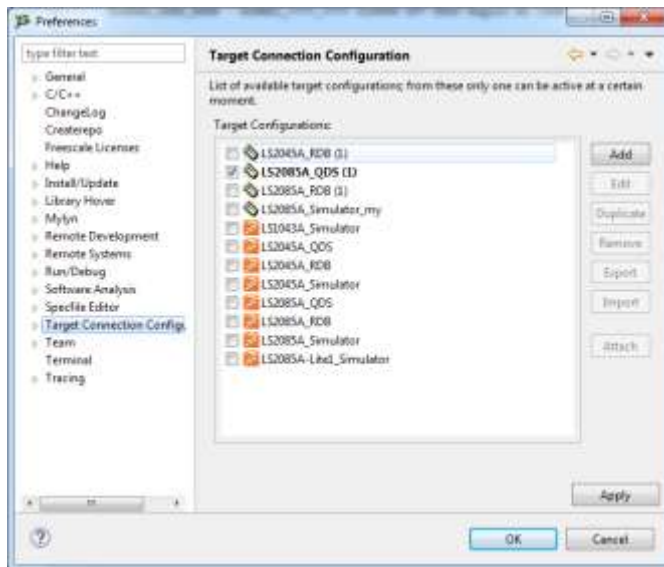
Establish a CodeWarrior Connection to the Target



Hardware Selection and Configuration Activity

Target Connection Configurator (TCC)

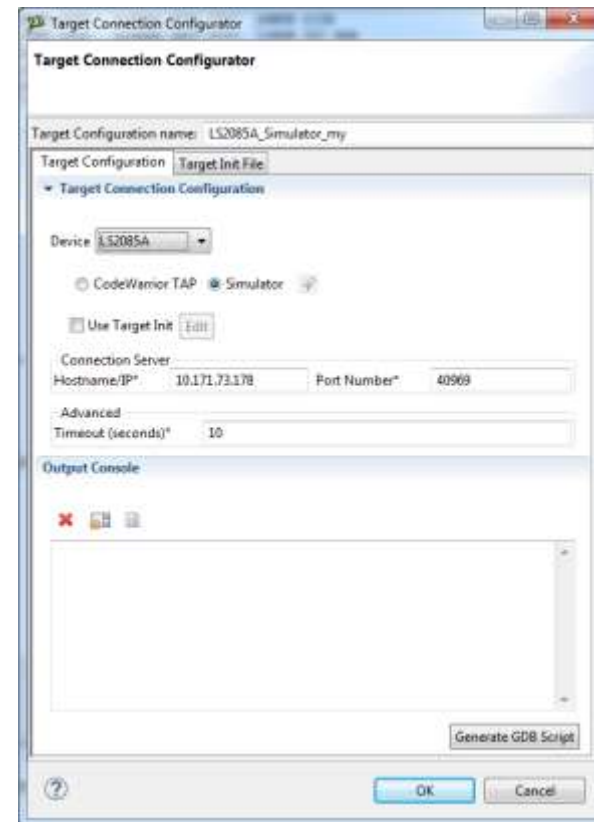
- For connection settings, target SoC and board selection go to:
- *Window -> Preferences -> Target connection configurator* or
- *Debug Configurations -> Debugger -> Configure target connection...*



Hardware Selection and Configuration Activity

Target Connection Configurator (TCC) (cont.)

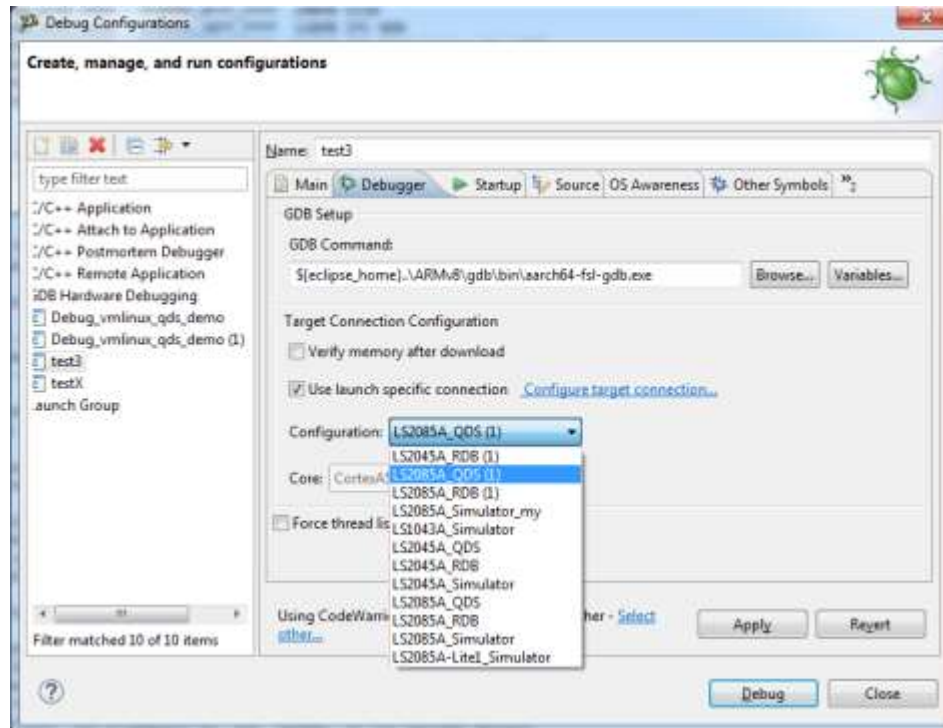
- To select the SoC & Board configuration simply check one of the pre-defined templates (📄) or select one and click *Duplicate* or *Edit* to create your own configuration (🔧)
- You can change:
 - **device** type
 - **connection** settings
 - **debug probe** type
- Or tweak the **initialization file**



Hardware Selection and Configuration Activity

Target Connection Configurator (TCC) (cont.)

- Selected configuration in TCC view will be the default one for all projects (launches), except if *Use launch specific connect* is checked.
- In this case, any existing configuration can be selected:



Activity

Flashing u-boot image to the Target with CW Flash



Flashing u-boot image to the Target Activity (5)

Using CW Flash command line interface

- Edit `CW_ARMv8\ARMv8\gdb_extensions\flash\cwflash.py` with your board and connection settings

```
#####
##### Copyright (C) 2015, Freescale Semiconductor, Inc.
##### All Rights Reserved
##### Version 1.0
#####

##### Parameters
#####

# Board type (Supported values: "QDS", "RDB").
BOARD_TYPE = "RDB"

# Flash types (Supported values: "nor", "nand").
FLASH_TYPE = "nor"

# CWTAP connection. If empty, it assumes that CWTAP uses an USB connection.
# For Ethernet connection please set the IP address (ex.: CWTAP_CONN = "192.168.0.1").
CWTAP_CONN = "10.171.74.169"

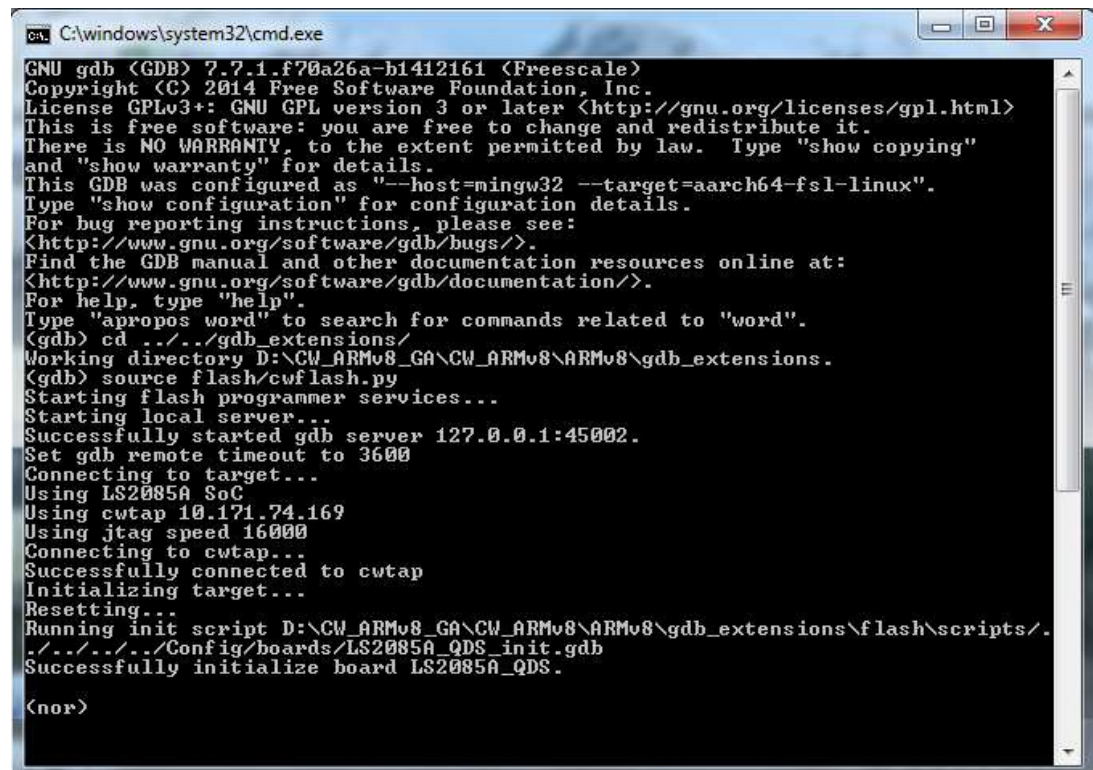
# Current SoC name.
SOC_NAME = "LS2085A"

# JTAG speed.
JTAG_SPEED = 16000
```

Flashing u-boot image to the Target Activity (5)

Using CW Flash command line interface (cont.)

- Start GDB console from `CW_ARMv8/ARMv8/gdb/bin/aarch64-fsl-gdb.bat`
- `cd ../../gdb_extensions`
- `source flash/cwflash.py`
- It will:
 - connect to the target
 - run initialization
 - select flash device



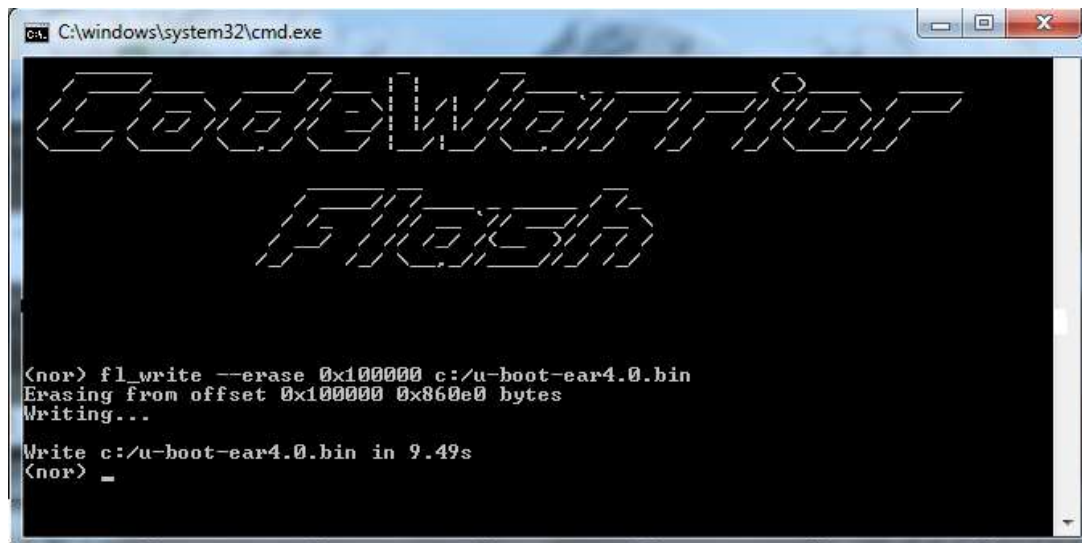
```
C:\windows\system32\cmd.exe
GNU gdb (GDB) 7.7.1.f70a26a-b1412161 (Freescale)
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=mingw32 --target=aarch64-fsl-linux".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) cd ../../gdb_extensions/
Working directory D:\CW_ARMv8_GA\CW_ARMv8\ARMv8\gdb_extensions.
(gdb) source flash/cwflash.py
Starting flash programmer services...
Starting local server...
Successfully started gdb server 127.0.0.1:45002.
Set gdb remote timeout to 3600
Connecting to target...
Using LS2085A SoC
Using cwtap 10.171.74.169
Using jtag speed 16000
Connecting to cwtap...
Successfully connected to cwtap
Initializing target...
Resetting...
Running init script D:\CW_ARMv8_GA\CW_ARMv8\ARMv8\gdb_extensions\flash\scripts\
../../../../../Config/boards/LS2085A_QDS_init.gdb
Successfully initialize board LS2085A_QDS.

<nor>
```

Flashing u-boot image to the Target Activity (5)

Using CW Flash command line interface (cont.)

- Issue following command:
 - `fl_write --erase 0x100000 {u-boot_image_path}`
- Wait a few seconds for the confirmation message
- You are ready to debug **u-boot**



```
C:\windows\system32\cmd.exe

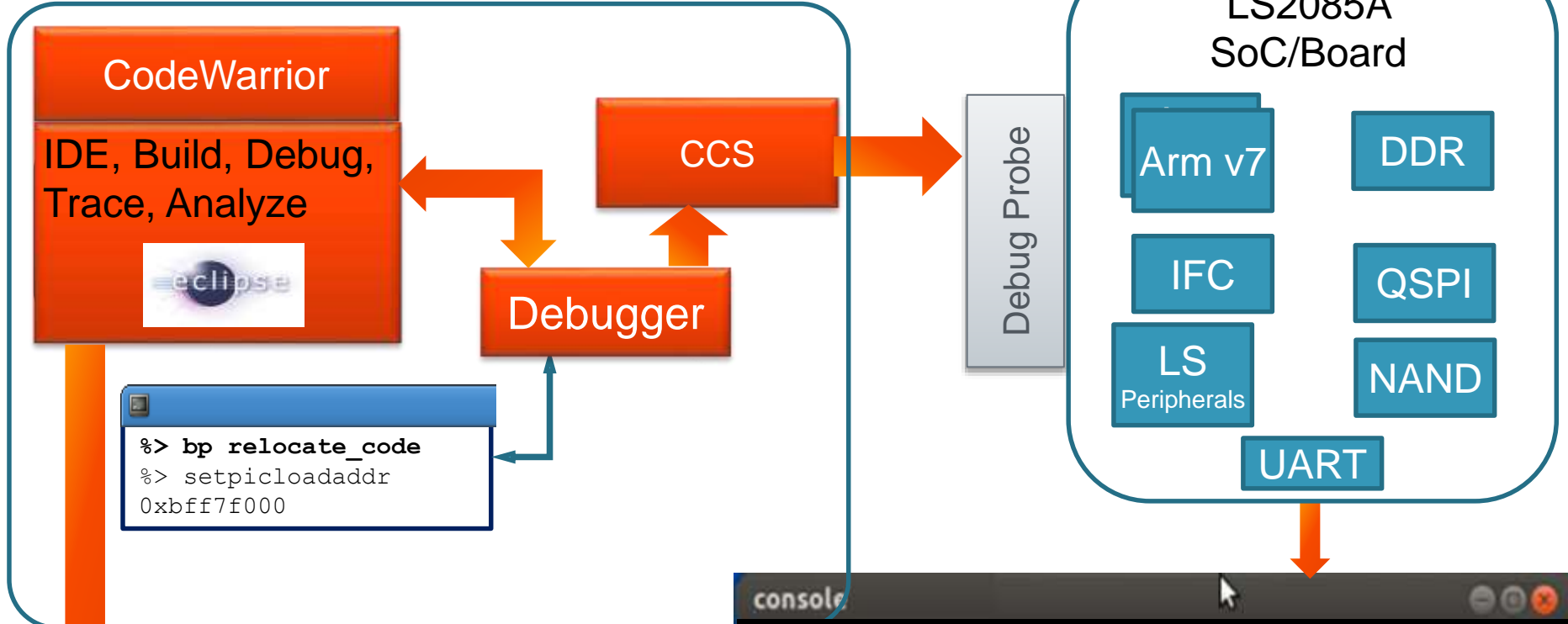
CodeWarrior
Flash

<nor> fl_write --erase 0x100000 c:/u-boot-ear4.0.bin
Erasing from offset 0x100000 0x860e0 bytes
Writing...
Write c:/u-boot-ear4.0.bin in 9.49s
<nor> _
```

U-Boot Debug



CodeWarrior : U-boot debug



```
%> bp relocate_code
%> setpicloadaddr
0xbff7f000
```

- CW project: import u-boot elf
- Run Control: run/suspend/step
- Breakpoints
- Registers View: GPR + SoC registers

```
console
U-Boot 2014.01-gb330cec (Apr 18 2014 - 17:38:22)
CPU: Freescale LayerScape LS2085A, Version: 1.0,
(0x87080310)
Clock Configuration:
CPU0(ARMv8):800 MHz,
Bus:400 MHz, DDR:400 MHz,
Board: LS2085AQDS
I2C: ready
relocate start
```

CodeWarrior : U-boot debug

- U-boot bring-up and debugging
 - Import u-boot ELF with symbol information
 - Debug from first u-boot instruction (in flash)
 - Debug after u-boot relocation in ram / relocate symbols
 - Debug to console prompt
 - Debug to kernel hand-off
- Registers View: GPR + SoC registers
- Debugging features:
 - Run control run/suspend/step
 - Breakpoints, in any ARMv8 EL mode
 - Disassembly, Memory view, Variable View, Expressions
- **Prerequisite**
 - U-boot image (optionally with symbolic information, useful for source level debug)

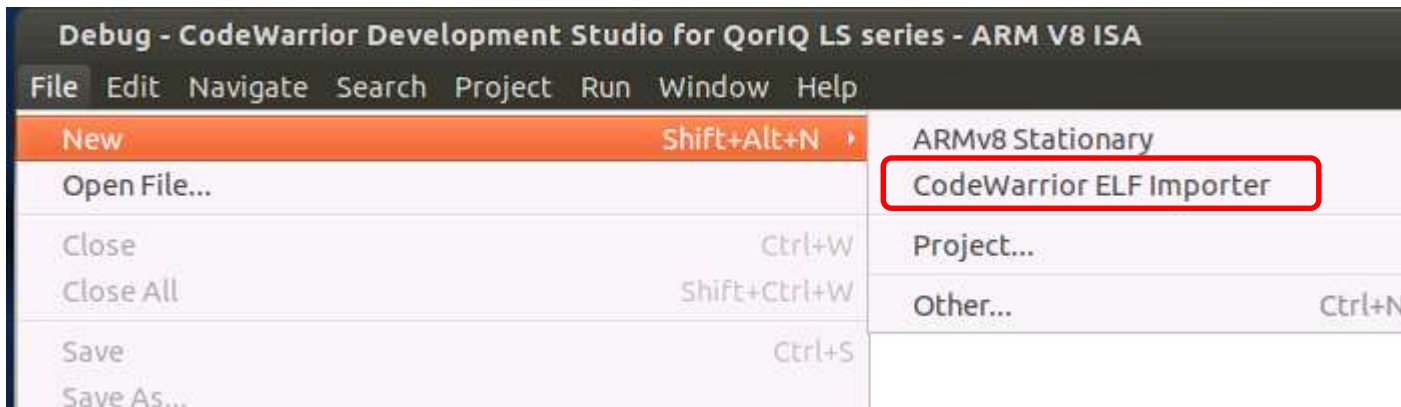
Activity

UBOOT- Debug



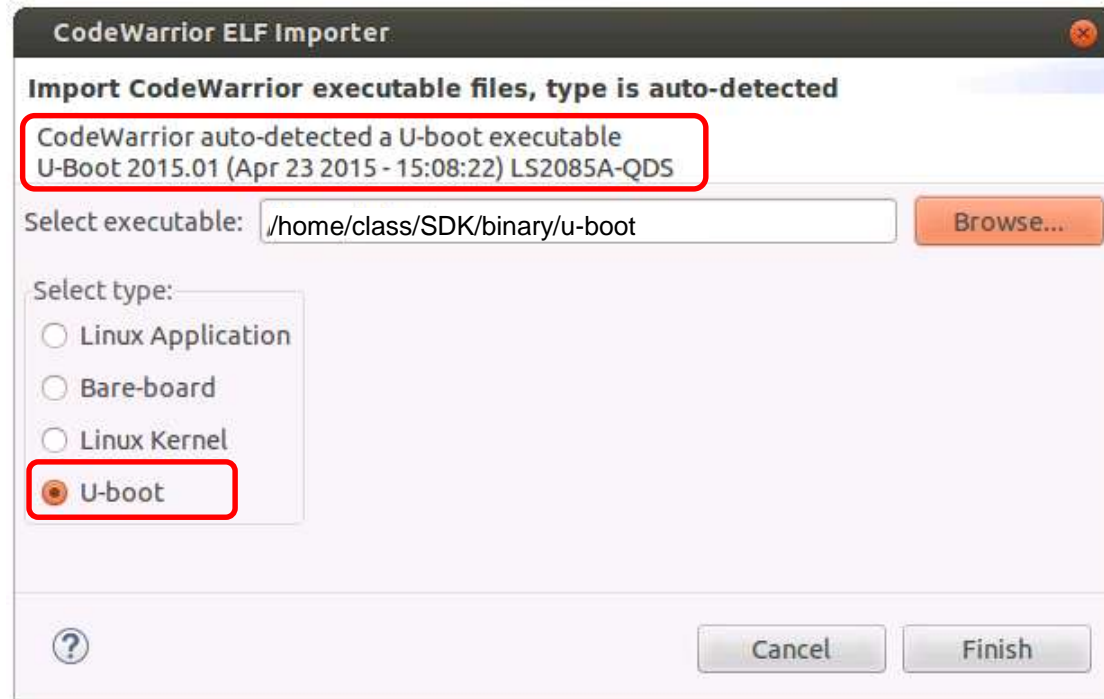
U-boot debug – Create project – Activity

- Select File > New > CodeWarrior ELF Importer
 - CodeWarrior Elf importer – automatically detects the ELF type and applies the correct awareness settings



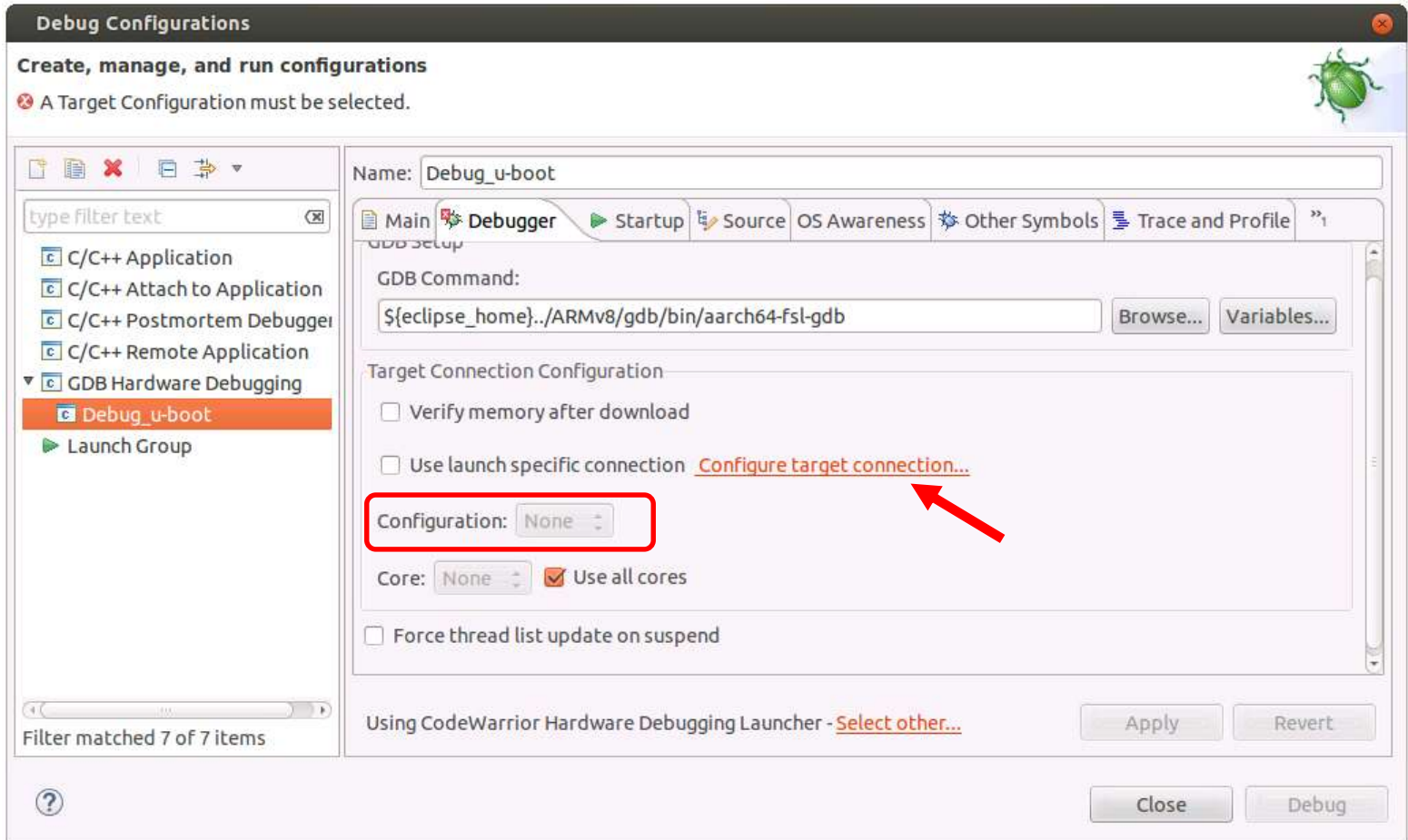
U-boot debug – Create project – Activity

- Click Browse...
- Select your u-boot file. The file must have debug information.
- CodeWarrior automatically:
 - Detect the ELF type as u-boot
 - Show u-boot summary information: u-boot version, build time, configuration
 - Apply the right debugger settings for debugging u-boot



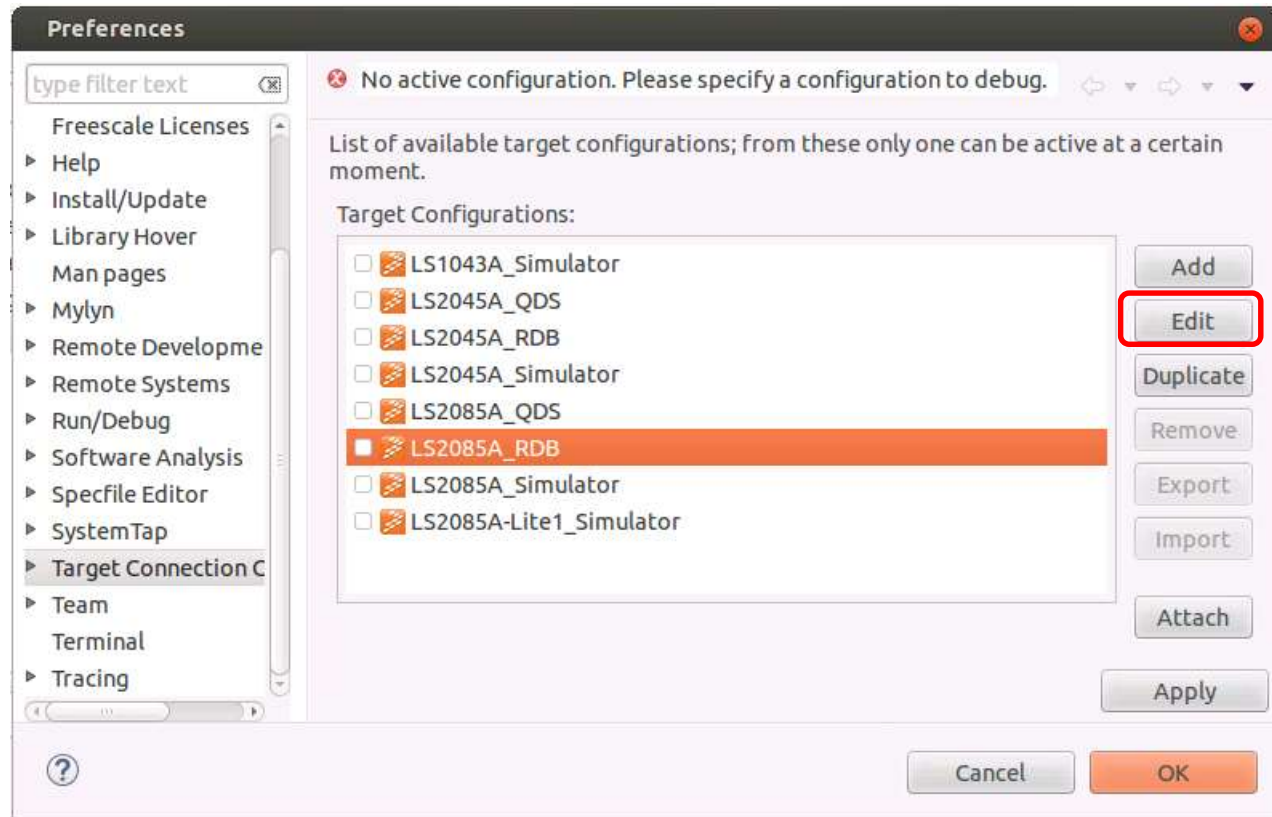
U-boot debug – Create Target Connection – Activity

- If no target connection is set, configure one following the link “[Configure target connection...](#)”



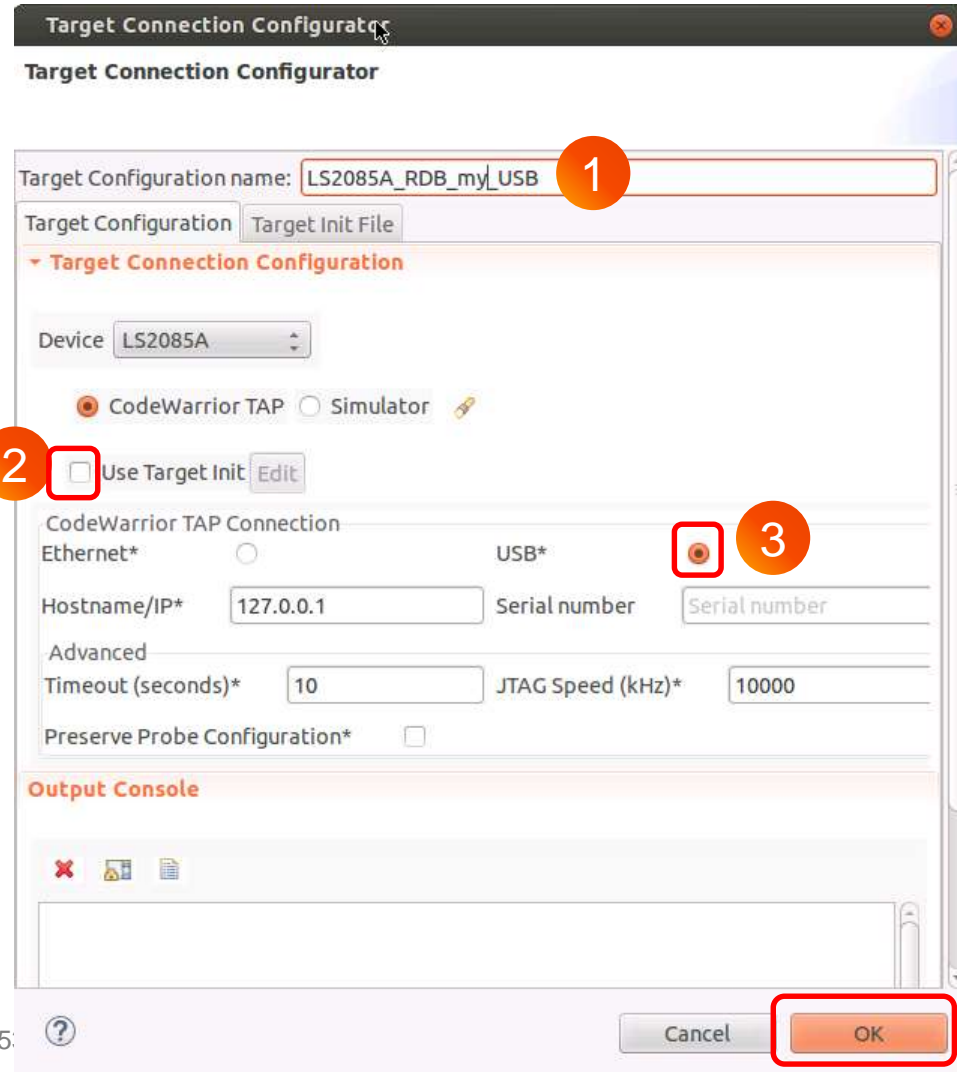
U-boot debug – Create Target Connection – Activity

- Select the desired target configuration template (e.g. LS2085A_RDB)
- Press Edit to change the template configuration



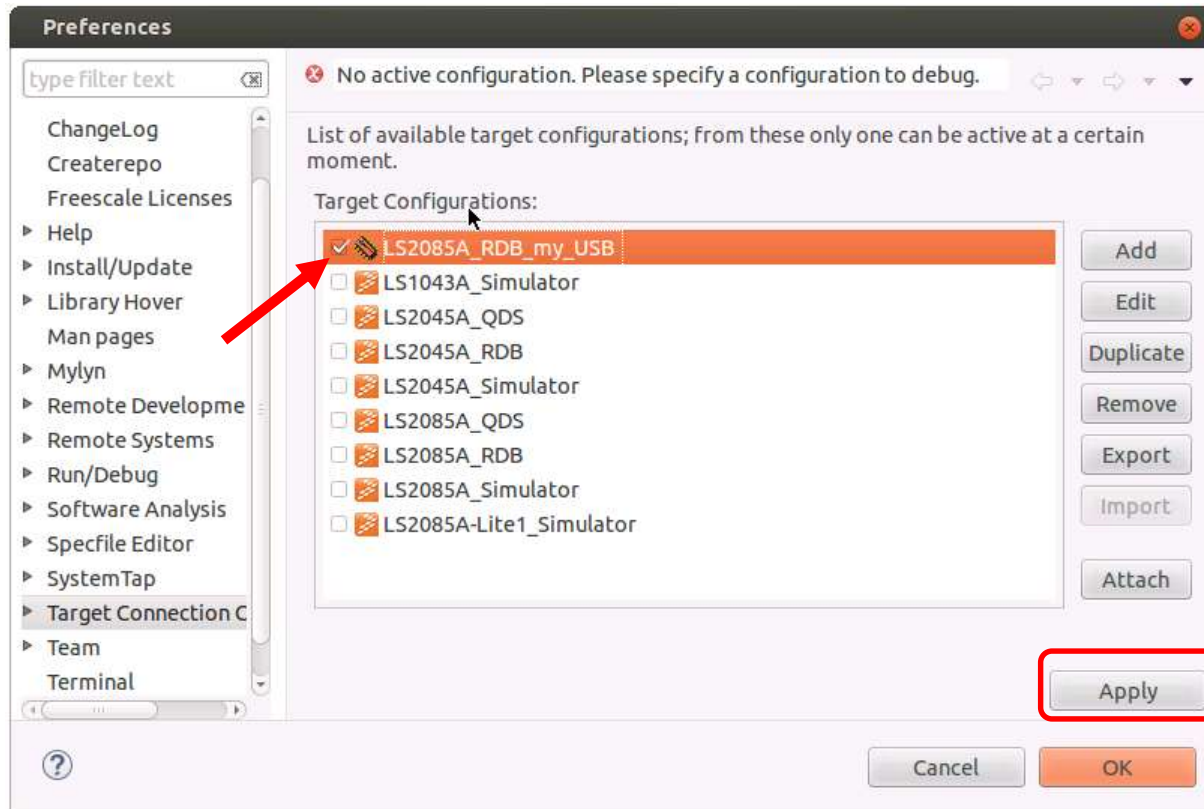
U-boot debug – Create Target Connection – Activity

- 1. Edit the name (different than the template name)
 - 2. Un-check “Use Target Init” checkbox
 - 3. Select USB
- or
- 3. Select Ethernet and set the correct CWTAP IP
- Press OK



U-boot debug – Create Target Connection – Activity

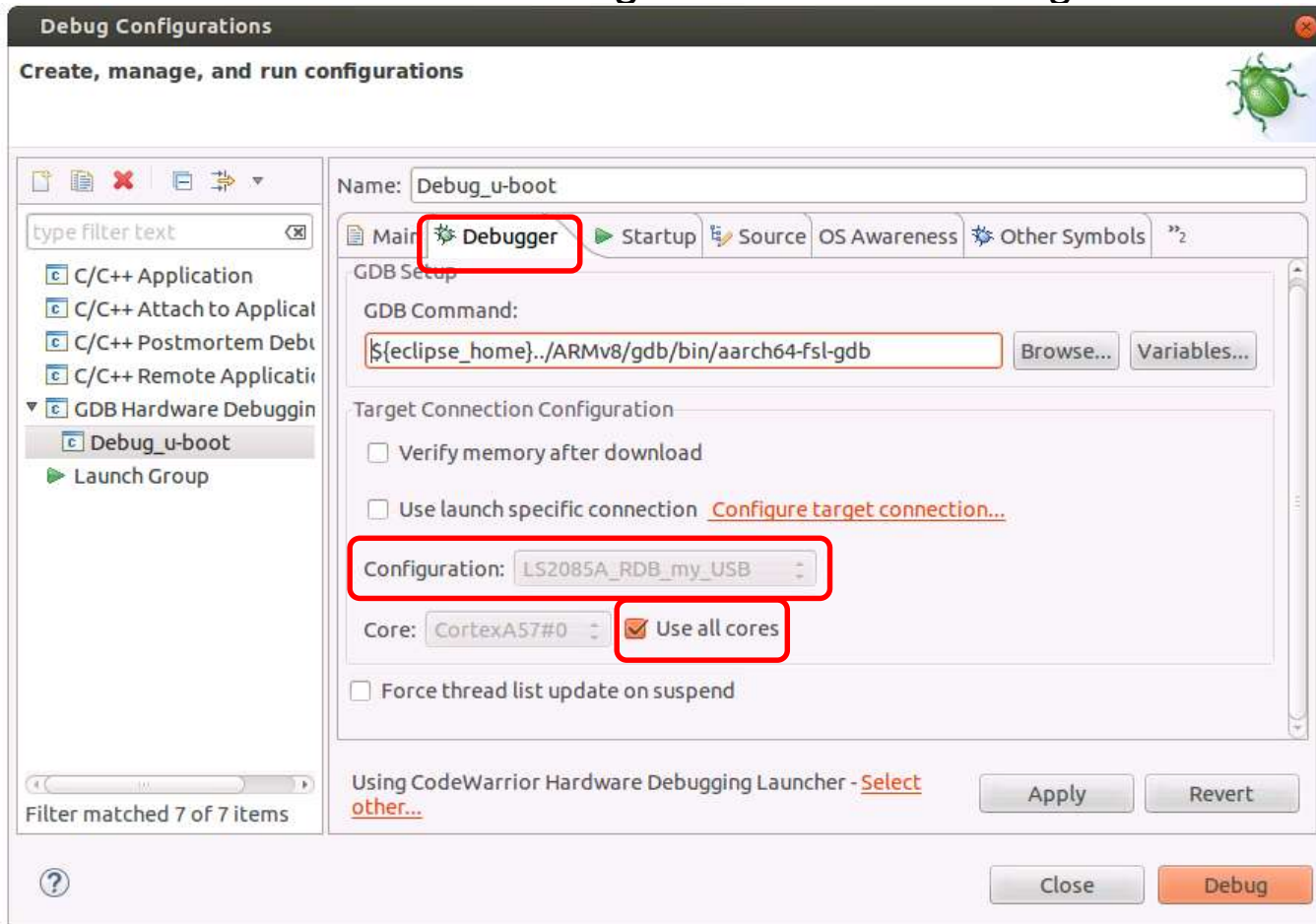
- Check the newly created target configurations to be set as active target configuration
- Apply the changes pressing Apply button



U-boot debug – Other Project Settings – Activity

- Debugger Tab

- Configuration automatically set to the default Target Connection
- Uncheck “Use all cores” to do single core u-boot debug



U-boot debug – Other Project Settings – Activity

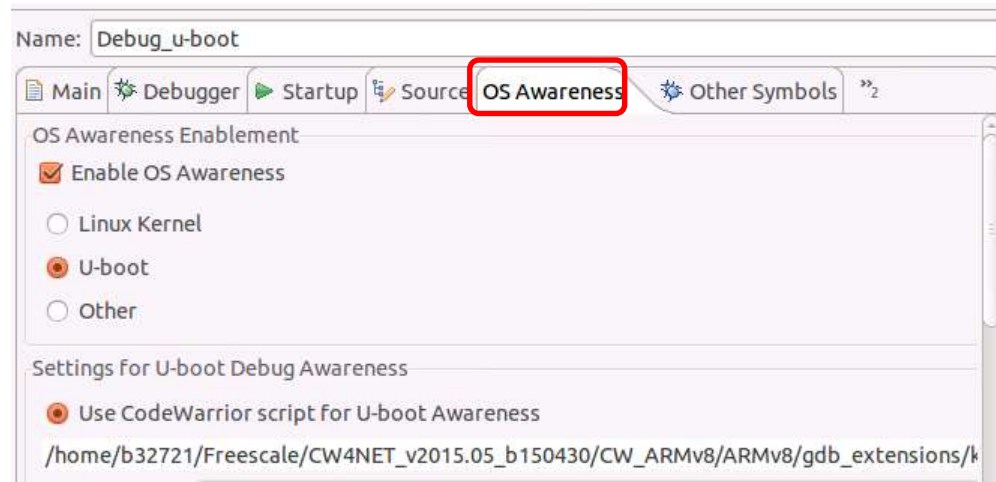
- Startup Tab

- Uncheck Reset and Delay to attach to the current running u-boot
- Check Reset and Delay = 0 to reset the target and debug u-boot from the first instruction after reset



- OS Awareness Tab

- U-boot awareness settings have been automatically applied when CodeWarrior ELF Importer detects the u-boot executable

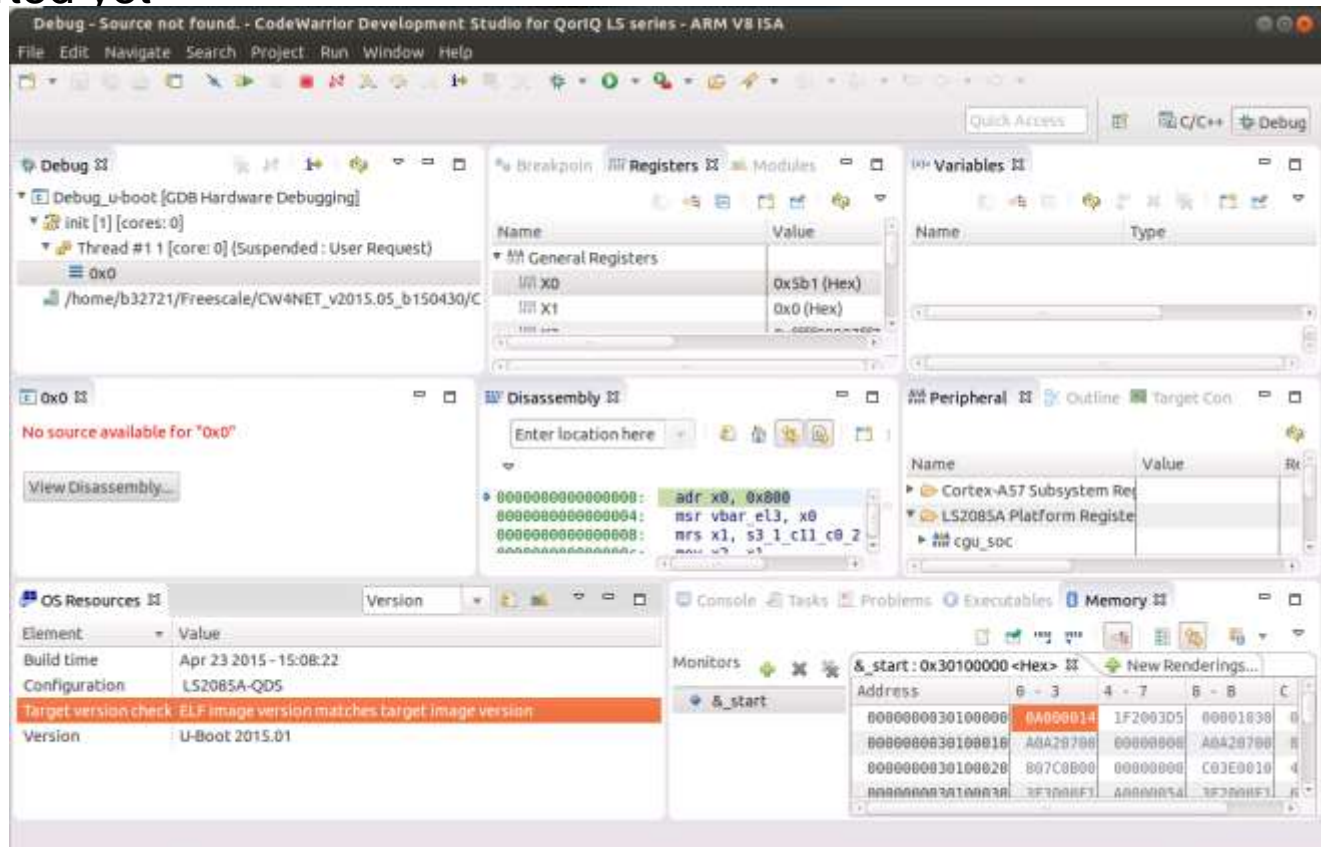


U-boot debug – Debug Overview – Activity

- **U-boot Awareness**
 - A single u-boot debug session while **the user is not aware about any stages or relocation offsets.**
 - The debugger automatically detects each u-boot stage and performs the corresponding action.
 - The user can visualize meaningful u-boot information about: u-boot version, build time or memory information
 - Target image vs Elf image version check
- Demonstrate a full u-boot debug session
 - debug from the first instruction after reset, to u-boot entry point, running from Flash, after relocation to DDRAM and until u-boot prompt is available
 - All is done in a single debug session with no other changes

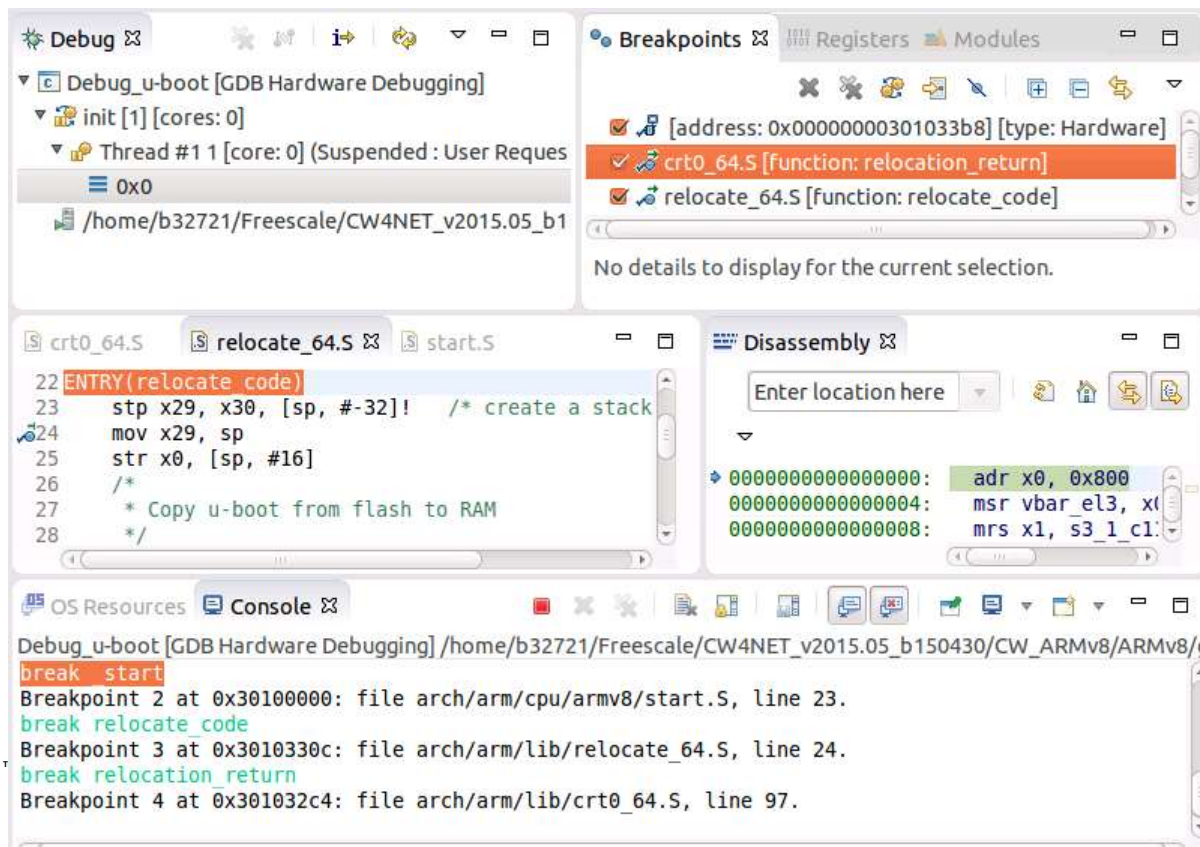
U-boot debug – Debug – Activity

- Click Launch Configuration > “Debug” button to start the debug session
- The target is reset and the debugger stops at the first instruction after reset
 - PC is 0x0 – BootRom
- No symbols or source file view available in u-boot ELF – the u-boot code has not been started yet



U-boot debug – Debug – Activity

- To demonstrate full u-boot debug in a single session, set several breakpoints in some key points of booting process
- Set breakpoints from console or from file:
 - `_start`: u-boot entry point
 - `relocate_code`: when running from Flash and relocation to DDRAM is started
 - `relocation_return`: first function executed from DDRAM



Debug_u-boot [GDB Hardware Debugging]

init [1] [cores: 0]

Thread #1 1 [core: 0] (Suspended : User Request)

0x0

/home/b32721/Freescale/CW4NET_v2015.05_b1

Breakpoints

[address: 0x00000000301033b8] [type: Hardware]

crt0_64.S [function: relocation_return]

relocate_64.S [function: relocate_code]

No details to display for the current selection.

Disassembly

Enter location here

0000000000000000: adr x0, 0x800

0000000000000004: msr vbar_el3, x0

0000000000000008: mrs x1, s3_1_c1

OS Resources Console

Debug_u-boot [GDB Hardware Debugging] /home/b32721/Freescale/CW4NET_v2015.05_b150430/CW_ARMv8/ARMv8/g

```
break start
Breakpoint 2 at 0x30100000: file arch/arm/cpu/armv8/start.S, line 23.
break relocate_code
Breakpoint 3 at 0x3010330c: file arch/arm/lib/relocate_64.S, line 24.
break relocation_return
Breakpoint 4 at 0x301032c4: file arch/arm/lib/crt0_64.S, line 97.
```

U-boot debug – Debug – Activity

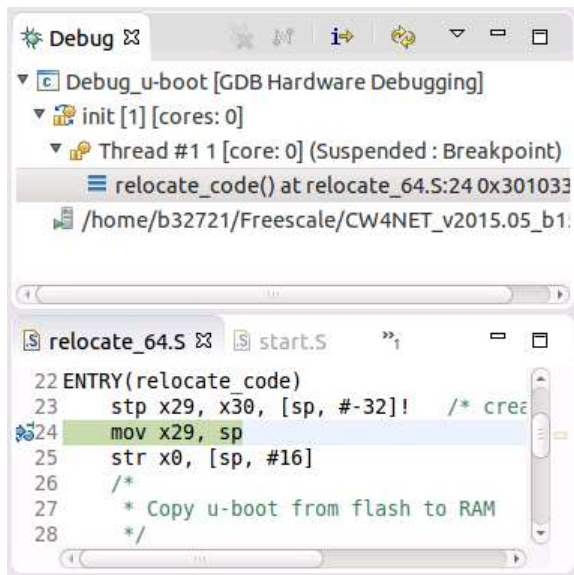
- Entry point breakpoints hit
- Note: breakpoint set as SW breakpoint (default) to a Flash (read-only) address. U-boot Awareness automatically changes the breakpoint type to HW for Flash address.

The screenshot displays the GDB interface for debugging U-boot. The 'Debug' pane shows the current thread is suspended at a breakpoint in the `_start()` function. The 'Breakpoints' pane shows the values of general registers X0, X1, and X2. The 'Disassembly' pane shows the assembly code for the `start` function, with the instruction `b 0x30100028 <reset>` highlighted. The 'Console' pane shows the GDB prompt and output, including a note that is highlighted with a red box:

```
Debug_u-boot [GDB Hardware Debugging] /home/b32721/Freescale/CW4NET_v2015.05_b150430/CW_ARMv8/ARMv8/g
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Note: automatically using hardware breakpoints for read-only addresses.
```

U-boot debug – Debug – Activity

- Start relocation breakpoint hit
- U-boot running from Flash (address 0x3010xxxx) prepare for running from DDRAM
- Booting messages on u-boot console



The screenshot shows a GDB debugger window titled "Debug_u-boot [GDB Hardware Debugging]". The "Thread #1 1 [core: 0] (Suspended: Breakpoint)" is selected. The current instruction is "relocate_code() at relocate_64.S:24 0x301033". Below, the source code for "relocate_64.S" is displayed, with line 24 highlighted: "mov x29, sp".

```
U-Boot 2015.01 (Apr 23 2015 - 15:08:22) LS2085A-QDS

Clock Configuration:
CPU0(A57):1600 MHz  CPU1(A57):1600 MHz  CPU2(A57):1600 MHz
CPU3(A57):1600 MHz  CPU4(A57):1600 MHz  CPU5(A57):1600 MHz
CPU6(A57):1600 MHz  CPU7(A57):1600 MHz
Bus: 600 MHz  DDR: 1333.333 MHz  DP-DDR: 1333.333 MHz

Reset Configuration Word (RCW):
00: 40282830 40400040 00000000 00000000
10: 00000000 00000000 00200000 00000000
20: 00c12980 00002580 00000000 00000000
30: 00000003 00000000 00000000 00000000
40: 00000000 00000000 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 00027000 00000000
70: 3f030007 00050000 00000000 00000000

Board: LS2085A-QDS, Board Arch: V1, Board version: B, boot from vBank: 0
FPGA: v5 (LS2085AQDS 2015 0218 1606), build 132 on Wed Feb 18 22:06:38 2015
SERDES1 Reference : Clock1 = 156.25MHz Clock2 = 100 separate SSCGMHz
SERDES2 Reference : Clock1 = 100 separate SSCGMHz Clock2 = 100 separate SSCGMHz
I2C: ready
DRAM: Initializing DDR...using SPD
Detected UDIMM 18ASF1G72AZ-2G1A1
Detected UDIMM 18ASF1G72AZ-2G1A1
DP-DDR: Detected UDIMM 18ASF1G72AZ-2G1A1
19.5 GiB
DDR 15.5 GiB (DDR4, 64-bit, CL=9, ECC on)
DDR Controller Interleaving Mode: 256B
DDR Chip-Select Interleaving Mode: CS0+CS1
DP-DDR 4 GiB (DDR4, 32-bit, CL=9, ECC on)
DDR Chip-Select Interleaving Mode: CS0+CS1
```



U-boot debug – Debug – Activity

- U-boot running from DDRAM (relocation_return breakpoint hit)
- DDRAM Address 0xFFF3xxxx
- U-boot Awareness breakpoint hit (see console) – debugger automatically handles the symbols relocation for DDRAM

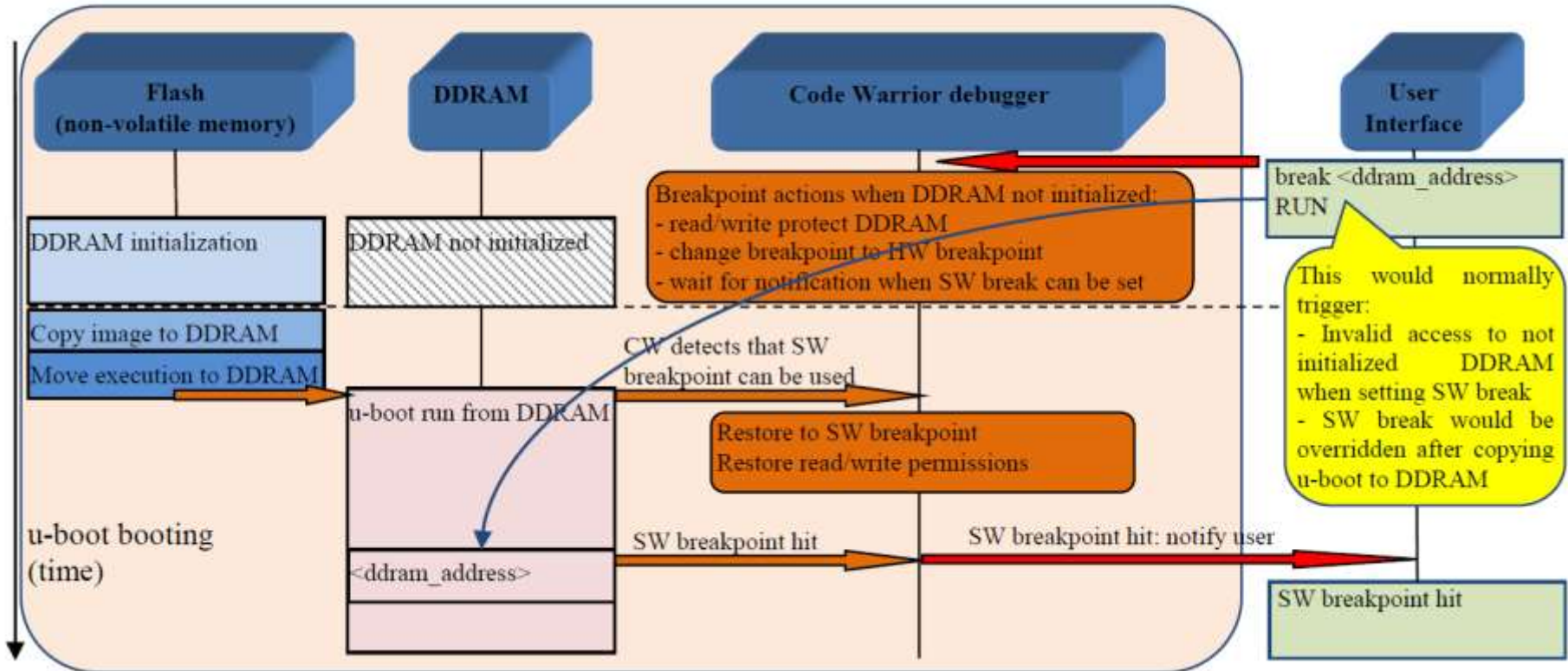
The screenshot displays a debugger interface with the following components:

- Debug Console:** Shows the current thread as `Thread #1 1 [core: 0] (Suspended : User Request)` and the current function as `_main() at crt0_64.S:97 0xffff3b2c4`. The file path is `/home/b32721/Freescale/CW4NET_v2015.05_b1`.
- Breakpoints:** Shows a list of breakpoints with columns for Name, Value, and Description.
- Registers:** Shows the General Register File with X0 and X1 registers. X0 has a value of `0xffffa7e80 (Hex)` and X1 has a value of `0xffffa7e60 (Hex)`.
- Disassembly:** Shows the disassembly of the current instruction at address `00000000ffff3b2c4`, which is `bl 0xffff38138 <c runtime cpl`. Other instructions shown include `ldr x0, = bss_start`, `ldr x0, 0xffff3b2f8 <clear_lc`, `ldr x1, = bss_end`, `ldr x1, 0xffff3b300 <clear_lc`, and `mov x2, #0`.
- Console:** Shows the output of the debugger, including the message `U-boot Awareness: u-boot relocation breakpoint is hit`.



U-boot debug – Debug – Activity

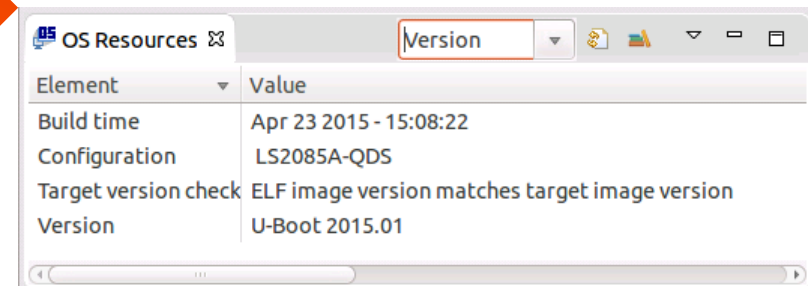
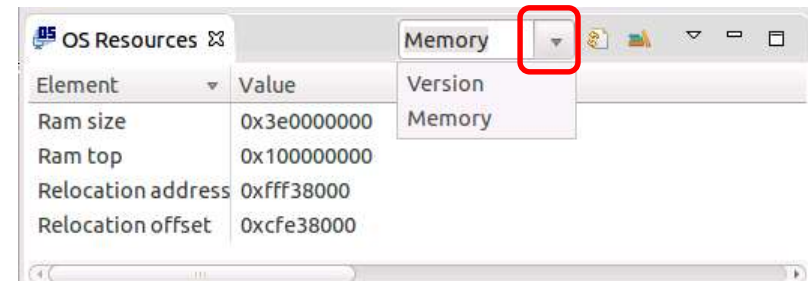
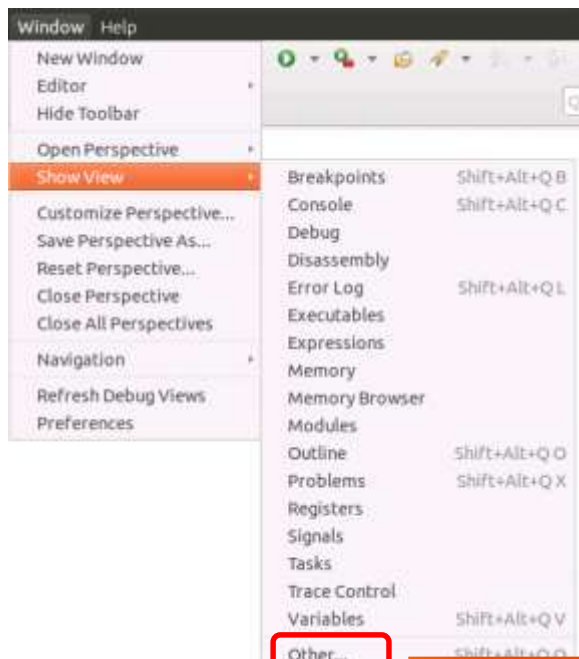
- U-boot Awareness allows setting SW breakpoints to DDRAM before DDRAM initialization and before u-boot relocation to DDRAM



U-boot debug – U-boot information – Activity

OS Resources: Windows > Show View > Other > Debugger > OS Resources

- Visualize meaningful u-boot information about: u-boot version, build time or memory information
- Target image vs Elf image version check



U-boot debug – Register / Peripheral – Activity

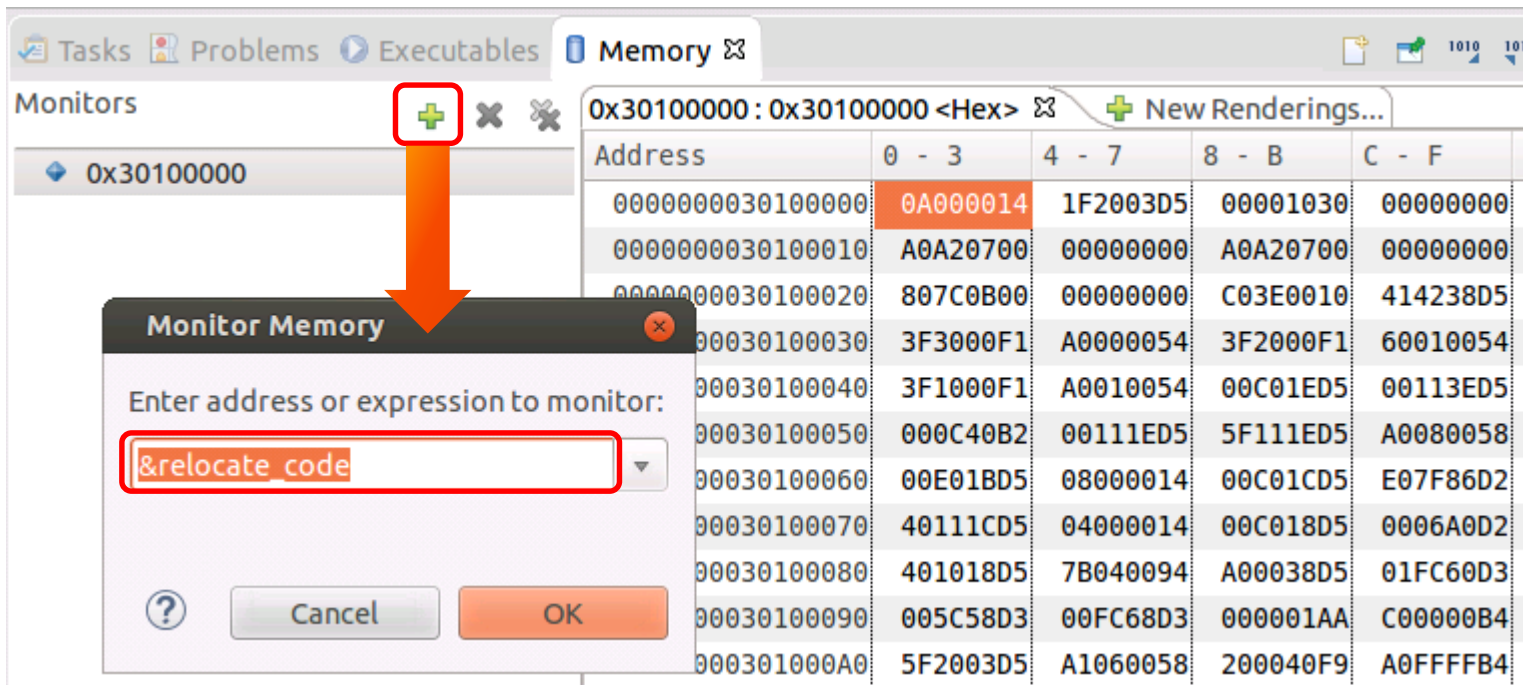
- Register view
- Peripheral: Windows > Show View > Other > Debugger > Peripherals

Name	Value
General Registers	
X0	0xffffa7e80 (Hex)
X1	0xffffa7e60 (Hex)
X2	0x40 (Hex)
X3	0x3f (Hex)
X4	0xffff9c100 (Hex)
X5	0xe7ff069f (Hex)
X6	0xfffffff0 (Hex)
X7	0x20 (Hex)
X8	0x10 (Hex)
X9	0xcfe38000 (Hex)
X10	0x3 (Hex)
X11	0x30164100 (Hex)
X12	0x80000002 (Hex)
X13	0x80000002 (Hex)
X14	0x80000002 (Hex)
X15	0x1800fad0 (Hex)
X16	0x1800f540 (Hex)
X17	0x1 (Hex)

Name	Value	Reset	Access	Location	Description
Cortex-A57 Subsystem Registers					
Core					
PSTATE					
Special_Purpose					
ID					
System					
Exception					
Memory					
Thread					
Timer					
Reset					
Debug					
External_Debug_Interface					
PERF					
LS2085A Platform Registers					
cpu_soc				0x116000	
cop_dcfg				0x100000	
cop_rst				0x1200b0	
cop_tap				0x3	
dbg_cdtc_wrapper_dbg				0x4000	
dbg_gdi				0x78000	
dbg_sdte_wrapper_dbg				0x84000	
DDR1				0x1080000	
DDR1_CS0_BNDS	0x000003ff	0x0	RW	0x1080000	Chip select 0 memory bounds
SA	0x0	0x0	RW	[31:16]	Starting address for chip select (bank)n. This value is compared against
EA	0x3ff	0x0	RW	[15:0]	Ending address for chip select (bank)n. This value is compared against
DDR1_CS1_BNDS	0x000003ff	0x0	RW	0x1080008	Chip select 1 memory bounds
DDR1_CS2_BNDS	0x000003ff	0x0	RW	0x1080010	Chip select 2 memory bounds

U-boot debug – Register / Peripheral – Activity

- Memory view: Window > Show View select “Memory”
- Add memory monitor
- Enter the address/expression



The screenshot shows the Memory monitor window in a debugger. The 'Monitors' list on the left contains one entry: '0x30100000'. A red box highlights the '+' icon next to this entry, with an orange arrow pointing down to the 'Monitor Memory' dialog box. The dialog box has the title 'Monitor Memory' and the text 'Enter address or expression to monitor:'. The input field contains the text '&relocate_code'. Below the input field are buttons for '?', 'Cancel', and 'OK'. The background shows a memory dump table with columns for Address and hex values.

Address	0 - 3	4 - 7	8 - B	C - F
0000000030100000	0A000014	1F2003D5	00001030	00000000
0000000030100010	A0A20700	00000000	A0A20700	00000000
0000000030100020	807C0B00	00000000	C03E0010	414238D5
00030100030	3F3000F1	A0000054	3F2000F1	60010054
00030100040	3F1000F1	A0010054	00C01ED5	00113ED5
00030100050	000C40B2	00111ED5	5F111ED5	A0080058
00030100060	00E01BD5	08000014	00C01CD5	E07F86D2
00030100070	40111CD5	04000014	00C018D5	0006A0D2
00030100080	401018D5	7B040094	A00038D5	01FC60D3
00030100090	005C58D3	00FC68D3	000001AA	C00000B4
000301000A0	5F2003D5	A1060058	200040F9	A0FFFFB4

u-boot Tracing

Demonstrate the CodeWarrior Trace capabilities

- HW trace configurations
- General trace controls
- Results and interpretations



u-boot tracing – Hardware setup

LS 2085A-RDB



CodeWarrior TAP over ETH/USB + Serial



Host PC running
CW ARMv8



- u-boot is a perfect example of a bare-board application use case with CW
- There are no other means to trace & profile the u-boot early stages without dedicated JTAG based tools

u-boot tracing: Software Analysis Overview

Use-case: Trace A57-core 0 by collecting ETM with timestamp into DTC in order to measure: execution speed, code coverage, platform initializations of standard u-boot

The screenshot shows the Trace Commander software interface. At the top, a dropdown menu displays 'Debug_u-boot.xml'. Below it, a 'TRACE GENERATORS' section contains several categories of trace generators: CORE (with buttons for Core 0 through Core 7), PXDI (with buttons for PXDI 0 through PXDI 3), DDDI (with buttons for DDDI 0 and DDDI 1), NOC_MAIN (with buttons for ac1, fdma, aiopother, tlu, caan1, caan2, wrlop1, wrlop2, wrlop3), NOC_HSIO (with buttons for tbu5, tbu6, tbu7), and STM (with a button for STM). At the bottom, a 'TRACE BUFFERS' section contains buttons for DTC, a central button with a play icon, and DOR. Red callout boxes provide the following information:

- Platform configuration**: xml file for setting up the options (points to the dropdown menu).
- Connect / Disconnect**: Allow trace control for each stage (points to the play icon button).
- single-click = start/stop trace**
double-click = configuration (points to the play icon button).
- All available SoC trace generator** (points to the CORE, PXDI, DDDI, NOC_MAIN, NOC_HSIO, and STM sections).
- Upload**: Save HW trace to local host (points to the central play icon button).

u-boot tracing: Software setup (cont'd)



double-click to configure trace

Trace Configurations

Software Analysis Configuration

Configure trace IP

Debug_u-boot.xml

- Trace Generators
 - CORE
 - Core 0
 - Core 1
 - Core 2
 - Core 3
 - Core 4
 - Core 5
 - Core 6
 - Core 7
 - DDDI
 - PXDI
 - NOC
 - STM
- Data Streams
- Results Folder

Core 0

1 Enable Trace

Trace

Trace scenarios

Program Trace

Bandwidth

MediumHigh

General Settings

2 Timestamp

Application Information

LoadAddress	New File	Add
Autodetected	X:\Layerscape2-SDK-20150515-yocto\build_ls2085ardb_r...	Remove

3

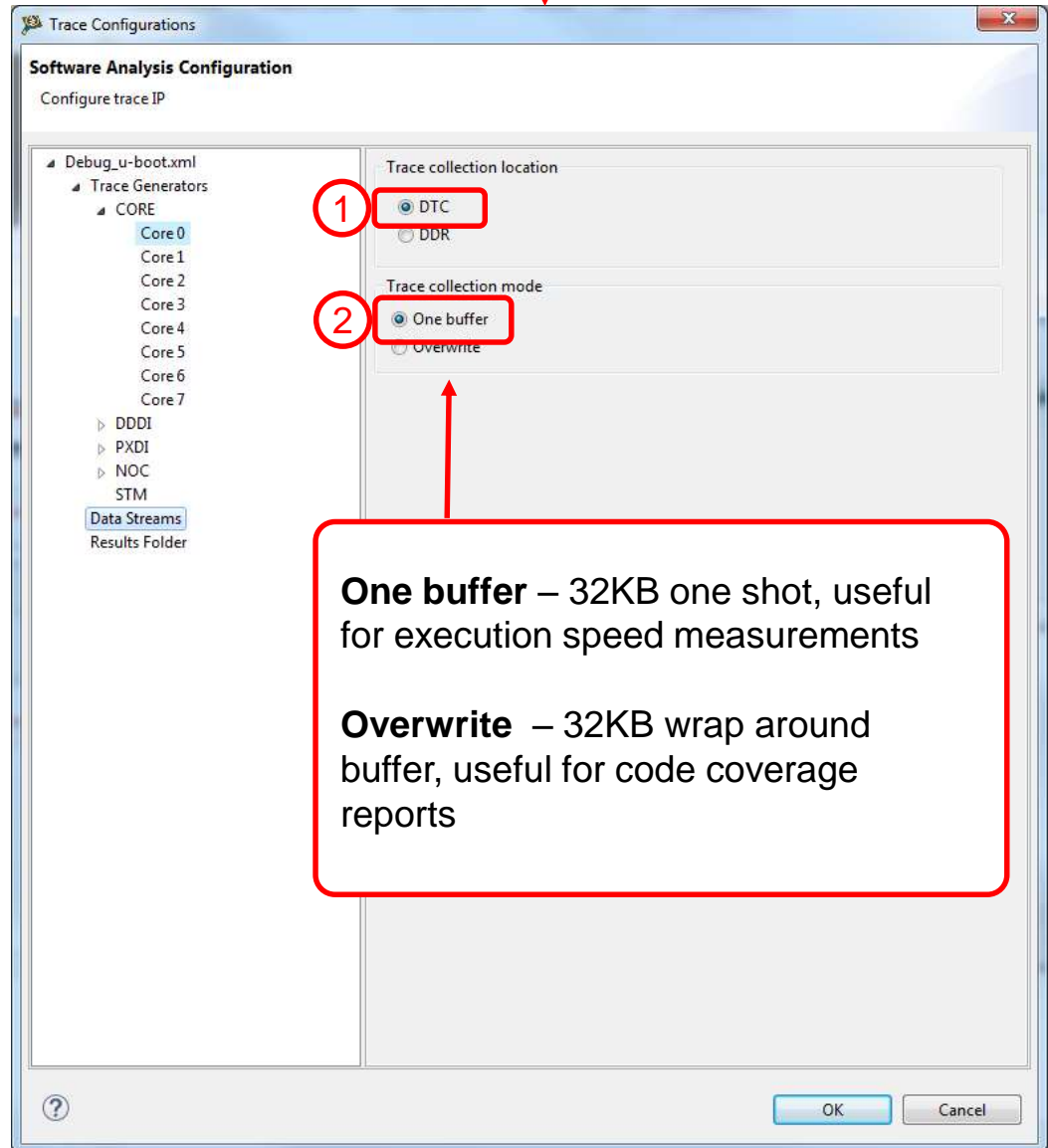
path to u-boot.elf, needed for obtaining debugging symbols

OK Cancel

u-boot tracing: Software setup (cont'd)



Select the destination for PTM stream and operation mode



u-boot tracing: actions per stage

1st stage – Trace BootRom

0x0 -first instruction after reset
_start (u-boot entry point)

2nd stage – Trace u-boot execution on Flash

_start (u-boot entry point – running from Flash)
relocate_code (u-boot relocation to DDRAM)

3rd stage – Trace u-boot relocation

relocate_code (u-boot relocation to DDRAM)
Relocation_return (first function executed in DDRAM)

4th stage – Trace u-boot in DDRAM

Board initializations

u-boot tracing: 1st stage setup

1

2

3

4

Set a breakpoint to `_start`, resume core execution and wait for BP to be hit

3

After CW suspends execution, import the trace data and save it as `trace_0`

u-boot tracing: 1st stage results

Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
Debug_u-boot							
DTC							
trace	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:32:...	
trace_0	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:32:...	1st stage

Quick links to results.
Note! Not all of them are relevant at this point

Index	Source	Type	Description	Address	Destination	Timestamp
1	Core 0	Info	SYNC packet - ETM			0
2	Core 0	Info	Trace On packet - ETM -> start tracing after a...			0
3	Core 0	Info	Context packet - ETM			0
			exception level: 3			
			state: 64-bit			
			security state: secure			
			ContextID: 0			
4	Core 0	Software Context	software context id = 0			0
5	Core 0	Linear	Function <no debug info>	0x0		0
6	Core 0	Linear	Function <no debug info>	0x0		8
7	Core 0	Linear	Function <no debug info>	0x0		58
8	Core 0	Linear	Function <no debug info>	0x0		224
9	Core 0	Linear	Function <no debug info>	0x0		224

BootRom execution time

Function Name	Num Calls	% Total calls of parent	% Total times it was called	Inclusive Time (Cycles)
Context 0				
f <start>				
f <no debug info>	1	100.00	100.00	3,466

u-boot tracing: 2nd stage setup

Debug - X:\Layerscape2-SDK-20150515-yocto\build_i20155arib_release\img\work\i20155arib-ti-linux-u-boot-2015.01+git-r0\target\armv8\relocate_64.5 - CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA

Debug Console: u-boot [GDB Hardware Debugging] u-boot [1] [cores: 0] Thread #1 [1] [core: 0] (Suspended: Breakpoint) relocate_code[] at relocate_64.5:24 [0x301035fc] C:\Freescale\CW\NET_u201505\CW_ARMv8\ARMv8\gdb\bin\asch64-6i-gdb.exe (7.7.1)

Breakpoints: [address: 0x00000000301036a0] [type: Hardware] relocate_64.5 [function: relocate_code] start.5 [function: start]

Disassembly: `00000000301035fc: mov x29, sp
0000000030103600: stp x8, [sp, #16]
0000000030103604: ld x1, [0x30103610 <relocate_done+8>
0000000030103608: stp x9, x8, x1
000000003010360c: b, q, 0x30103650 <relocate_done>
0000000030103610: ld x2, [0x301036b0 <relocate_done+96>
0000000030103614: ld x10, x11, [x1], #10
0000000030103618: stp x10, x11, [x8], #16
000000003010361c: <w> x1, x2
0000000030103620: b, c, 0x30103614 <relocate_code+28>
0000000030103624: stp x8, [sp, #24]
0000000030103628: ld x2, [0x301036c0 <relocate_done+104>
000000003010362c: stp x3, [0x301036c8 <relocate_done+112>
0000000030103630: <f> x1, x2`

Analysis Results:

Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
Debug_u-boot							
DTC						2015.05.26 04:35...	
trace_1	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:35...	2nd stage
trace_2	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:32...	1st stage

Console: Debug_u-boot [GDB Hardware Debugging] u-boot
break_start
break_relocate_code

Set a breakpoint to relocate_code, resume core execution and wait for BP to be hit

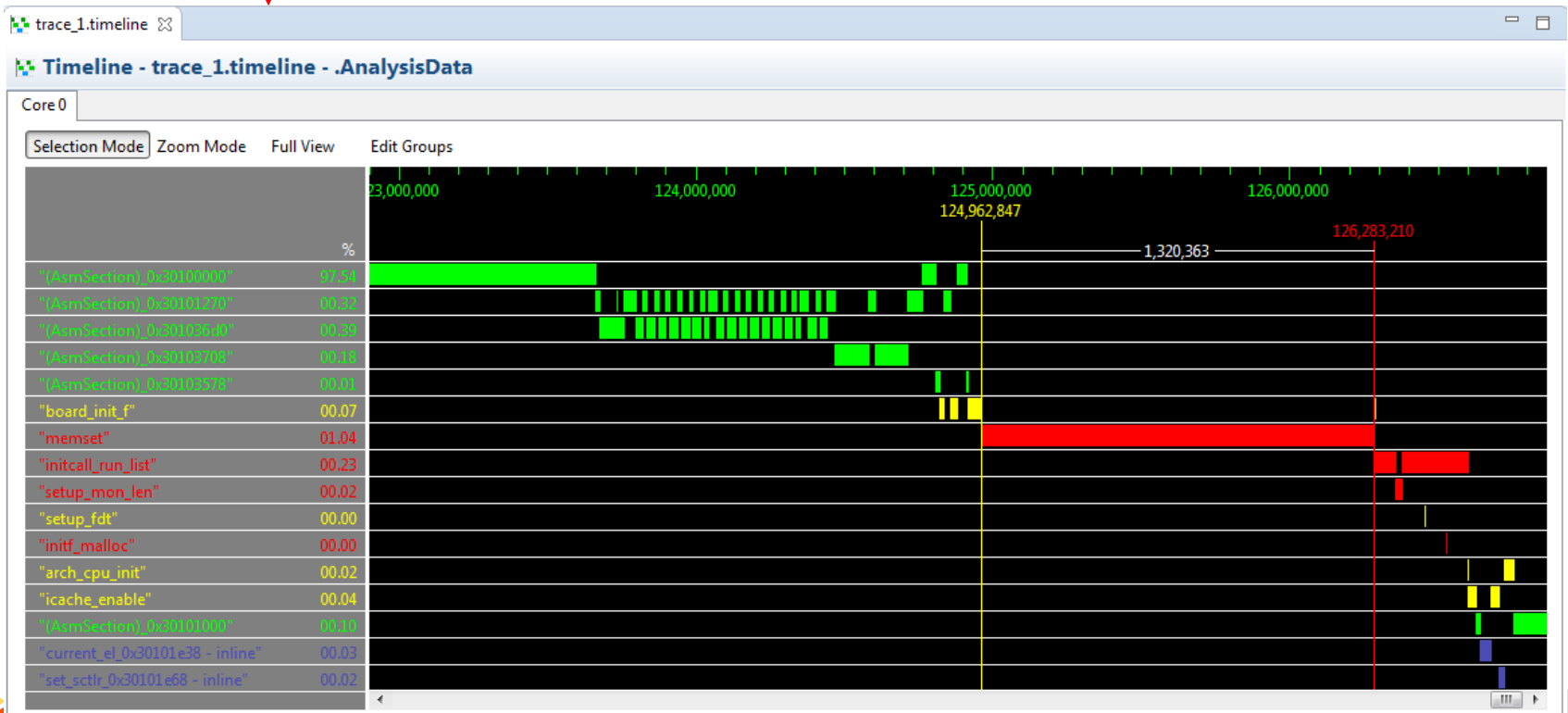
After CW suspends execution, import the trace data and save it as trace_1



u-boot tracing: 2nd stage results

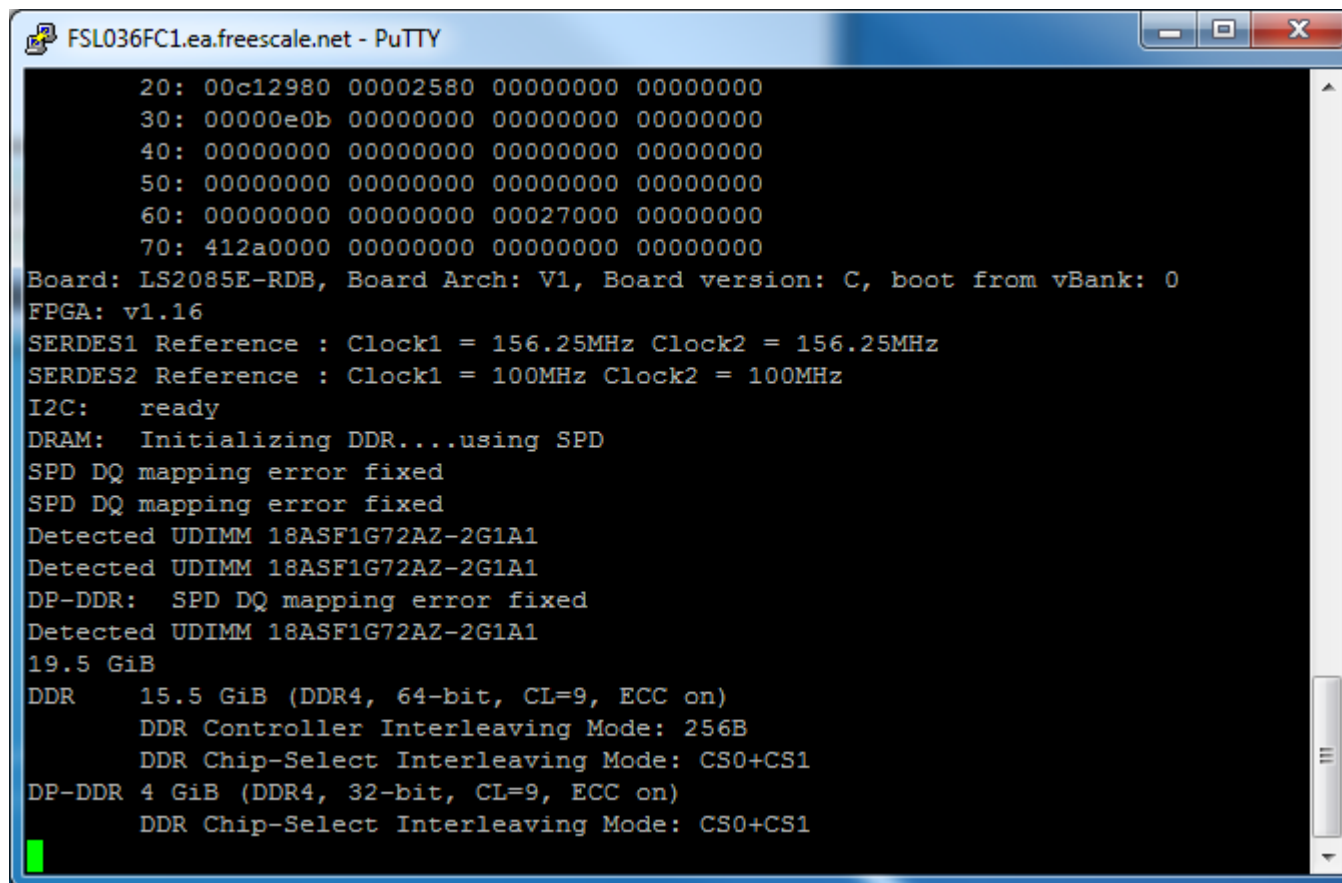
Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
Debug_u-boot							
DTC							
trace	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:35:...	
trace_1	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:35:...	2nd stage
trace_0	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:32:...	1st stage

Sequential view of u-boot routines:
- Easy highlighting of special routines, cycle count measurements



u-boot tracing: 2nd stage results (cont'd)

During 2nd stage the u-boot should display booting messages on console



```
FSL036FC1.ea.freescale.net - PuTTY
20: 00c12980 00002580 00000000 00000000
30: 00000e0b 00000000 00000000 00000000
40: 00000000 00000000 00000000 00000000
50: 00000000 00000000 00000000 00000000
60: 00000000 00000000 00027000 00000000
70: 412a0000 00000000 00000000 00000000
Board: LS2085E-RDB, Board Arch: V1, Board version: C, boot from vBank: 0
FPGA: v1.16
SERDES1 Reference : Clock1 = 156.25MHz Clock2 = 156.25MHz
SERDES2 Reference : Clock1 = 100MHz Clock2 = 100MHz
I2C: ready
DRAM: Initializing DDR...using SPD
SPD DQ mapping error fixed
SPD DQ mapping error fixed
Detected UDIMM 18ASF1G72AZ-2G1A1
Detected UDIMM 18ASF1G72AZ-2G1A1
DP-DDR: SPD DQ mapping error fixed
Detected UDIMM 18ASF1G72AZ-2G1A1
19.5 GiB
DDR 15.5 GiB (DDR4, 64-bit, CL=9, ECC on)
DDR Controller Interleaving Mode: 256B
DDR Chip-Select Interleaving Mode: CS0+CS1
DP-DDR 4 GiB (DDR4, 32-bit, CL=9, ECC on)
DDR Chip-Select Interleaving Mode: CS0+CS1
```

u-boot tracing: 3rd stage setup

Debug - X:\layerscope2-SDK-20150515-yocto/build_h2085ardb_release\img\work\h2085ardb-fd-linux/u-boot2015.01+git-r0\git\arm\bin\crt0_M4S - CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA

Breakpoints: 13

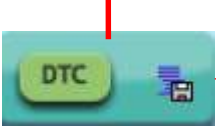
- (F) [address: 0d0000000301036ad] [type: Hardware]
- (F) xrt0_64.S [function: relocation_return]
- (F) relocate_64.S [function: relocate_code]
- (F) start5.S [function: start5]

Disassembly: start

```
00000000ffff1e504: bl 0xffff1b00 <<_runtime_cpu_setup>
00000000ffff1e508: ld  w0, 0xffff1e50 <<clear_loop+16>
00000000ffff1e50c: ld  x1, 0xffff1e50 <<clear_loop+44>
00000000ffff1e508: mov  x2, #0b // #0
clear_loop:
00000000ffff1e5c4: st  x2, [x0]
00000000ffff1e5d0: add  w0, w0, #wb3
00000000ffff1e5cc: cm  w0, x1
00000000ffff1e5d0: b.c  0xffff1e5c4 <<clear_loop>
00000000ffff1e5d4: mov  w0, x10
00000000ffff1e5d8: ld  x1, [x18, #0b]
00000000ffff1e5dc: b  0xffff2114 <<board_init_r>
00000000ffff1e5e0: ld  w16, 0xffff205dc <<main_loop+32>
00000000ffff1e5e4: ...
```

Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
trace_2	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:36:...	3rd stage
trace_1	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:35:...	2nd stage
trace_0	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.05.26 04:32:...	1st stage

Set a breakpoint to relocation_return, resume core execution and wait for BP to be hit



After CW suspends execution, import the trace data and save it as trace_2



u-boot tracing: 3rd stage results

Two types of code coverage reports:

- File level
- Instruction level

Summary Table

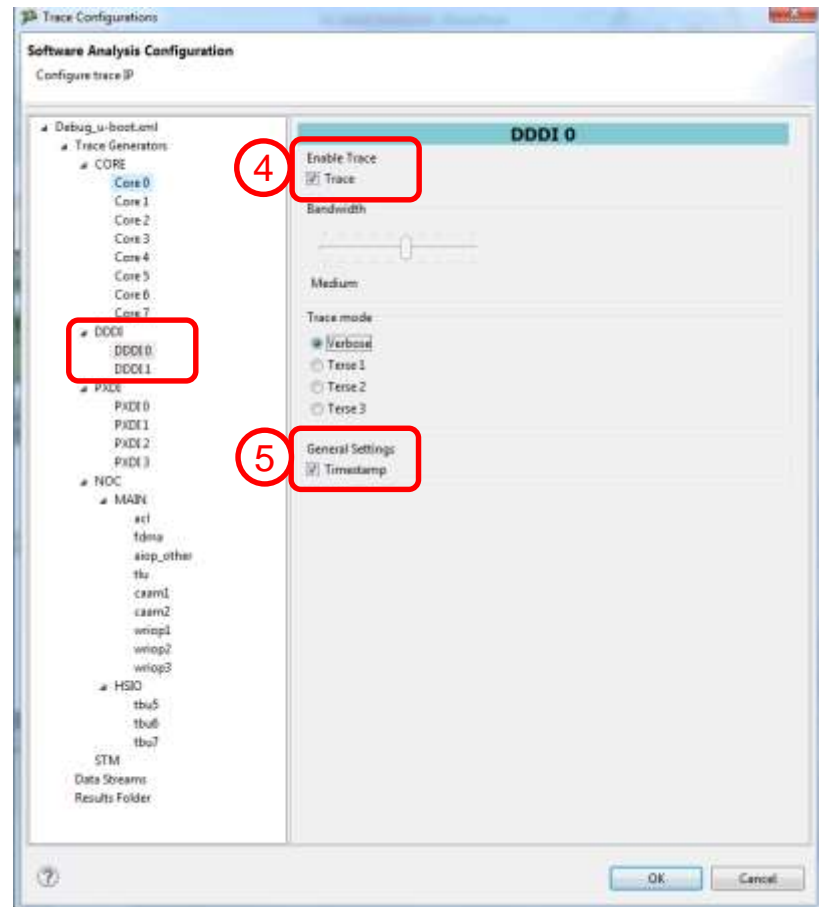
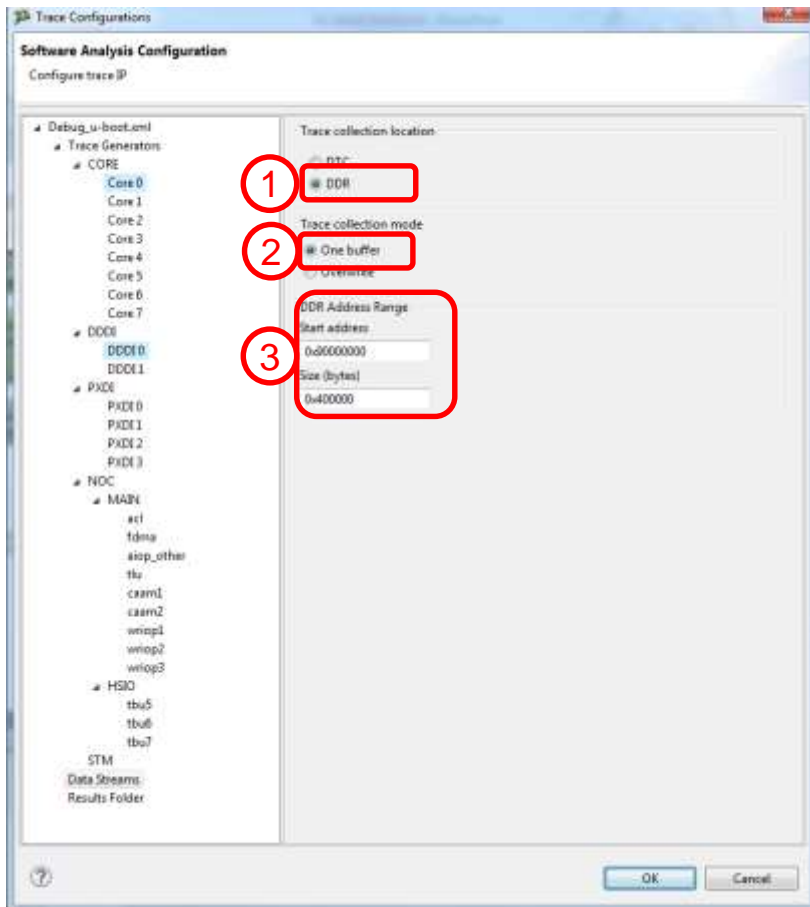
File/Function	Address	Covered Source %	Partially Covered Source %	Not Covered Source %	Total number of source lines	ASM Decision Co...
Context 0						
relocate_64.S	N/A	23.08 %	0.00 %	76.92 %	39	11.11 %
(AsmSection)_0x301035f8	0x301035f8	23.08 %	0.00 %	76.92 %	39	11.11 %
cache.S	N/A	0.00 %	0.00 %	100.00 %	77	0.00 %
cache v8.c	N/A	0.00 %	0.00 %	100.00 %	40	0.00 %

Details Table

Address	Instruction	Coverage	ASM Decision Coverage	ASM Count
0x30103	ldr x1, #172	covered		1
0x30103	subs x9, x0, x1	covered		1
0x30103	b.eq #76	covered	only not-taken	1
0x30103	ldr x2, #168	covered		1
0x30103	ldp x10, x11, [x1], #16	covered		9,704
0x30103	stp x10, x11, [x0], #16	covered		9,704
0x30103	cmp x1, x2	covered		9,704
0x30103	b.lo #-12	covered	only taken	9,704
0x30103	str x0, [sp, #24]	not covered		0
0x30103	ldr x2, #152	not covered		0

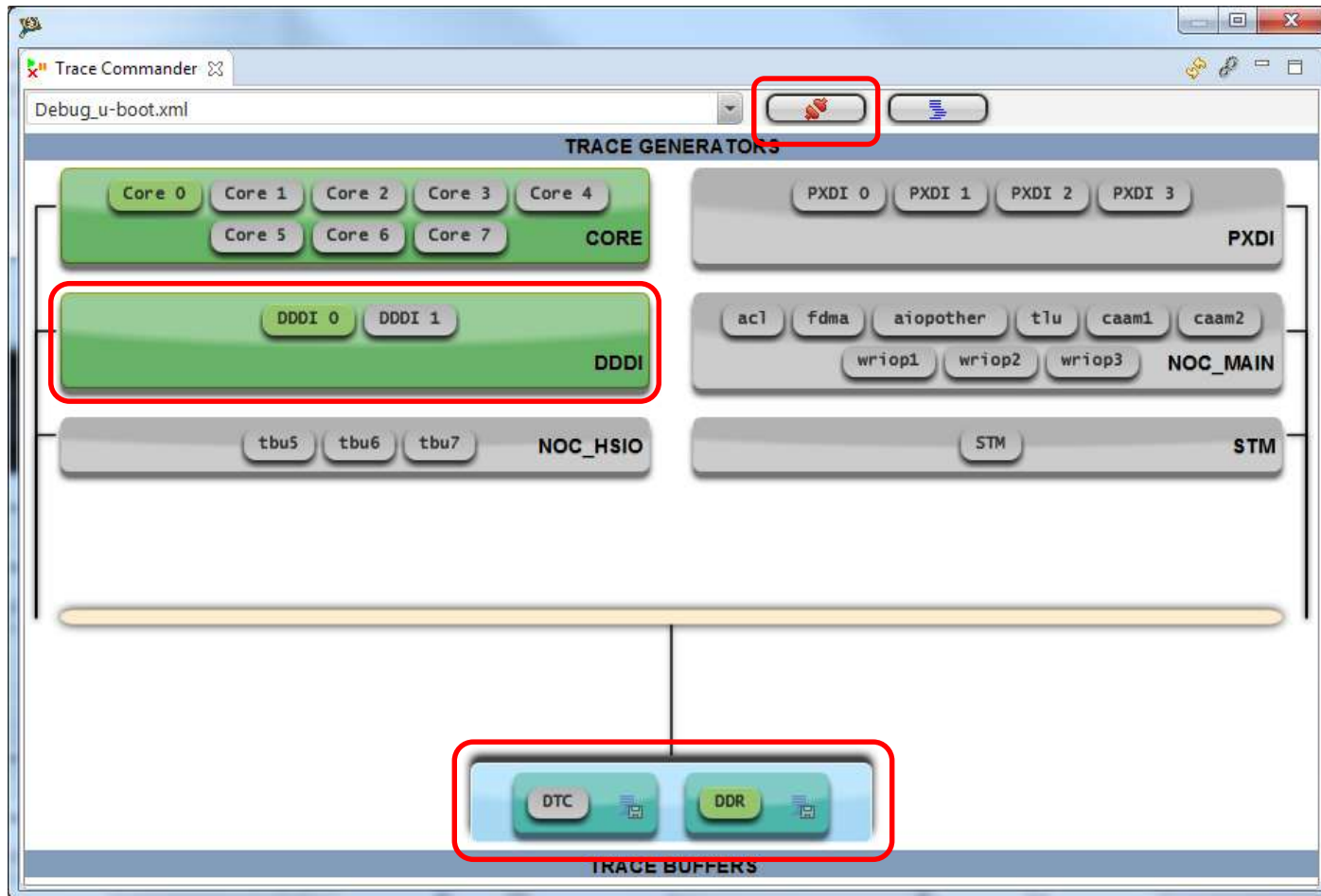
u-boot tracing: 4th stage setup

- At this point the DDR is initialized. Move Trace Buffer into DDR in order to remove 32KB size limitation;
- DDR trace event can be captured too























u-boot tracing: 4th stage setup (cont'd)

- Re-apply target settings



u-boot tracing: 4th stage results

Analysis Results

Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
Debug_u-boot							
DDR							
trace_3	 Trace	 Timeline	 Code Coverage	 Performance	 Call Tree	2015.05.26 04:52:...	4th stage
DTC							
trace_2	 Trace	 Timeline	 Code Coverage	 Performance	 Call Tree	2015.05.26 04:47:...	3rd stage
trace_1	 Trace	 Timeline	 Code Coverage	 Performance	 Call Tree	2015.05.26 04:35:...	2nd stage
trace_0	 Trace	 Timeline	 Code Coverage	 Performance	 Call Tree	2015.05.26 04:32:...	1st stage

trace_3.csv

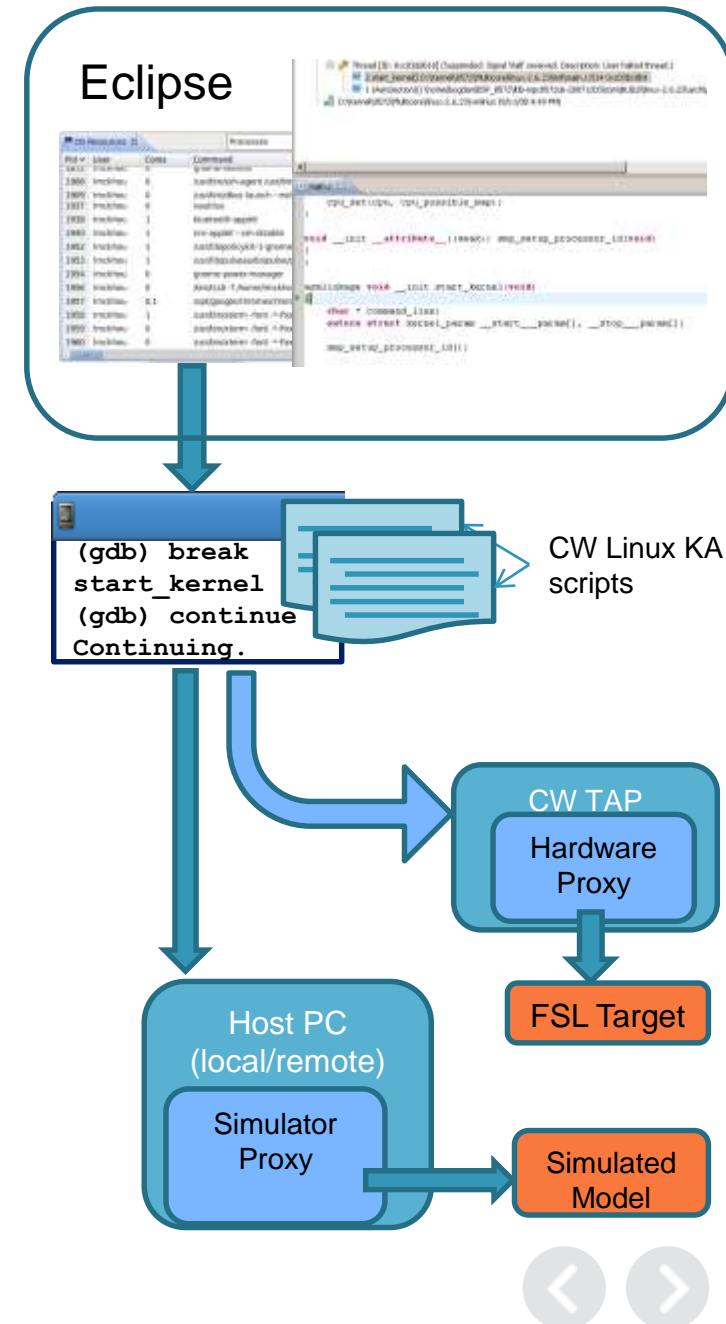
Index	Source	Type	Description	Address	Destination	Timestamp
1	Core 0	Info	SYNC packet - ETM			0
2	Core 0	Info	Trace On packet - ETM -> start tracing after a...			0
3	Core 0	Info	Context packet - ETM exception level: 3 state: 64-bit security state: secure ContextID: 0			0
4	Core 0	Software Context	software context id = 0			0
5	DDDI	Custom	Port: DDDI1 Verbose mode Transaction source = GPP0 Transaction type: Read Transaction size = 512 bits DDR EC bits = 0x1 Address = 0x07ff1b200			1903481
6	DDDI	Custom	Port: DDDI1			1903483
7	DDDI	Custom	Port: DDDI1			1903487
8	DDDI	Custom	Port: DDDI1			1903514

Linux Kernel Debug



Linux Kernel Awareness

- Linux Kernel Awareness features kernel threads information
 - Kernel modules list
 - Kernel Threads list
 - MMU Awareness
 - Kernel Module Debug, module insert/remove detection
- Available from Eclipse UI and command line in debugger console



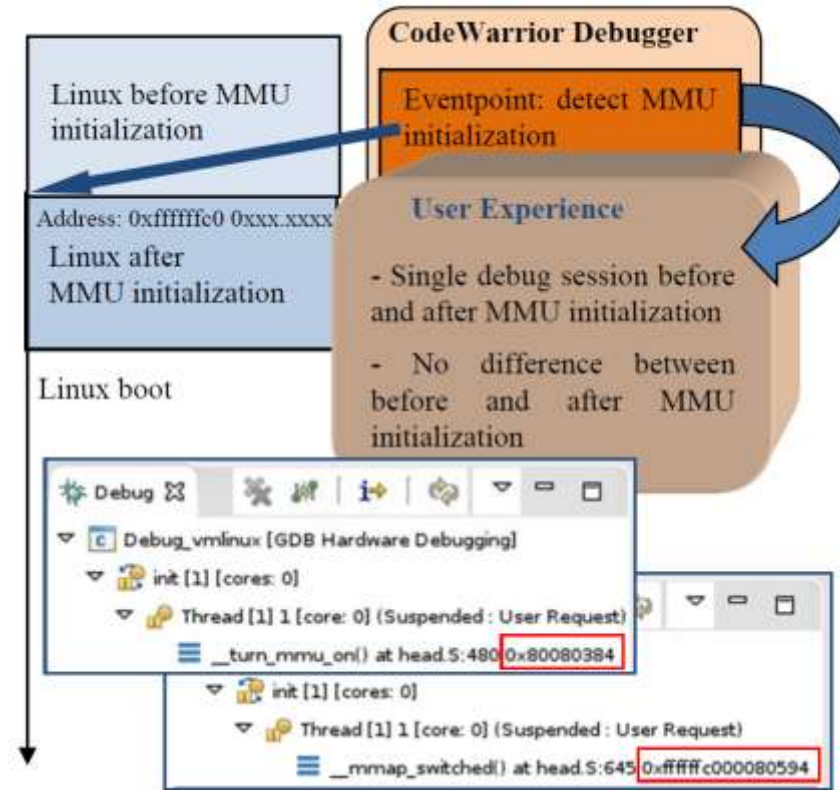
Linux kernel debug – Capabilities

- **Full Linux debugger**
 - View program's source code (C/C++ or disassembly), memory, registers, stack frames, variables, etc.
 - Breakpoints, run control
- **No kernel changes** required
- **Multicore** debugging
- **MMU awareness**
- **System information** display (kernel info, per core threads list, kernel modules list)
- Loadable **kernel modules debug**
- **Scenarios** supported
 - **Attach** to a running Linux kernel
 - **Attach** to u-boot & start Linux from u-boot

Linux kernel debug – Capabilities

- MMU awareness

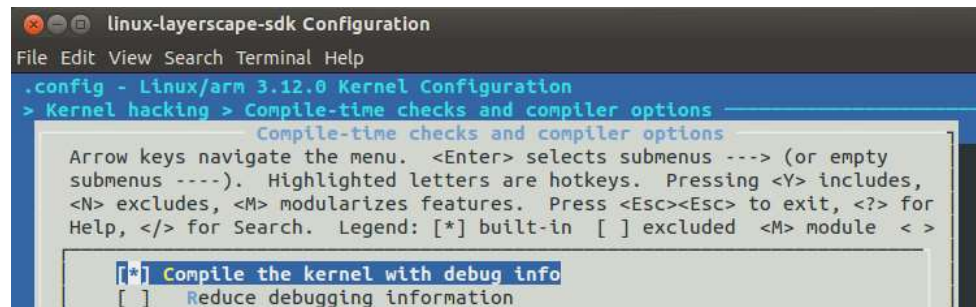
- Detects the point where MMU initialization is done, computes and applies the relocation offset. The user is not aware of two debug configuration settings: before or after MMU initialization.
- There is no difference between debugging before and after MMU initialization
- No special action is required when moving before and after MMU initialization during the same debug session
- No need for the user to know the current MMU initialization state and to manually apply the relocation offset



Linux kernel debug – Prerequisites – Activity

- LS2085A SDK

- <http://linux.freescale.net/labDownload2/viewDownloads.php?Filter=LS2085A&field=PL>
- Download and install (SDK EAR4.0)
 - Layerscape2-SDK-SOURCE-20150515-yocto.iso
- Configure Linux kernel; most important thing: **debug symbols**
 - bitbake -c menuconfig virtual/kernel
 - Go to **Kernel hacking** → **Compile-time checks and compiler options** and check **Compile the kernel with debug info**



```
linux-layerscape-sdk Configuration
File Edit View Search Terminal Help
.config - Linux/arm 3.12.0 Kernel Configuration
> Kernel hacking > Compile-time checks and compiler options
  Compile-time checks and compiler options
  Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
  submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes,
  <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for
  Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module <>
  [*] Compile the kernel with debug info
  [ ] Reduce debugging information
```

- Build image
 - bitbake fsl-image-kernelitb

Linux kernel debug – Prerequisites –Activity (cont)

- LS2085A SDK

- Build Linux kernel
 - **bitbake virtual/kernel**
- Get the images (paths relative to <YoctoInstallationPath>)
 - **u-boot**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/u-boot-ls2085aqds_config.bin
 - **ulmage**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/ulmage-LS2085ARDB.bin
 - **DTB (device tree)**:
build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/ulmage-LS2085ARDB.dtb
 - **Ramdisk**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/fsl-image-core-LS2085ARDB.ext2.gz.u-boot
 - **vmlinux (in sync with ulmage)**:
build_ls2085ardb_release/tmp/work/ls2085ardb-fsl-linux/linux-ls2-sdk/3.19-r0/git/vmlinux
 - **ITB Image**:
build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/kernel-ls2085ardb.itb

Activity

Linux kernel debug



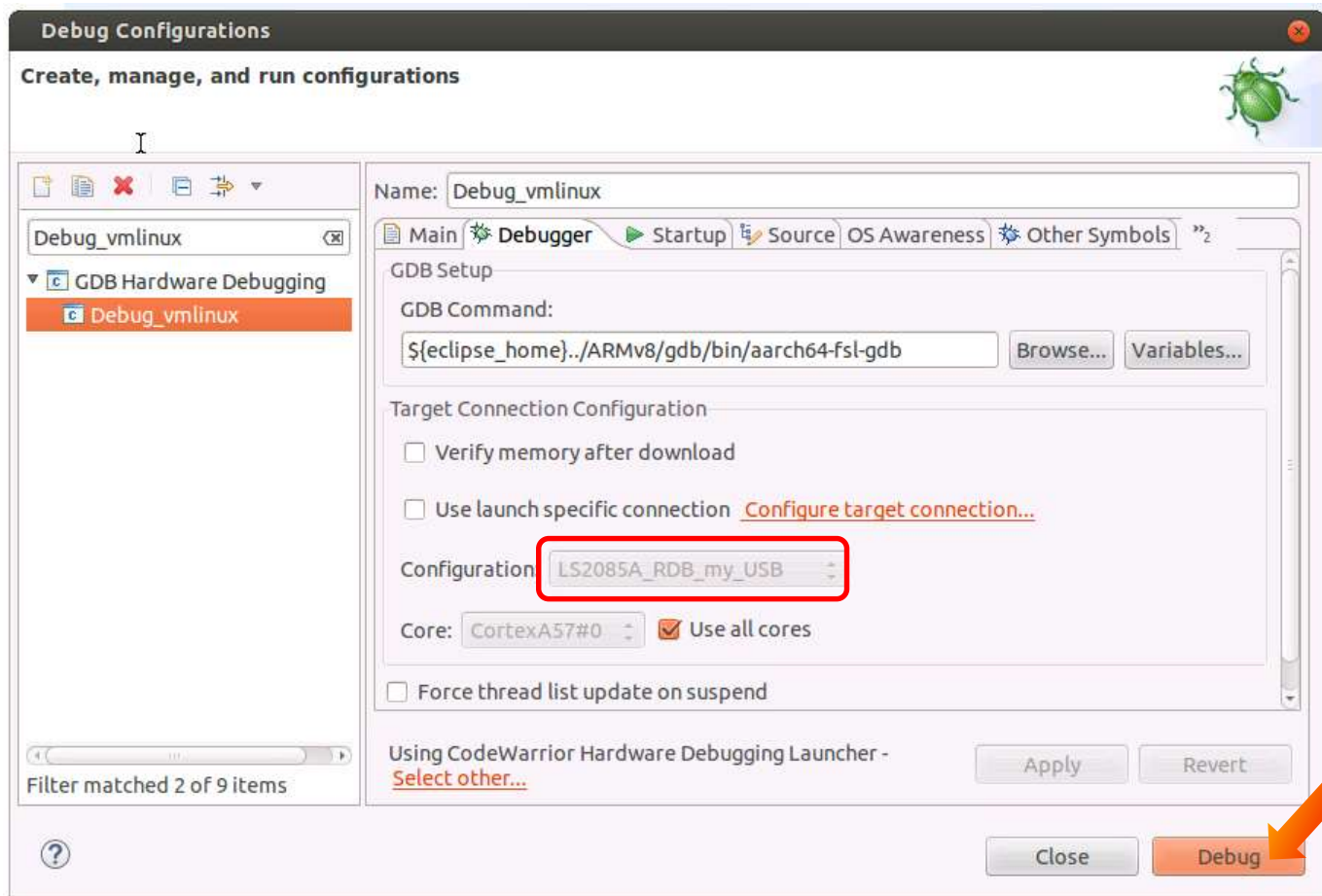
Linux kernel debug – Create project – Activity

- Select File > New > CodeWarrior ELF Importer
 - CodeWarrior Elf importer – automatically detects the ELF type and applies the correct awareness settings
- Select vmlinux file. CodeWarrior automatically:
 - Detect the ELF type as Linux Kernel
 - Show Linux Kernel summary information: version, build time
 - Apply the right debugger settings for debugging Linux Kernel

The screenshot shows the CodeWarrior ELF Importer dialog box. The title bar reads "CodeWarrior ELF Importer". The main text says "Import CodeWarrior executable files, type is auto-detected". Below this, it states "CodeWarrior auto-detected a Linux Kernel executable" followed by the version and build information: "3.19.3-Layerscape2-SDK+g9f41bb6; #1 SMP PREEMPT Fri May 15". The "Select executable:" field contains the path "/home/class/SDK/binary/vmlinux". Under "Select type:", the "Linux Kernel" radio button is selected. The "Finish" button is highlighted with an orange arrow.

Linux kernel debug – Create project – Activity

- The Debug Configuration for Linux Kernel project automatically created
 - Linux kernel awareness settings applied
 - Default Target connection applied: LS2085A_RDB_my_USB
 - To create new connection: see u-boot debug create Target Connection



Linux kernel debug – Prepare u-boot for booting Linux – Activity (cont)

- Setup the **LS2085A RDB board**
- Setup a **TFTP server** (e.g. using TPFTP32 on Windows)
- Make sure **u-boot** is correctly booting (use Flash Programmer to flash it in NOR)
- Setup networking in u-boot for booting Linux via **TFTP**
 - Set ENV variables (**ipaddr**, **netmask**, **gatewayip**, **serverip**)
- Setup loading addresses for the images
 - **loadaddr** → **0xa0000000**
- Set paths to files loaded via TFTP (example below)
 - setenv **imagefile** ls2/kernel-ls2085ardb.itb
- Set the command to boot Linux (example below)
 - setenv **bootLinux** 'setenv bootargs console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600,115200 default_hugepagesz=2m hugepagesz=2m hugepages=16 ramdisk_size=0x8000000; \$othbootargs;tftp \$loadaddr \$imagefile; **bootm** \$loadaddr'
- To **boot Linux**:
 - **boot** – boot Linux from flash
 - **run bootLinux** – boot Linux from TFTP

Linux kernel debug – Attach with CodeWarrior to u-boot – Activity (cont)

- Reset the LS2 RDB board so that **u-boot prompt** is displayed
- From CodeWarrior **Run** → **Debug Configurations**, select the Linux Debug configuration and hit **Debug**

No symbols available as while being in u-boot, the Linux has not been started yet

U-boot console

The screenshot shows the CodeWarrior IDE interface during a Linux debug session. The main window displays the u-boot console output, including SoC information (LS2085E), clock configuration (1600 MHz for all CPUs), and reset configuration words. A disassembly window is open, showing instructions at memory address 0xffff5a8f4. A callout box points to the disassembly window with the text "No symbols available for '0xffff5a8f4'". Another callout box points to the console window with the text "U-boot console".

```
mc [b32721@udp122517uds]~/Freescale/CW4NET_v2015.05
U-Boot 2015.01Layerscape2-SDK+g16c10aa (May 15 2015 - 11:49:32)

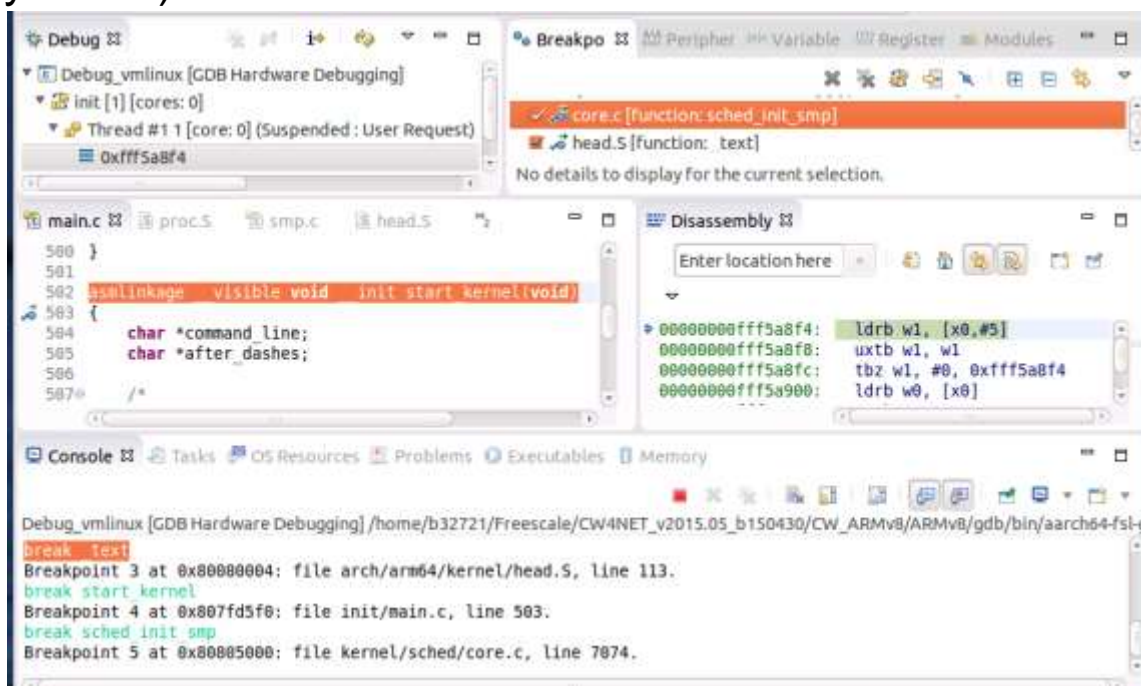
SoC: LS2085E (0x87010010)
Clock Configuration:
CPU0(A57):1600 MHz CPU1(A57):1600 MHz CPU2(A57):1600 MHz
CPU3(A57):1600 MHz CPU4(A57):1600 MHz CPU5(A57):1600 MHz
CPU6(A57):1600 MHz CPU7(A57):1600 MHz
Bus: 600 MHz DDR: 1333.333 MT/s DP-DDR:

/s
Reset Configuration Word (RCW):
00: 40282830 40400040 00000000 00000000
10: 00000000 00000000 00200000 00000000
20: 00c12980 00002580 00000000 00000000
30: 00000003 00000000 00000000 00000000
40: 00000000 00000000 00000000 00000000
```



Linux kernel debug – Debug the Linux kernel from the entry point – Activity (cont)

- To demonstrate full Linux Kernel debug in a single session, set several breakpoints in some key points of booting process
- Set breakpoints from console or from file:
 - `_text`: Linux Kernel entry point (before MMU initialization)
 - `start_kernel`: kernel setup (after MMU initialization)
 - `sched_init_smp`: SMP initialization (after Linux Kernel setup the secondary cores)



Linux kernel debug – Debug the Linux kernel from the entry point – Activity (cont)

- Entry point breakpoints hit
- Note: breakpoint set as SW breakpoint (default) to a memory address before u-boot copy the ulmage into DDRAM. Linux Kernel Awareness automatically changes the breakpoint type to HW

Before Linux SMP initialization:
Only one core is shown in the
Debug view

Linux Kernel Awareness
automatically apply the symbol
relocation corresponding to
mapping before MMU
initialization

Console output stopped at
Starting kernel...

The screenshot shows the GDB interface for debugging the Linux kernel. The 'Debug' window displays the thread list for 'Debug_vmlinux [GDB Hardware Debugging]', showing 'init [1] [cores: 0]' and 'Thread #1 1 [core: 0] (Suspended: User Request)'. The current thread is at 'efi_head() at head.S:113' with a memory address of '0x80080004' highlighted in red. The 'Breakpo' window shows breakpoints for 'core.c [function: scned_init_smp]', 'head.S [function: _text]', and 'main.c [function: start_kernel]'. The source code editor shows the assembly code for 'head.S', with line 113 highlighted. The 'Disassembly' window shows the disassembly of the instruction at '0x80081000', which is 'b 0x80081000 <stext>'. The 'Console' window shows the boot process output, which has stopped at 'Starting kernel ...'.



Linux kernel debug – Debug the Linux kernel from the entry point – Activity (cont)

- Start Kernel breakpoints hit
- Using an internal eventpoint, Linux Kernel Awareness detect the point where the Kernel initialize the MMU and apply the corresponding settings

Before Linux SMP initialization:
Only one core is shown in the
Debug view

Linux Kernel Awareness
automatically apply the symbol
relocation corresponding to
mapping after MMU
initialization

The screenshot displays the GDB interface for debugging the Linux kernel. The 'Debug' window shows a single thread 'Thread #1 1 [core: 0] (Suspended: User Request)' at the address '0xffff8000007fd5f0'. The 'Breakpo' window shows breakpoints for 'core.c [function: sched_init_smp]', 'head.S [function: _text]', and 'main.c [function: start_kernel]'. The 'Disassembly' window shows assembly code for 'start_kernel' with instructions like 'stp x29, x30, [sp, #-80]!' and 'set_task_stack_end_magic'. The source code window shows 'asmlinkage visible void init start kernel(void)' and 'char *command_line;'.

Linux kernel debug – Debug the Linux kernel from the entry point – Activity (cont)

- SMP initialization breakpoints hit
- Using an internal eventpoint, Linux Kernel Awareness detect the point where the Kernel setup the secondary cores

After Linux SMP initialization:
8 cores are shown in the
Debug view

The screenshot displays the GDB interface during Linux kernel debugging. The 'Debug' window shows a list of threads for 8 cores (0-7), all in a 'Suspended: Container' state. The 'Breakpoint' window shows the kernel boot logs, including messages for CPU4 through CPU7, indicating that 8 processors were activated. The 'core.c' window shows the source code for the 'init_sched_init_smp' function, and the 'Disassembly' window shows the corresponding assembly instructions.

```
void init_sched_init_smp(void)
{
    cpumask_var_t non_isolated_cpus;
    alloc_cpumask_var(&non_isolated_cpus, GFP_KERNEL);
    alloc_cpumask_var(&fallback_doms, GFP_KERNEL);
    sched_init_numa();
}
```

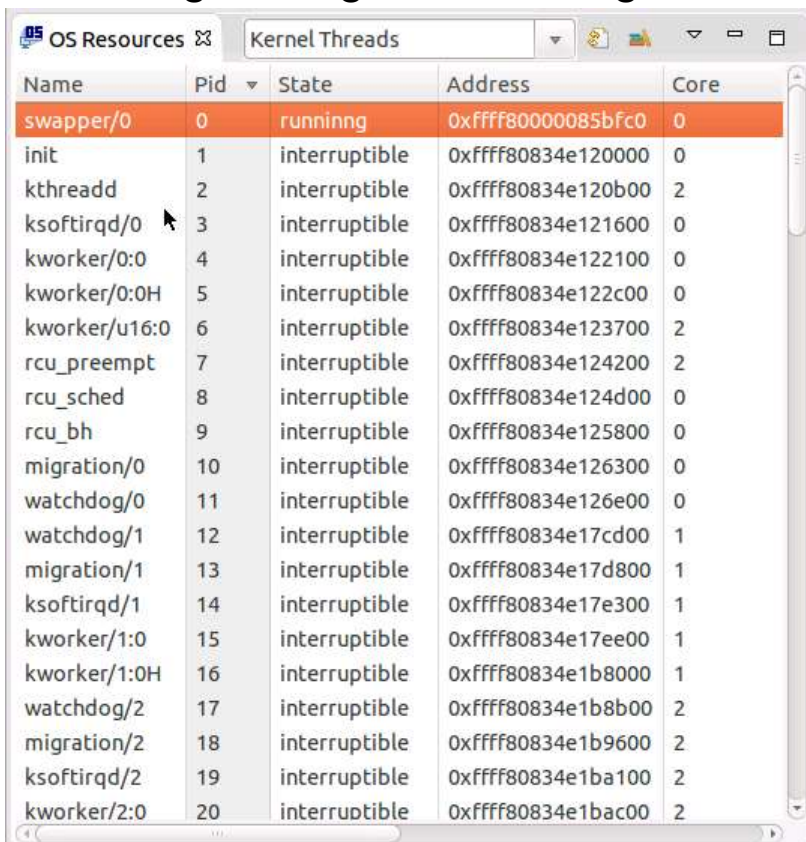
```
stp x29, x30, [sp,#-64]!
mov x29, sp
stp x19, x20, [sp,#16]
mutex_lock(&sched_domains_mu)
adrp x19, 0xffff800008660000
add x0, x19, #0x460
add x0, x0, #0xb30
```



Linux kernel debug – Linux kernel Information– Activity (cont)

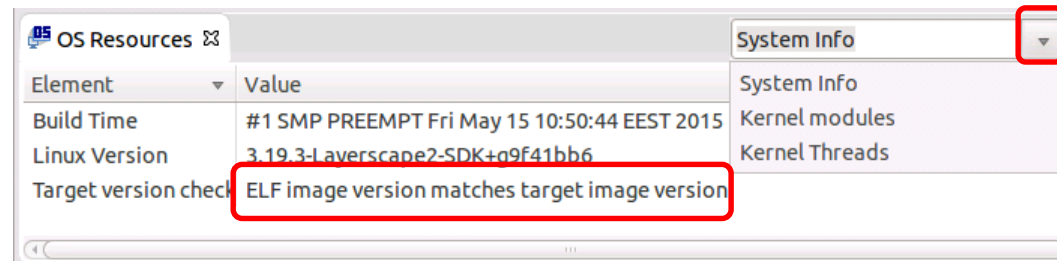
OS Resources: Windows > Show View > Other > Debugger > OS Resources

- Visualize meaningful Linux Kernel information about: Linux Kernel version, build time, Linux Kernel Thread, Linux kernel Modules
- Target image vs Elf image version check



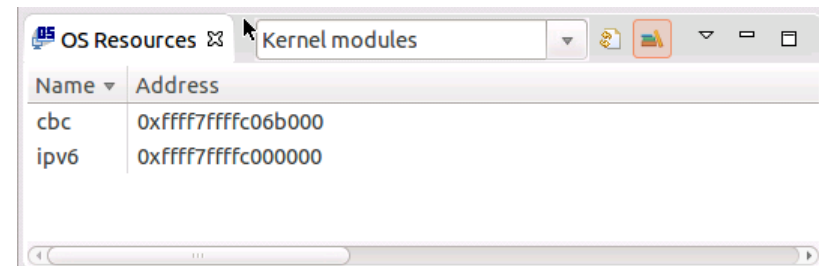
The screenshot shows the 'OS Resources' window with the 'Kernel Threads' view selected. It displays a list of kernel threads with columns for Name, Pid, State, Address, and Core. The 'swapper/0' thread is highlighted in orange.

Name	Pid	State	Address	Core
swapper/0	0	running	0xffff80000085bfc0	0
init	1	interruptible	0xffff80834e120000	0
kthreadd	2	interruptible	0xffff80834e120b00	2
ksoftirqd/0	3	interruptible	0xffff80834e121600	0
kworker/0:0	4	interruptible	0xffff80834e122100	0
kworker/0:0H	5	interruptible	0xffff80834e122c00	0
kworker/u16:0	6	interruptible	0xffff80834e123700	2
rcu_preempt	7	interruptible	0xffff80834e124200	2
rcu_sched	8	interruptible	0xffff80834e124d00	0
rcu_bh	9	interruptible	0xffff80834e125800	0
migration/0	10	interruptible	0xffff80834e126300	0
watchdog/0	11	interruptible	0xffff80834e126e00	0
watchdog/1	12	interruptible	0xffff80834e17cd00	1
migration/1	13	interruptible	0xffff80834e17d800	1
ksoftirqd/1	14	interruptible	0xffff80834e17e300	1
kworker/1:0	15	interruptible	0xffff80834e17ee00	1
kworker/1:0H	16	interruptible	0xffff80834e1b8000	1
watchdog/2	17	interruptible	0xffff80834e1b8b00	2
migration/2	18	interruptible	0xffff80834e1b9600	2
ksoftirqd/2	19	interruptible	0xffff80834e1ba100	2
kworker/2:0	20	interruptible	0xffff80834e1bac00	2



The screenshot shows the 'OS Resources' window with the 'System Info' view selected. It displays a table of system information. The 'Target version check' row is highlighted with a red box, showing 'ELF image version matches target image version'.

Element	Value
Build Time	#1 SMP PREEMPT Fri May 15 10:50:44 EEST 2015
Linux Version	3.19.3-Layerscape2-SDK+q9f41bb6
Target version check	ELF image version matches target image version

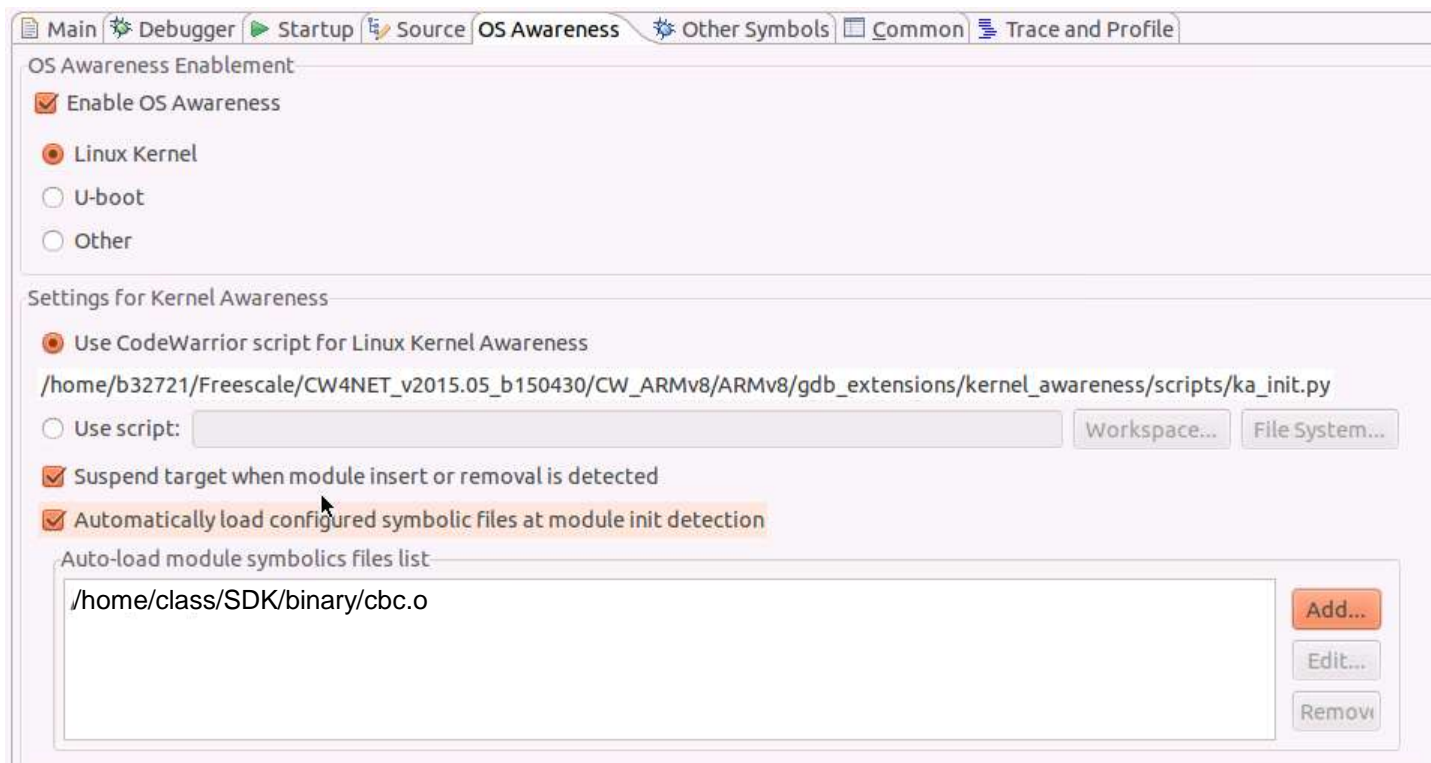


The screenshot shows the 'OS Resources' window with the 'Kernel modules' view selected. It displays a table of kernel modules with columns for Name and Address.

Name	Address
cbc	0xffff7ffffc06b000
ipv6	0xffff7ffffc000000

Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- Configure Kernel module debug parameters: Debug configuration -> OS Awareness tab
 - Check “Suspend target when module insert or removal is detected”
 - Check “Automatically load configured symbolic files at module init detection”.
 - Add the module symbolics files
- Note:** you can load the symbolics file also after attaching to linux kernel



Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- On the Linux console execute **modprobe cbc**
- Target will stop in load_module->do_init_module function
- The symbolics are automatically loaded if configured so
- Or the symbolics can be loaded at anytime after module insertion: in OS Resources->Kernel Module->Module Management Button

The screenshot displays a GDB debugging session for a Linux kernel module. The 'Debug' window shows the call stack with the following entries:

- Thread #1 1 [core: 0] (Suspended: Container)
- Thread #2 2 [core: 1] (Suspended: Container)
- Thread #3 3 [core: 2] (Suspended: Breakpoint)
- load_module() at module.c:3,351 0xffff800000113148
- SYSC_finit_module() at module.c:3,427 0xffff8000001136c8
- SyS_finit_module() at module.c:3,408 0xffff8000001136c8

The 'OS Resources' window shows the 'Kernel modules' section with the following table:

Name	Address
cbc	0xffff7ffffc06b000 <crypto_cbc_setkey>
ipv6	0xffff7ffffc000000

The terminal window shows the command `modprobe cbc` being executed. An orange callout bubble points to the terminal with the text "Insert Module".

The disassembler window shows the following assembly code for the `do_init_module` function:

```
ffff80000011314c: bl 0xffff800000169e24 <vfree>
ffff800000113150: ldr x0, [x23,#48]
ffff800000113154: mov w1, #0xd0
ffff800000113158: bl 0xffff80000017e9ec <kmem_cache_grow>
ffff80000011315c: mov x20, x0
                if (!freemem) f
```

Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- The symbolics can be loaded at anytime after module insertion:
 - in OS Resources->Kernel Module->Modules Management Button

The screenshot shows the OS Resources window with the 'Kernel modules' tab selected. A table lists loaded modules:

Name	Address
cbc	0xffff7ffffc06b000 <crypto_cbc_setkey>
ipv6	0xffff7ffffc000000

An orange callout box with the number '1' points to the 'Modules Management' button in the top right corner of the OS Resources window. The callout text reads: "The Modules Management allows operations for the loaded modules".

The 'Module Management' dialog is open, showing 'Loaded modules operations'. It contains a table with the following data:

Module	Symbols path
cbc	/home/class/SDK/binary/cbc.o
ipv6	No symbolics file loaded

Buttons for 'Load symbolics...' and 'Unload symbolics' are visible. A 'Select module symbolics file' dialog is also open, prompting to 'Add symbolics file for "ipv6" module?'. The file path '/home/class/SDK/binary/ipv6.o' is entered in the text field. Buttons for 'Workspace...', 'File System...', 'Cancel', and 'OK' are present.

Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- Set breakpoints in the module files (from file or from console)
- Continue to hit the breakpoints

The screenshot displays a debugger window with several panels:

- Thread:** Shows three threads, with Thread #33 suspended at a breakpoint.
- Break:** Lists two breakpoints: `cbc.c [function: crypto_cbc_module_init]` and `cbc.c [line: 284]`.
- Disassembly:** Shows assembly instructions starting at `ffff80000113148: add x23, x23, #0xc78`.
- Console:** Shows the command `break crypto cbc module init` and the response `Breakpoint 1 at 0xffff7ffffc06d000: file crypto/cbc.c, line 278.`
- OS Resources:** Lists kernel modules: `cbc` at `0xffff7ffffc06b000` and `ipv6` at `0xffff7ffffc000000`.

Source code in the `cbc.c` file:

```
276
277 static int __init crypto_cbc_module_init(void)
278 {
279     return crypto_register_template(&crypto_cbc_tmpl);
280 }
281
282 static void __exit crypto_cbc_module_exit(void)
283 {
284     crypto_unregister_template(&crypto_cbc_tmpl);
```

Double-click to set breakpoint

Set breakpoint from console



Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- Target stops when breakpoints are hit
- Perform debug as usual
- Continue and remove the module from linux console

The screenshot displays the GDB debugger interface with several panels:

- Debug:** Shows a list of threads. Thread #33 [core: 2] is suspended at a breakpoint in `crypto_cbc_module_init()` at `cbc.c:278`. Other threads are suspended in container or main.c.
- Break:** Shows a breakpoint set at `cbc.c [function: crypto_cbc_module_init]` and `cbc.c [line: 284]`.
- Source:** Shows the source code for `cbc.c`. Line 278 is highlighted, showing the start of the `crypto_cbc_module_init` function.
- Disassembly:** Shows the assembly code for `crypto_cbc_module_init`. The instruction `stp x29, x30, [sp, #-16]!` is highlighted.
- Console:** Shows the GDB output: `Debug_vmlinux [GDB Hardware Debugging] /home/b32721/Freescale/CW4NET_v2015.05_b150430/`, `Breakpoint 1 at 0xffff7ffffc06d000: file crypto/cbc.c, line 278.`, and `info break` output showing two breakpoints.
- Kernel modules:** Shows a list of loaded kernel modules: `cbc` at `0xffff7ffffc06b000 <crypto_cbc_setkey>` and `ipv6` at `0xffff7ffffc000000`.



Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- Target stops at module exit
- Perform debug as usual

The screenshot shows a debugger interface with the following components:

- Debug Window:** Shows three threads. Thread #44 [core: 3] is suspended at a breakpoint in `crypto_cbc_module_exit()` at `cbc.c:284`. Below it, `SYSC_delete_module()` and `Sys_delete_module()` are also visible.
- Breakpoints Window:** Shows two breakpoints: `cbc.c [function: crypto_cbc_module_init]` and `cbc.c [line: 284]`.
- Source Window:** Shows the source code for `cbc.c`. Line 284, `crypto_unregister_template(&crypto_cbc_tm)`, is highlighted.
- Console Window:** Shows OS resources for 'cbc' (Address: `0xfffff7ffffc06b000`) and 'ipv6' (Address: `0xfffff7ffffc000000`).
- Terminal Window:** Shows a shell prompt `root@ls2085aqds:~#` with the following commands:

```
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~#  
root@ls2085aqds:~# insmod cbc.ko  
root@ls2085aqds:~# rmmod cbc.ko
```

An orange callout bubble with the text "Remove the Module" points to the `rmmod cbc.ko` command.

Linux Kernel Trace



Linux Tools – LTTng

- Linux Trace Toolkit – next generation: kernel and user-space tracer with view and analysis tools.
- LTTNG has been separated out of the Linux Trace Toolkit. Now a separate project called Trace Compass.
 - <http://projects.eclipse.org/projects/tools.tracecompass>

LTTNG

- Trace Compass is a Eclipse tool for viewing and analyzing any type of logs or traces.
 - Provide views, graphs, metrics, etc. to help extract useful information from traces, in a way that is more user-friendly and informative than huge text dumps
- Eclipse: “LTTng Kernel” perspective
- View the results
 - Events: timestamp, trace, Marker, Content
 - Histogram: trace event distribution in time
 - Control flow: processes list and their state in time
 - Resources: CPU resources per interrupts type
 - Statistics: event counters cpu time, cumulative /elapsed time
- Import or create a LTTng trace

Traces / Logs

- Trace Compass supports many trace formats:
 - [Common Trace Format \(CTF\)](#), including but not limited to:
 - Linux [LTTng](#) kernel traces
 - Linux [LTTng-UST](#) userspace traces
 - Linux Perf traces (using the out-of-tree [patchset](#) to convert to CTF)
 - [GDB traces](#) for debugging
 - The [libpcap](#) (PACket CAPture) format, for network traces

Linux Trace

- Static probe points strategically located inside the kernel code
- Register/unregister with tracepoints via callback mechanism
- Can be used to profile, debug and understand kernel behavior

- Trace synchronization
 - Time correction
 - Multi-core
 - Dependency analysis, delay analyzer
 - Dependencies among processes

Activity

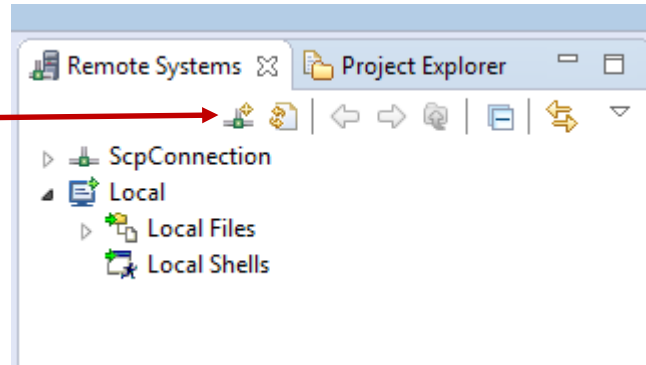
Trace Compass



Trace Compass – new RSE connection

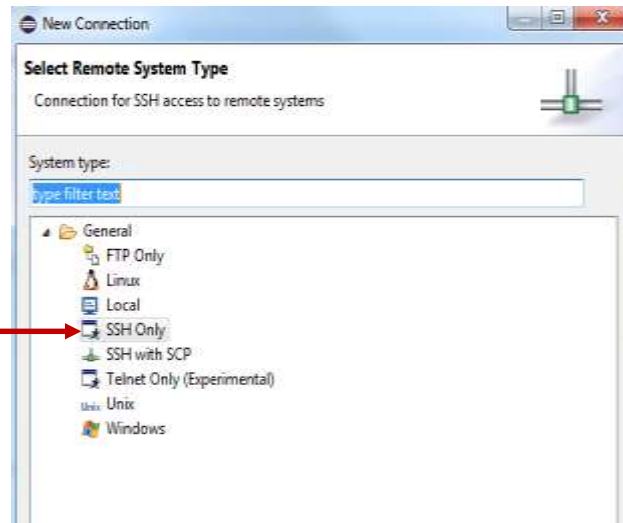
1. Open Remote Systems view (Window->Show View->Other->Remote Systems->Remote Systems)

New RSE connection



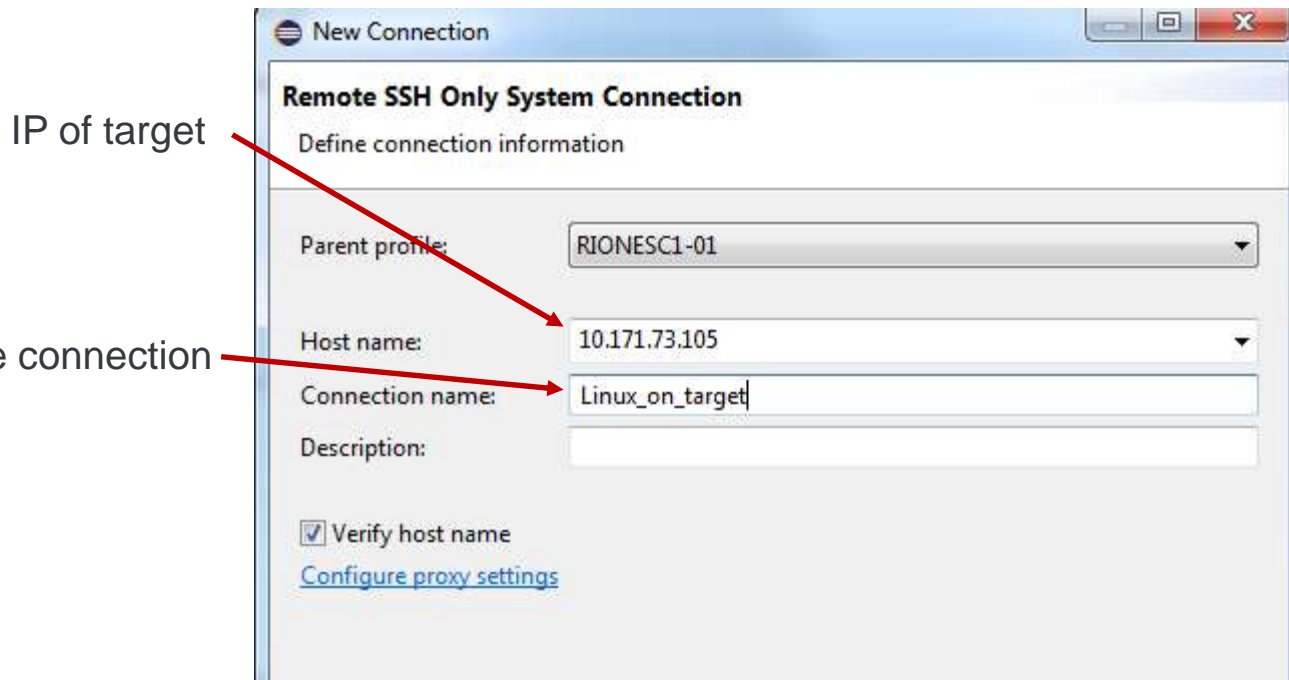
2. Create a Linux based RSE connection

SSH Only



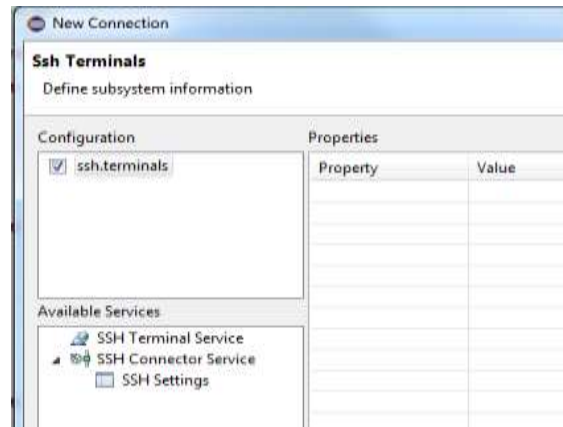
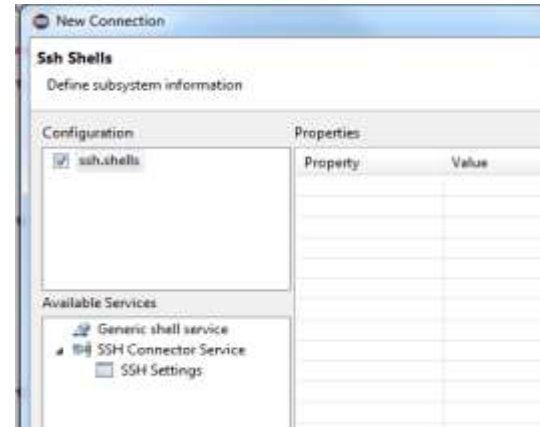
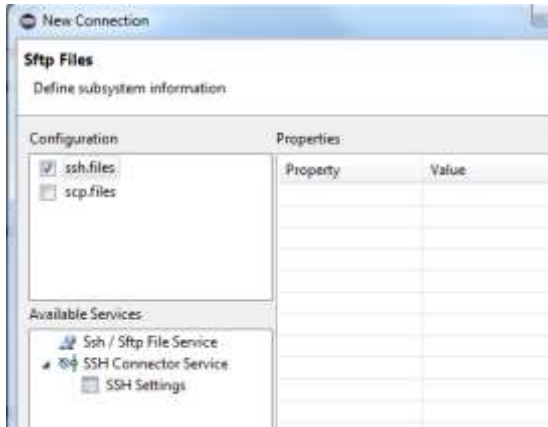
Trace Compass – new RSE connection

3. Follow the steps to create the RSE connection over SSH



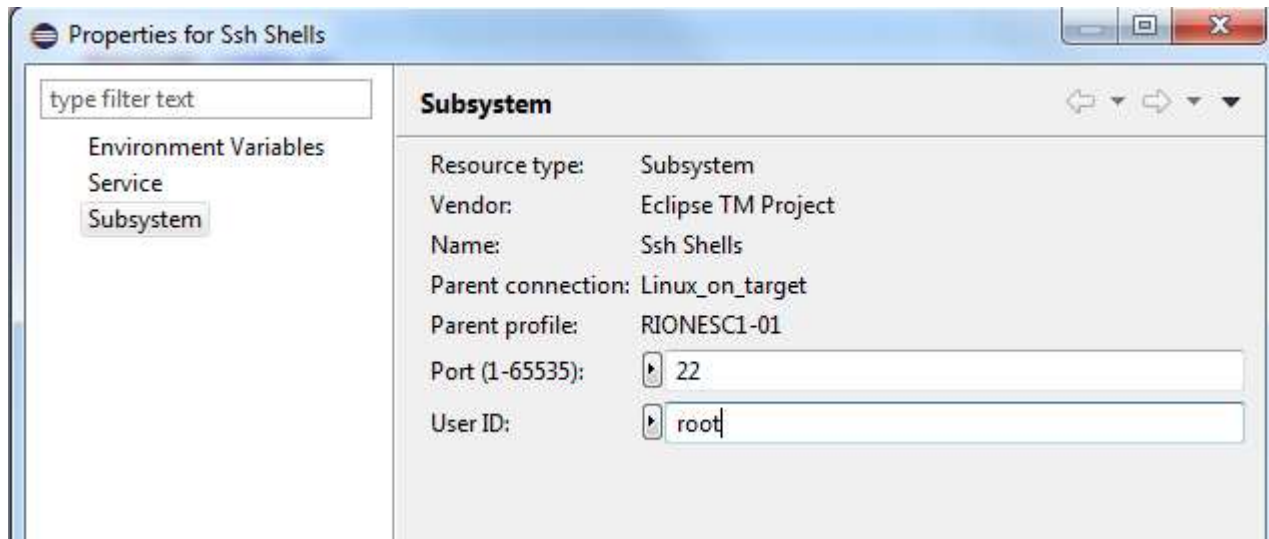
Trace Compass – new RSE connection

4. Continue to follow RSE connection creation wizard



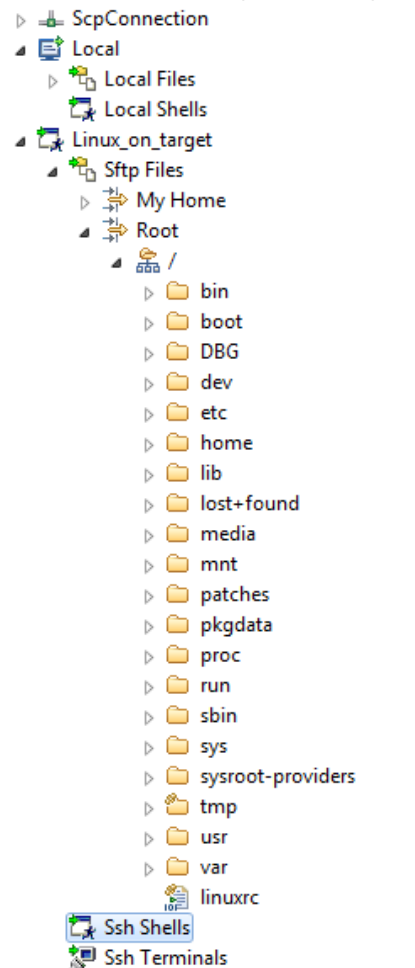
Trace Compass – new RSE connection

5. Right-click on Ssh Shells -> Properties -> Subsystem. Verify the port (default is 22; change if port is forward). Set *root* as user ID.



Trace Compass – new RSE connection

6. Now expand the Sftp Files node and you will be able to browse the target file system:



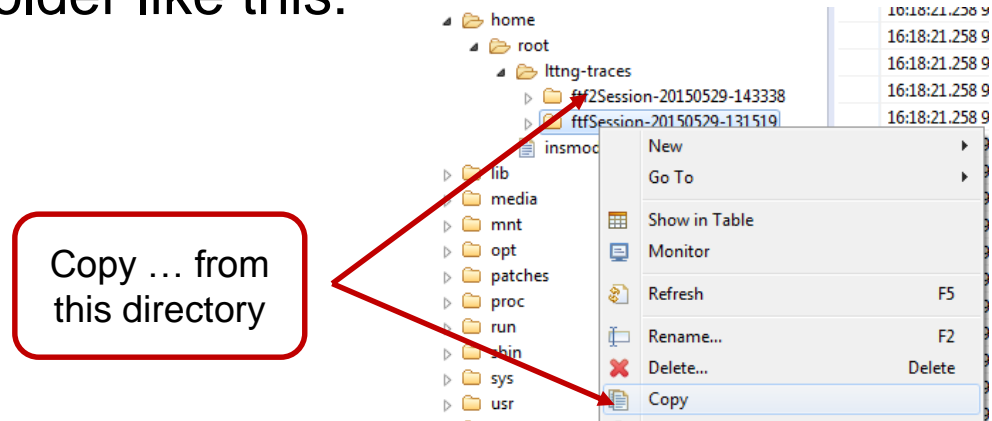
Trace Compass – trace session

1. Open a Terminal over a RSE connection from CodeWarrior
2. Load LTTng modules:
`modprobe lttng-tracer`
3. Check that LTTng modules are loaded:
`lsmod`
4. Create a new LTTng session:
`lttng create ftfSession`
5. Enable all events for Kernel tracing:
`lttng enable-event --kernel --all`
6. Start tracing session:
`lttng start`
7. Run some applications (e.g., ls, top)
8. Stop tracing session:
`lttng stop`
9. Destroy session:
`lttng destroy`

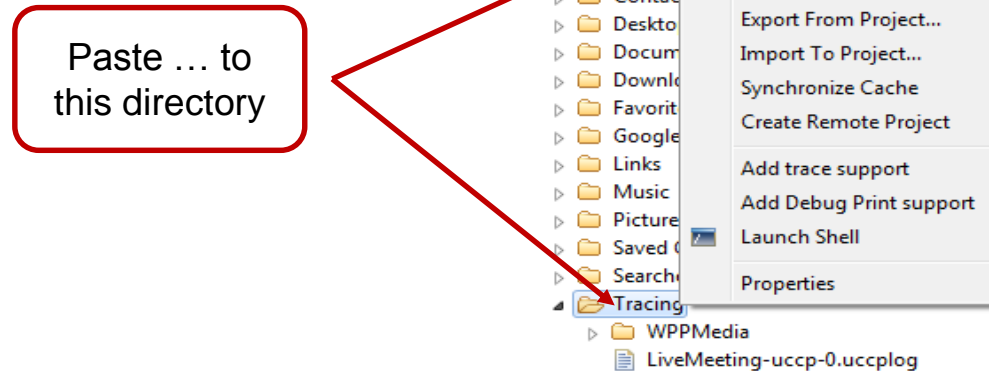
Trace Compass – trace session

10. Notice the newly created folder in your home dir (*ltnng-traces*)

11. Copy the session folder like this:

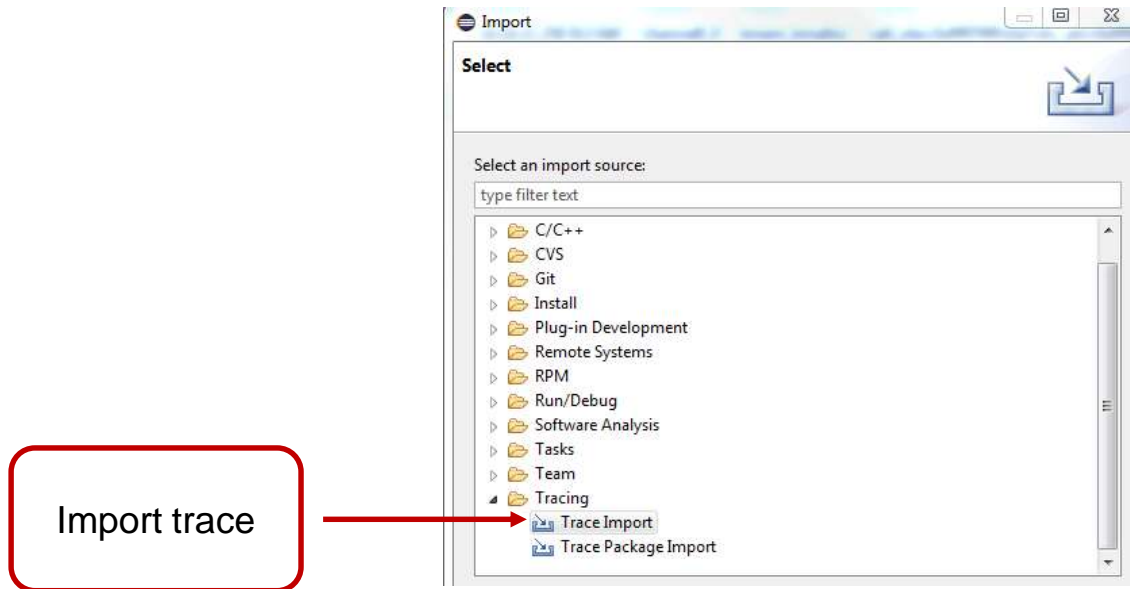


12. Paste in Local node (RSE):



Trace Compass – trace session

13. Open *Project Explorer* view
14. Right-click and choose *Import*
15. Select *Tracing->Trace Import*

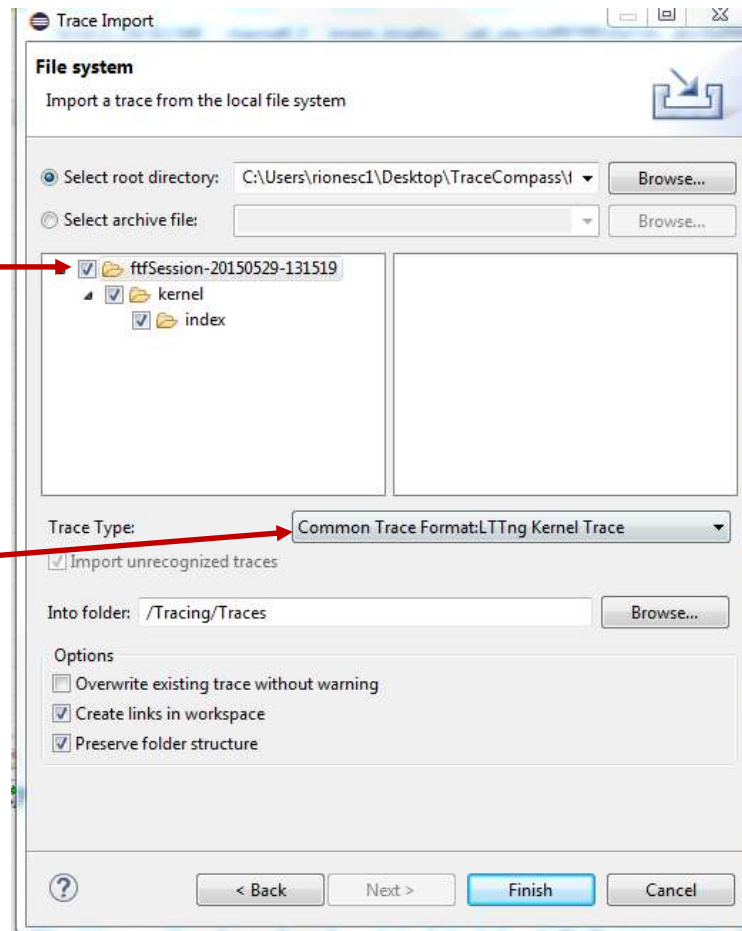


Trace Compass – trace session

16. Choose the copied folder with trace session; check the file to import; select *Trace Type* as *LTTng Kernel Trace*

Select session file

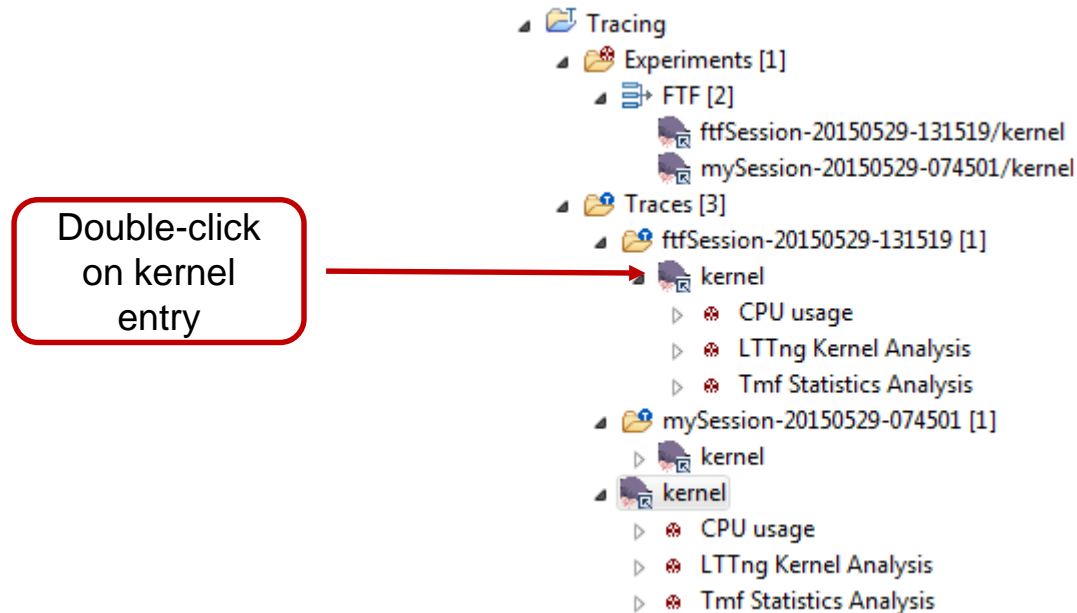
Select Trace Type



Trace Compass – trace session

17. Open *LTTng Kernel* perspective

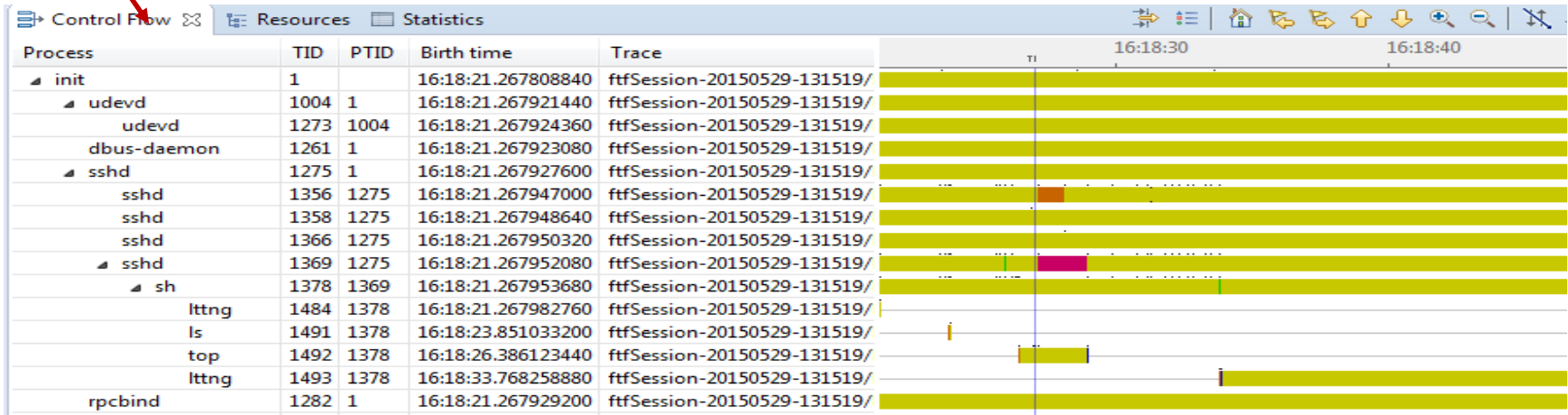
18. Double-click on imported trace session from *Project Explorer* view (kernel entry):



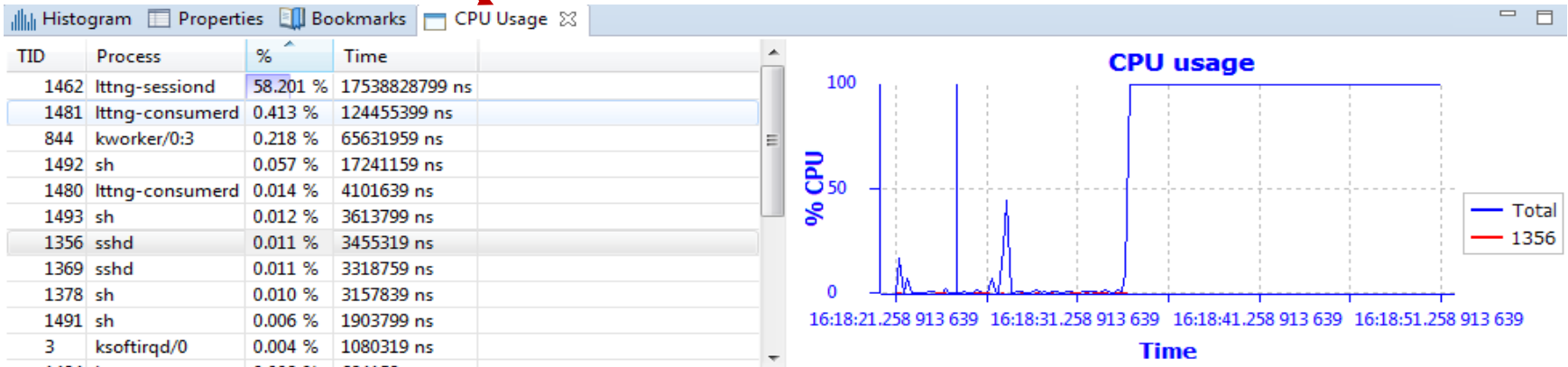
19. Various views will open and you can explore trace results

Trace Compass views

Control Flow

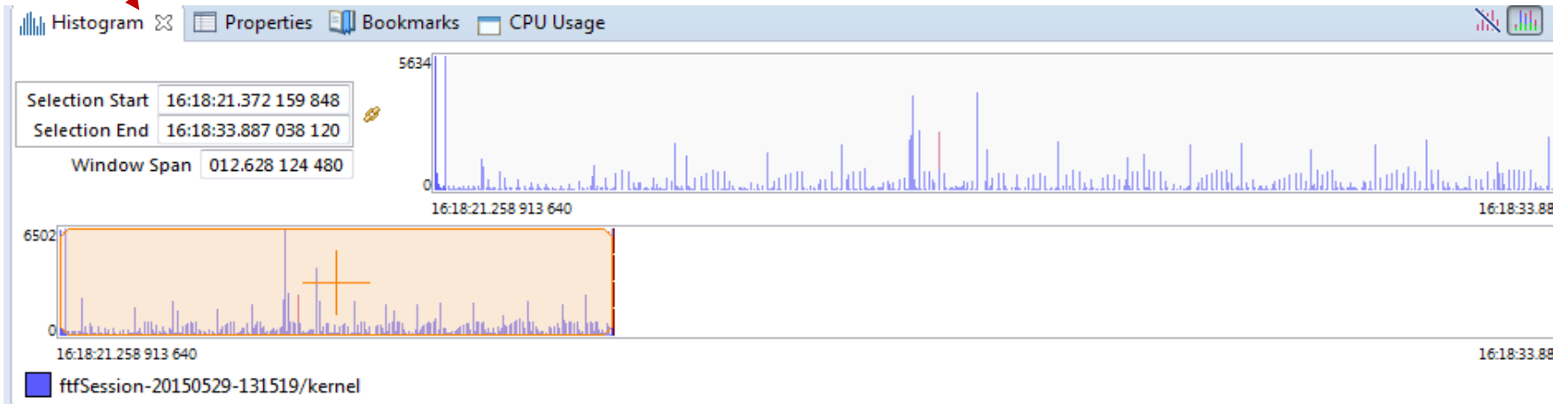


CPU Usage



Trace Compass views

Histogram

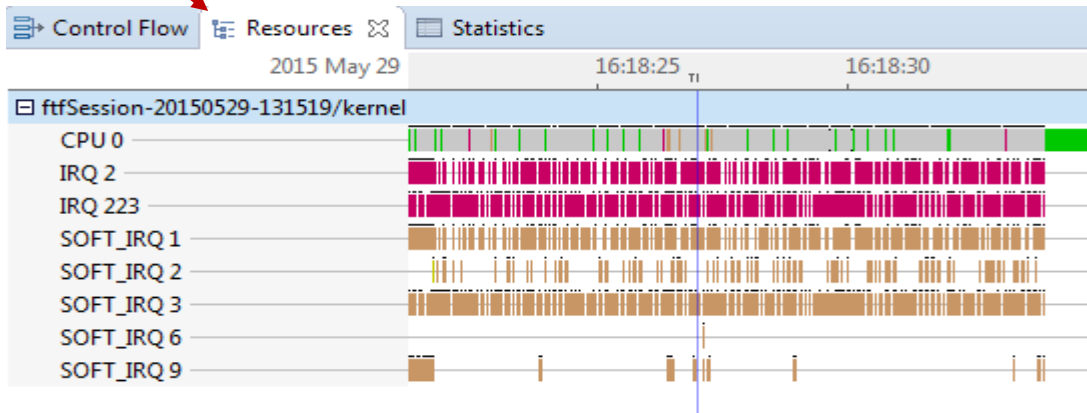


Events

Timestamp	Channel	Type	Content
<srch>	<srch>	<srch>	<srch>
16:18:31.412 262 880	channel0_0	kmem_cache...	call_site=0xffff8000002fd250, ptr=0xffff80832fe6e380, bytes_req=128, bytes_alloc=128, gfp_flags=3...
16:18:31.412 263 560	channel0_0	kmem_kfree	call_site=0xffff8000002f4b0c, ptr=0x0
16:18:31.412 264 440	channel0_0	writeback_dir...	name=0:15, ino=8143007, index=771
16:18:31.412 266 000	channel0_0	mm_page_all...	page=0xffff7c01cebbac40, order=0, gfp_flags=16908506, migratetype=2
16:18:31.412 268 600	channel0_0	kmem_cache...	call_site=0xffff8000002fd250, ptr=0xffff80832fe6e000, bytes_req=128, bytes_alloc=128, gfp_flags=3...
16:18:31.412 269 200	channel0_0	kmem_kfree	call_site=0xffff8000002f4b0c, ptr=0x0
16:18:31.412 270 120	channel0_0	writeback_dir...	name=0:15, ino=8143007, index=772
16:18:31.412 271 600	channel0_0	mm_page_all...	page=0xffff7c01cec0ee00, order=0, gfp_flags=16908506, migratetype=2
16:18:31.412 274 440	channel0_0	kmem_cache...	call_site=0xffff8000002fd250, ptr=0xffff80832fe6ea80, bytes_req=128, bytes_alloc=128, gfp_flags=3...
16:18:31.412 275 040	channel0_0	kmem kfree	call site=0xffff8000002f4b0c, ptr=0x0

Trace Compass views

Resources



Statistics

The Statistics view shows a table of event types for the session 'Global - ftfSession-20150529-131519/kernel'. The table lists event types, their percentage of total events, the total number of events, and the number of events currently in selection.

Level	Event Types		Events total	Events in selection
Event Types	rpc_task_run_action	14.2 %	19,514	0
	kmem_cache_alloc	6.3 %	10,105	0
	kmem_cache_free	6 %	9,636	0
	kmem_kfree	5.9 %	9,411	0
	rcu_utilization	4.9 %	7,884	0
	kmem_kmalloc	4.4 %	7,151	0
	irq_handler_exit	3.5 %	5,685	0
	irq_handler_entry	3.5 %	5,685	0
	workqueue_queue_work	3.2 %	5,122	0
	workqueue_execute_start	3.2 %	5,122	0
	workqueue_activate_work	3.2 %	5,122	0
	workqueue_execute_end	3.2 %	5,122	0
	skb_consume	2.8 %	4,466	0

Project Explorer – Trace Compass session

- Tracing
 - Experiments [1]
 - FTF [2]
 - ftfSession-20150529-131519/kernel
 - mySession-20150529-074501/kernel
 - Traces [2]
 - ftfSession-20150529-131519 [1]
 - kernel
 - CPU usage
 - LTTng Kernel Analysis
 - Tmf Statistics Analysis
 - mySession-20150529-074501 [1]
 - kernel

Linux Application Debug



CodeWarrior– Debugging ARM Target

Debug - simple_linux_app/src/main.c - CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access C/C++ Debug

Debug

- simple_linux_app [C/C++ Remote Application]
 - simple_linux_app.elf [1781] [cores: 6]
 - Thread #1 1781 [core: 6] (Suspended : Breakpoint)
 - main() at main.c:29 0x400558

Remote Shell
/home/b32721/Freescale/CW4NET_v2015.05_b150430/C

OS Resources

Pid	User	Processes
1	root	Processes
2	root	Process groups
3	root	Threads
4	root	File descriptors
5	root	Sockets
6	root	Shared-memory regions
7	root	Semaphores
8	root	Message queues
9	root	Kernel modules
10	root	0 [rcu_bh]
11	root	0 [migration/0]
12	root	1 [watchdog/0]
13	root	1 [watchdog/1]
14	root	1 [migration/1]
16	root	1 [ksoftirqd/1]
17	root	1 [kworker/1:0H]
18	root	2 [watchdog/2]
19	root	2 [migration/2]
20	root	2 [ksoftirqd/2]
21	root	2 [kworker/2:0]
22	root	2 [kworker/2:0H]

Disassembly

Enter location here

Address	Instruction
000000000400558:	adrp x0, 0x400000
00000000040055c:	add x0, x0, #0x600
000000000400560:	bl 0x4003e0 <puts@plt>

Console

```
simple_linux_app [C/C++ Remote Application] Remote Shell
root@ls2085aqds:~# chmod +x /home/root/simple_linux_app
Process /home/root/simple_linux_app.elf created; pid = 
Listening on port 1234
Remote debugging from host 192.168.1.1
```

Remote Systems

- Local
- ScpConnection
 - Scp Files
 - Ssh Shells
 - Ssh Terminals



Two ways to run GDB

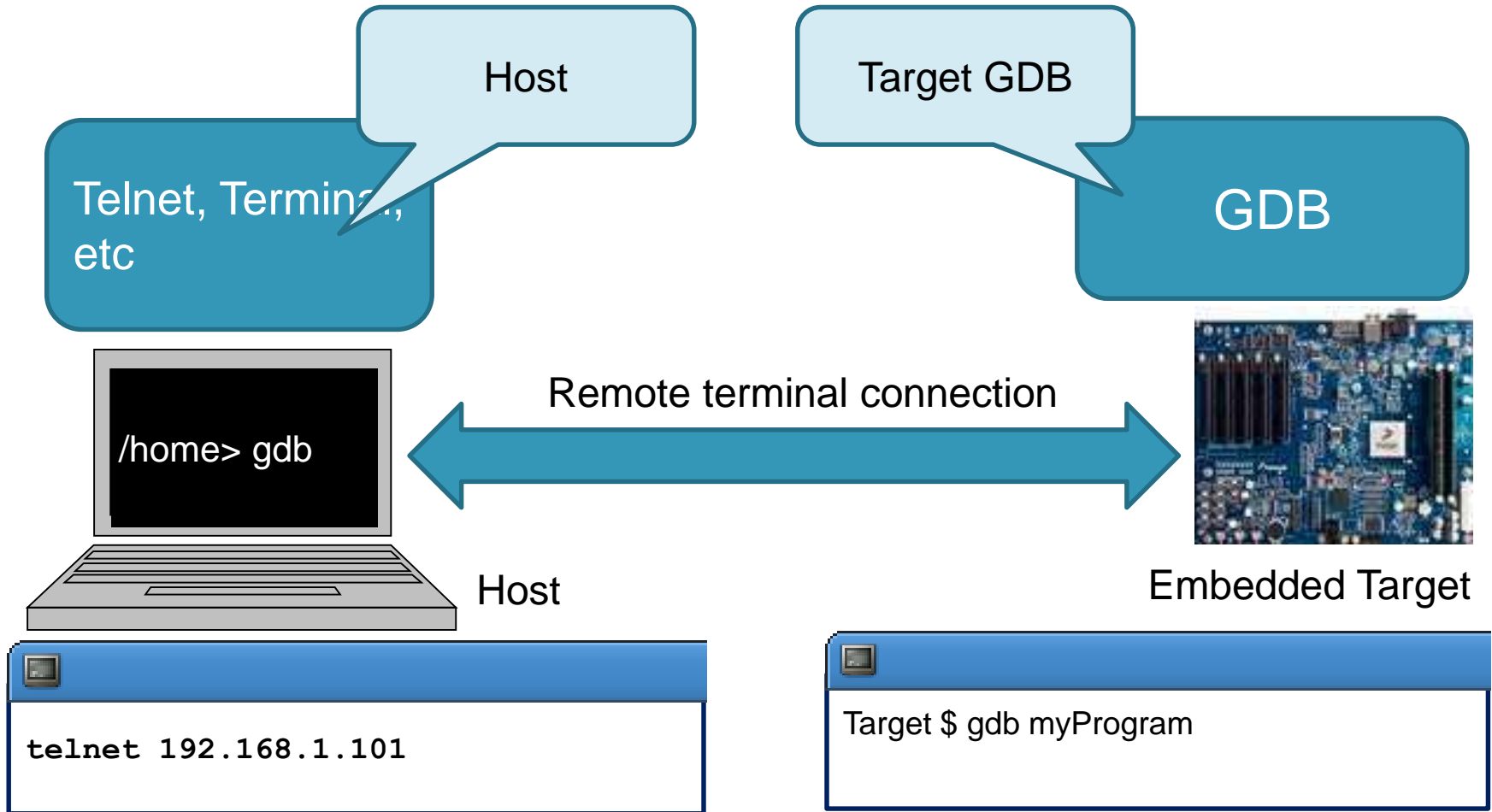
Target (self-hosted)

- GDB runs on the target (DUT)
 - E.g. Target OS: Linux
- Debugs an application running on the same system
- Interface with the target system using other applications
 - telnet into the target system to run GDB from the Linux command prompt

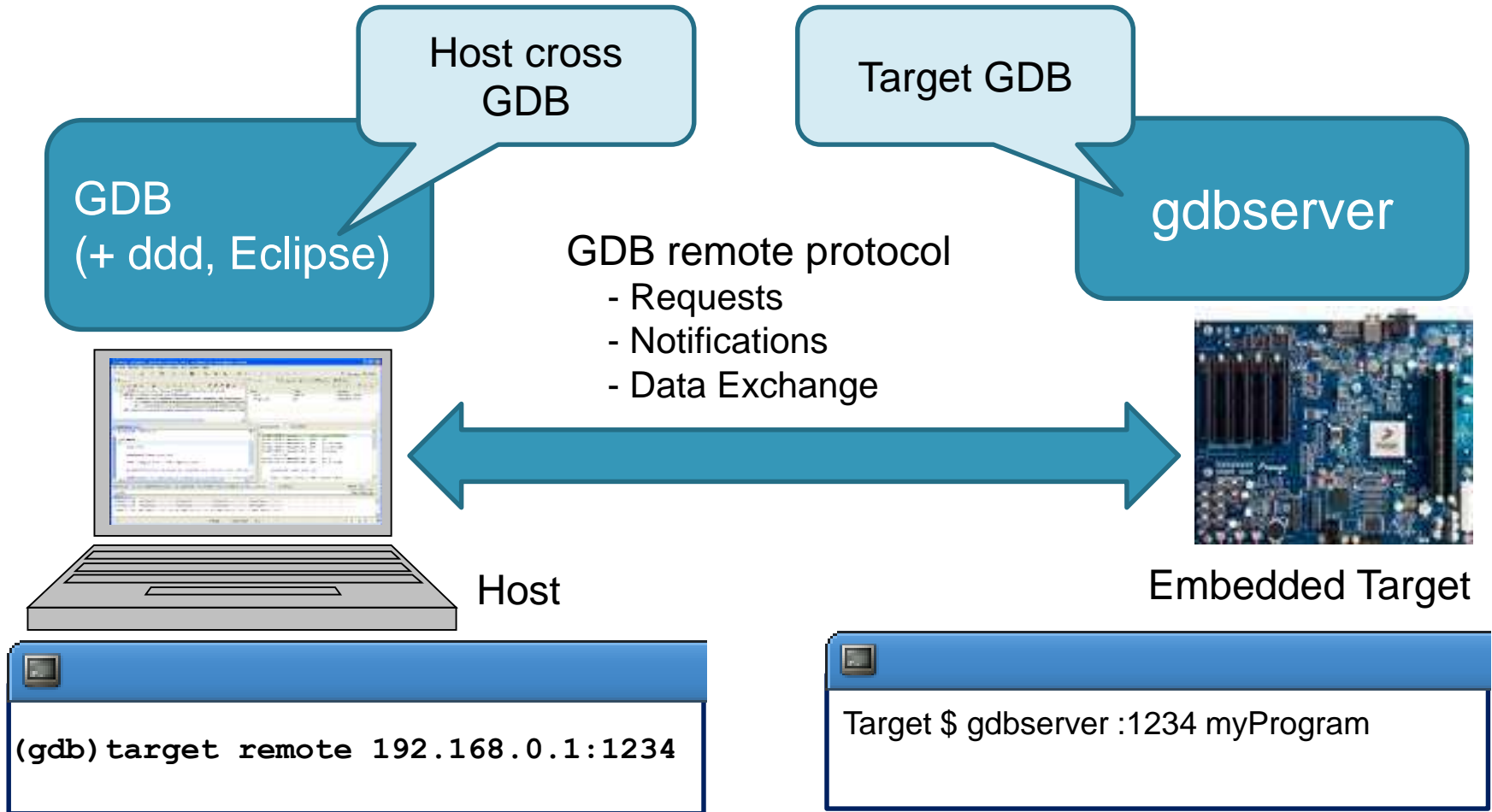
Native (Host)

- GDB runs on the development host
 - Host OS and Target OS are not necessarily the same
- Remotely debugs an application running on the target
 - Socket connection or UART connection over the OS's drivers and interface carries GDB commands and responses
 - Host GDB communicates with target GDB server

GDB Self-Hosted Target Debugging ARM Target



GDB Host Remote Debugging ARM Target



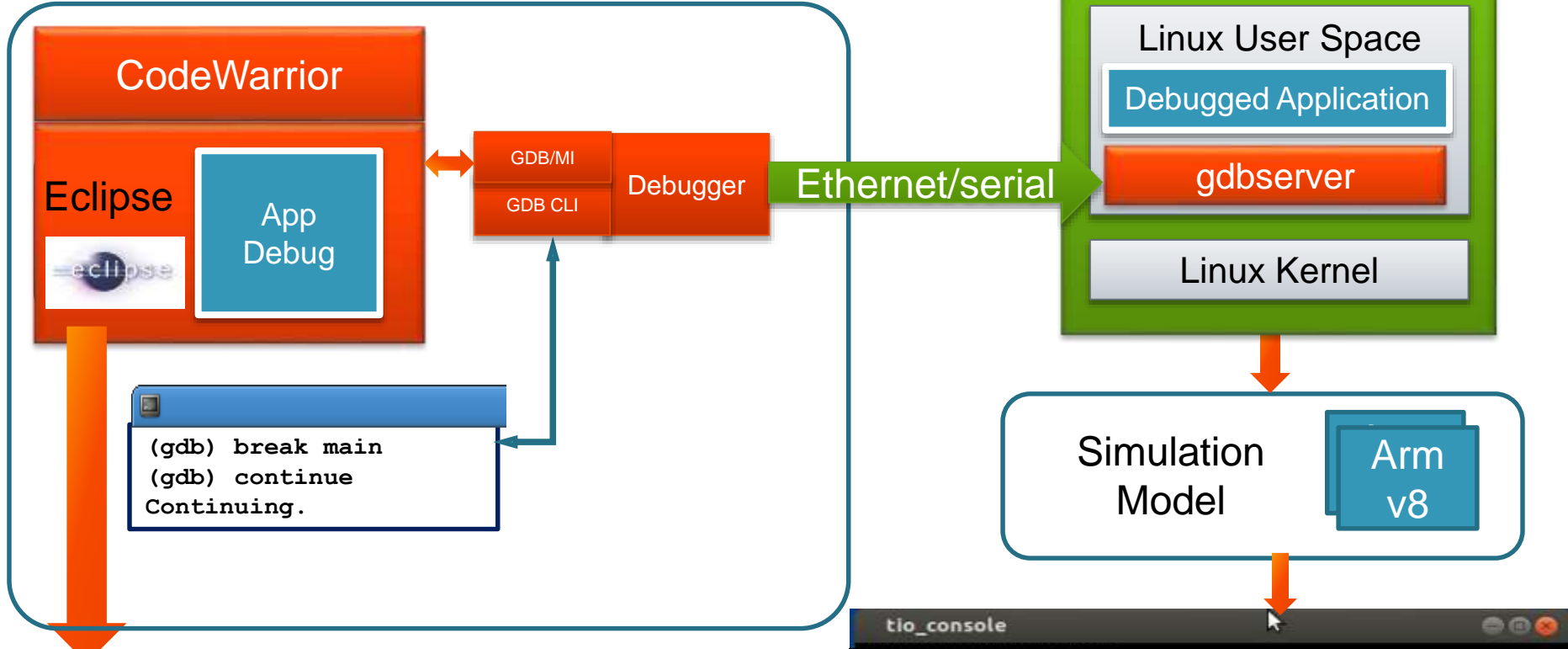
Linux application debug – Capabilities

- **gdbserver** Debug agent
 - User-space application
 - Uses **ptrace**
- Debug **scenarios** supported
 - **Download**, start & debug application from main
 - **Attach** to a running process
- Features
 - Read/write memory, registers, variables
 - **Threads creation/death detection**
 - Shared libraries awareness
 - Configurable signal policies
 - I/O redirection
- **OS Resources**
- CodeWarrior – GDB server interaction
 - Ethernet connection
 - Serial connection

Linux application debug – Prerequisites (2)

- **LS2085A RDB** board
- **Linux** running on the target
- **Network connectivity** inside Linux
- **GDB server** debug agent on the target
- Ways of putting GDB server on the target
 - GDB server is included by default in the SDK image – no change required
 - Compile GDB Agent separately
 - **bitbake –c cleansstate gdb**
 - **bitbake gdb**
 - Use **SCP** to put GDBAgent on the target (we'll find the **ELF** in <YoctoInstallationPath>/fsl-qoriq-sdk/build_ls2085ardb_release/tmp/work/aarch64-fsl-linux/gdb/7.7.1+fsl-r0/build/gdb/gdbserver/gdbserver)

Linux Application Debug



CodeWarrior

Eclipse

App
Debug

GDB/MI

GDB CLI

Debugger

Ethernet/serial

Remote Linux System

Linux User Space

Debugged Application

gdbserver

Linux Kernel

Simulation
Model

Arm
v8

- CW project: project wizard
- Eclipse DSF – automatic download and launch application
- Run Control: run/suspend/step
- Breakpoints
- Registers View: GPR registers

tio_console

```
root@freescale $ uname -a
Linux freescale 3.12.0+ #1 Wed Feb 26
09:45:41 IST 2014 aarch64 GNU/Linux
root@freescale $
root@freescale $
root@freescale $ ./myLinuxApplication
running Linux Application
```

Standalone (CLI) Linux Application debug

- Compiling the application

```
$ aarch64-linux-gnu-gcc hello.c -o hello -g
```

- Debugging the application

```
Host$ gdb ./hello
GNU gdb (GDB) 7.6
...
This GDB was configured as
"--host=x86_64-linux-gnu
  --target=aaarch64-linux
(gdb)target remote 192.168.0.1:1234
(gdb)break main
(gdb)continue
```

```
Target$ gdbserver :1234 ./hello
Process ./hello created;
Listening on port 1234

Remote debugging from host
192.168.0.2
```

- GDB commands

- run/Ctrl-C(stop), next, step, nexti, stepi, breakpoint
- backtrace, frame, print, x, dump
- set, call, jump, return

Activity

Linux Application Debug – Simple Example



Linux application debug – Debugging a simple Linux Application debug project – Activity

- Select File > New > ARMv8 Stationary
- Set the project name, select Linux Application and press Finish button

The image shows two screenshots from the CodeWarrior IDE. The left screenshot shows the 'New' menu with 'ARMv8 Stationary' selected. An orange arrow points from this menu item to the right screenshot. The right screenshot shows the 'ARMv8 Project' dialog box. The 'Project name' field is set to 'simple_linux_app'. The 'Linux Application Debug' category is selected in the 'Available stationaries' list. The 'Finish' button is highlighted with an orange callout.

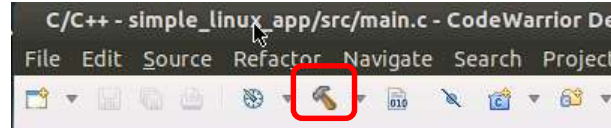
Set the project name

Select Linux Application type

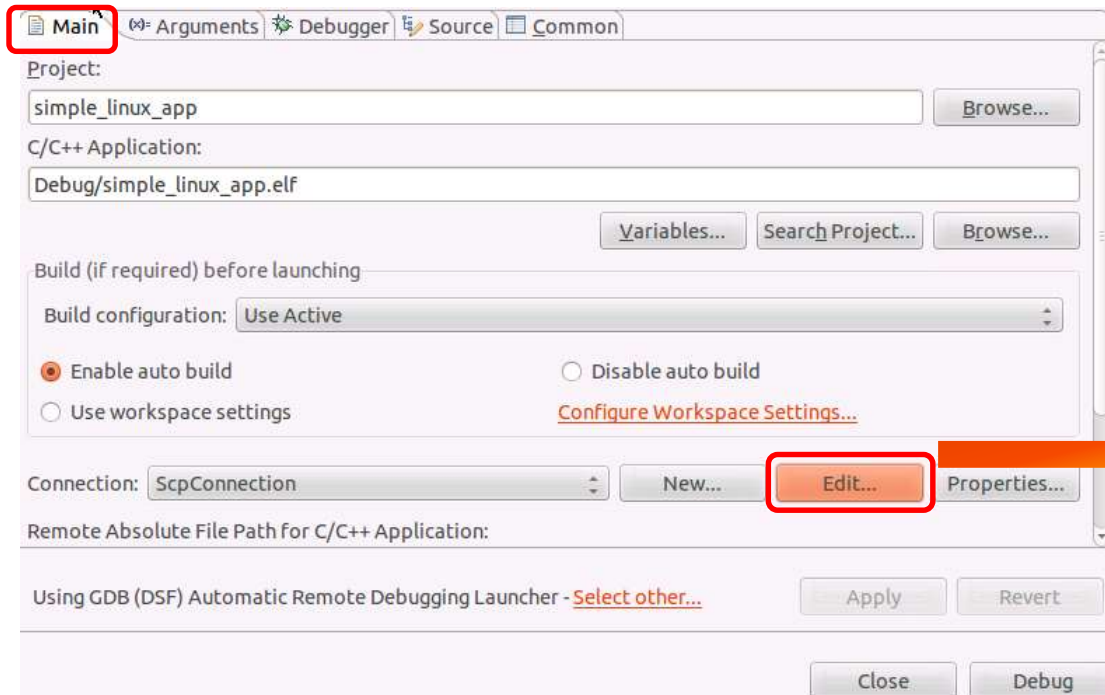
Press Finish

Linux application debug – Debugging a simple Linux Application debug project – Activity

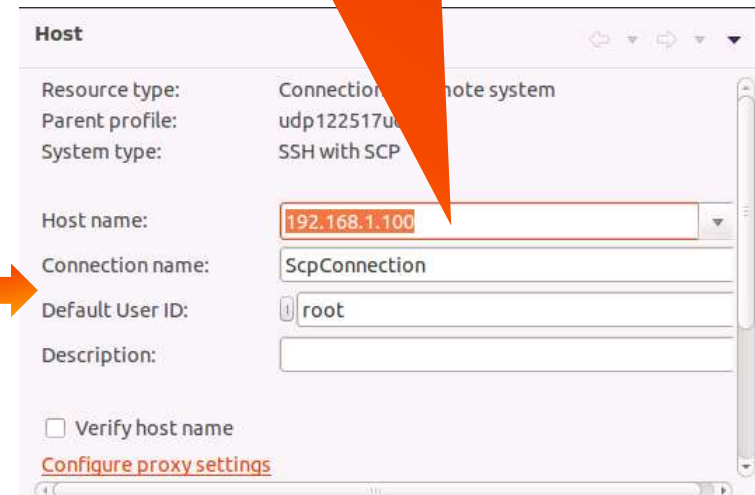
- Build Project



- Configure project settings: Run -> Debug Configurations and select the project from C/C++ Remote Application



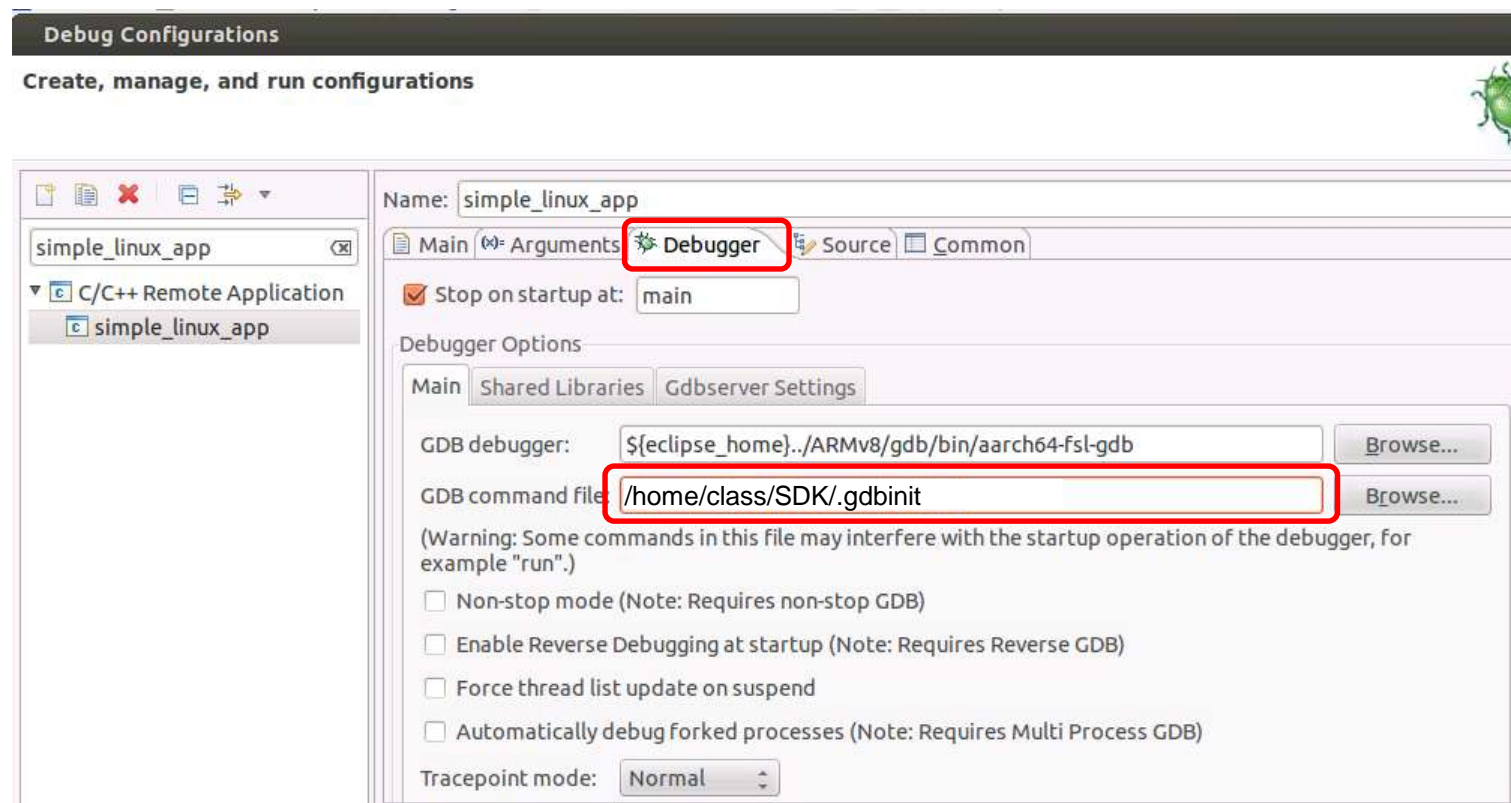
Set the IP address of the remote Linux target



Linux application debug – Debugging a simple Linux Application debug project – Activity

- Set the gdb initialization file where the sysroot is set. .gdbinit file contains:

```
set sysroot <YoctoInstallationPath>/fsl-qorik-sdk/build_ls2085ardb_release/tmp/sysroots/ls2085ardb
```



Linux application debug – Debugging a simple Linux Application debug project – Activity

- Press Debug button and perform usual debugging

Debug - simple_linux_app/src/main.c - CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access C/C++ Debug

Debug

- simple_linux_app [C/C++ Remote Application]
- simple_linux_app.elf [1781] [cores: 6]
- Thread #1 1781 [core: 6] (Suspended : Breakpoint)
- main() at main.c:29 0x400558
- Remote Shell
- /home/b32721/Freescale/CW4NET_v2015.05_b150430/c

OS Resources

Pid	User	Processes
1	root	Processes
2	root	Process groups
3	root	Threads
4	root	File descriptors
5	root	Sockets
6	root	Shared-memory regions [1]
7	root	Semaphores [0]
8	root	Message queues [1]
9	root	Kernel modules
10	root	[rcu_bh]
11	root	[migration/0]
12	root	[watchdog/0]
13	root	[watchdog/1]
14	root	[migration/1]
15	root	[ksoftirqd/1]
16	root	[kworker/1:0H]
17	root	[watchdog/2]
18	root	[migration/2]
19	root	[ksoftirqd/2]
20	root	[kworker/2:0]
21	root	[kworker/2:0H]
22	root	[watchdog/3]

Disassembly

Enter location here

Address	Disassembly
000000000400558:	adrp x0, 0x400000
00000000040055c:	add x0, x0, #0x600
000000000400560:	bl 0x4003e0 <puts@plt>

main.c

```
26 int
27 main(void)
28 {
29     printf("Hello ARM World!" "\n");
30     return 0;
31 }
```

Console

```
simple_linux_app [C/C++ Remote Application] Remote Shell
root@ls2085aqds:~# chmod +x /home/root/simple_linux_app
Process /home/root/simple_linux_app.elf created; pid =
Listening on port 1234
Remote debugging from host 192.168.1.1
```

Remote Systems

- Local
- ScpConnection
- Scp Files
- Ssh Shells
- Ssh Terminals

Linux application debug – Debugging a simple Linux Application debug project – Activity

- Notes:
 - Code Warrior automatically connect to the remote target (over ssh) start the gdbserver on the configured port, debugging the current application
 - No need for the user to connect to target and configure or run programs
 - OS Resources Window provides system information: processes, threads, sockets, shared memory...

Activity

Attach to an existing Linux process



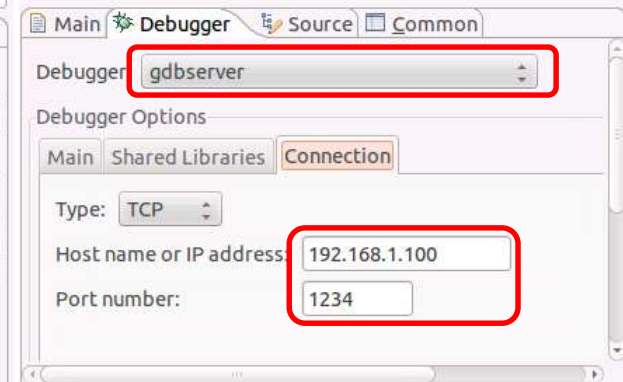
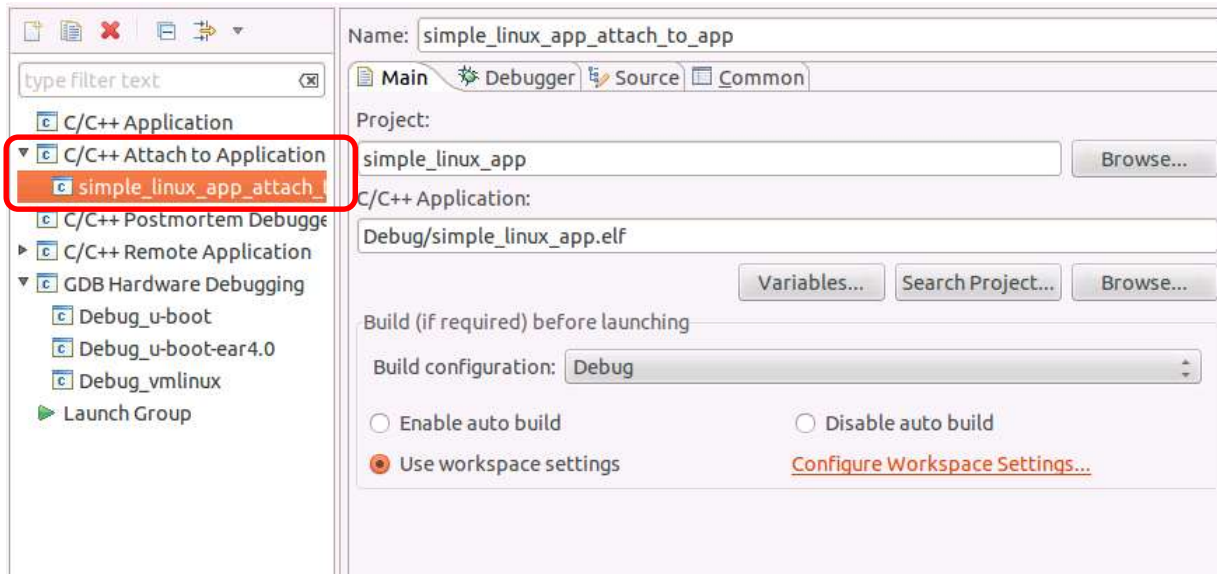
Linux application debug – Attach to a running Linux Application example – Activity (prerequisites)

- The application **simple_linux_app** should be copied on target (using a CodeWarrior download session or manually using **scp**)
- Assume a **ssh/telnet console** is active on the target board
- Run the application on target
 - root@ls2085ardb:~# ./simple_linux_app.elf
- This will make sure that the test application will be running on target.

- Manually start the gdbserver to allow attaching to any linux application:
 - root@ls2085ardb:~# gdbserver --multi :1234

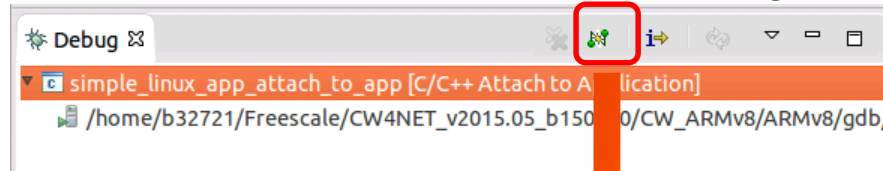
Linux application debug – Attach to a running Linux Application example – Activity

- Open **Debug Configurations: Run → Debug Configurations**
- For C/C++ Attach to Application: create a new launch
- The Main tab will automatically be completed
- In the Debugger tab:
 - Main sub-tab: add gdbinit file
 - Connection sub-tab: set the target parameters: IP address and port
- Press Debug button to start debugging

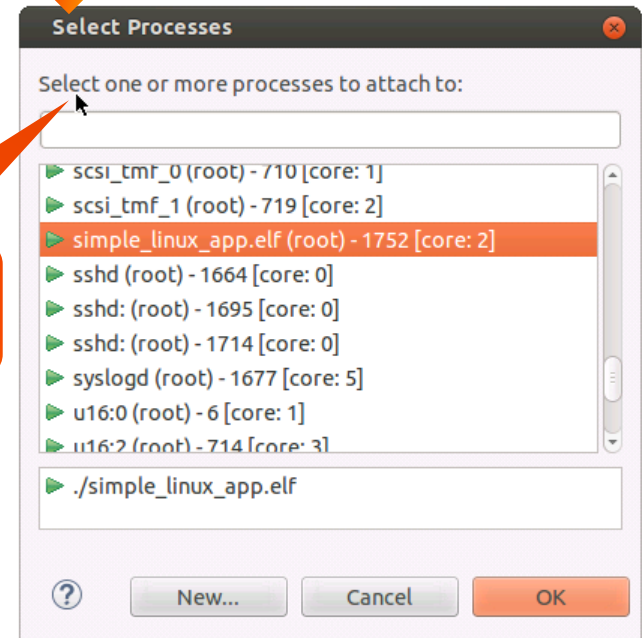


Linux application debug – Attach to a running Linux Application example – Activity

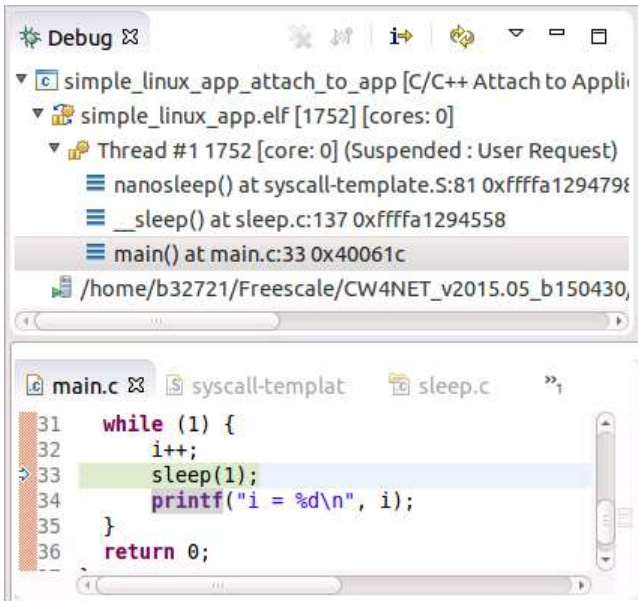
- Hit “Connect to a process” icon to open the pop-up dialog for selecting the application



- From the pop-up dialog select the relevant running application
- The debugger will attach and user will be able to **suspend** and debug the application as usual



You can filter processes list simple_*



NADK reflector - Debug Print

Demonstrate the CodeWarrior Debug Print capabilities

- Debug Print - Fundamentals
- General Debug Print controls
- HW & SW configurations
- Results and interpretations



Debug Print – Fundamentals

Debug Print consists in:

- **Server** side: running on target Linux OS for collecting Kernel Ring Buffer logs and application messages to standard output;
- **Client** side: running under CW for getting data out of the server, display and various configurations

Linux OS running on
LS 2085A-RDB

Server side

ls.target.server – reads logs
and sends log data over TCP/IP

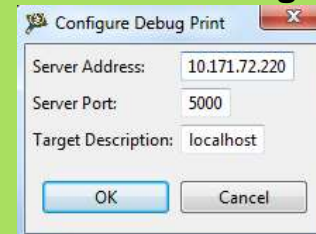
libls.linux.debugprint.lib.so –
redirect user space application
messages to **ls.target.server**

TCP/IP over Eth

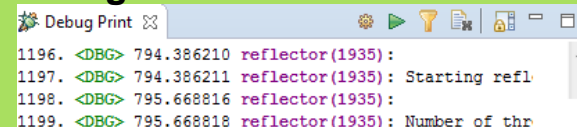
No Debug Probe

CW ARMv8 running
on Host PC

Client for ls.target.server



Debug Print Viewer



NADK reflector – Hardware setup

LS 2085A-RDB



Loop back



TCP/IP over Eth link

No Debug Probe

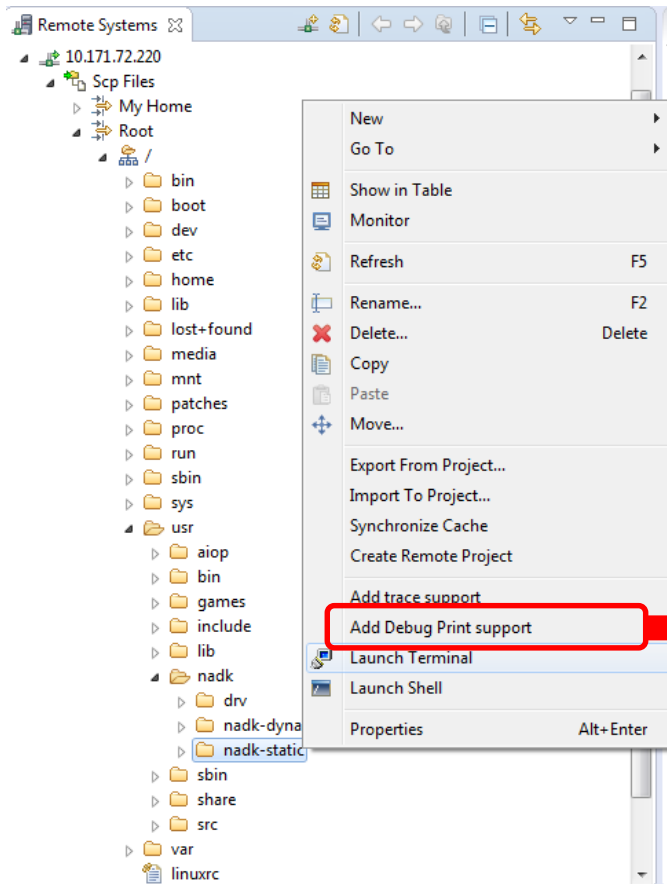
Host PC running
CW ARMv8



- NADK reflector is a user space application used to demonstrate basic network functionalities (*check AN4940*)
- Debug Print provides an easy method for checking Kernel & Application activities

Server Side: setup

Using RSE all Debug Print Server utilities can be copied directly into the target Linux OS via “Add Debug Print Support”



linux.armv8.debugprint folder is created with all the prerequisites

Server Side: setup (cont'd)

Starting the server: `ls.target.server [PORT] [-k]`

PORT : default 5000

-k : it does not clear the kernel buffer, but uses an internal server logic for determining which are the newer messages

E.g.: starting the Debug Print server with default settings

```
root@ls2085ardb:/usr/nadk/nadk-static#./Linux.armv8.debugprint/bin/Ls.target.server
Using port 5000
Initializing
```

Client Side: setup

Open the Debug Print Viewer and connect the Client with the Server using TCP/IP and Port

The image illustrates the steps to set up the Debug Print Viewer on the client side:

- 1** Open the **Show View** dialog and select **Debug Print** from the list of views.
- 2** The **Debug Print** window opens, displaying a **Configure Debug Print** dialog. The configuration includes:
 - Server Address: 10.171.72.220
 - Server Port: 5000
 - Target Description: localhost
- 3** Click **OK** in the **Configure Debug Print** dialog to apply the settings.

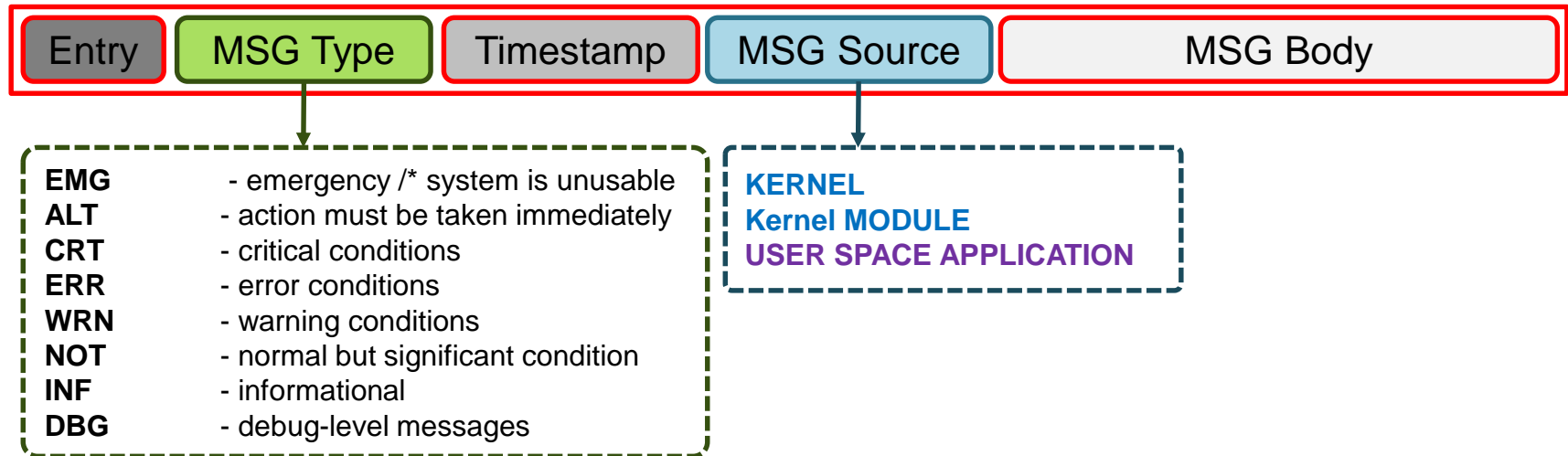
The **Debug Print** window features several control buttons:

- Start/Stop Client**: Represented by a play/pause button.
- Clear console**: Represented by a document with an 'X' icon.
- Custom filter**: Represented by a gear icon.
- Scroll lock/unlock**: Represented by a lock icon.

Client Side: Debug Print messages format

```
Info: Connected to target localhost (10.171.72.220:5000)
1. <INF> 0.000000 (kernel): Booting Linux on physical CPU 0x0
2. <INF> 0.000000 (kernel): Initializing cgroup subsys cpu
3. <NOT> 0.000000 (kernel): Linux version 3.19.3-Layerscape2-SDK+g9f41bb6 (b11883@fsr-ub1264-120) (gcc
version 4.8.3 20140401 (prerelease) (Linaro GCC 4.8-2014.04) ) #1 SMP PREEMPT Fri May 15 09:07:30 EEST 2015
4. <WRN> 0.000000 CPU: AArch64 Processor [411fd071] revision 1
```

Entry format:

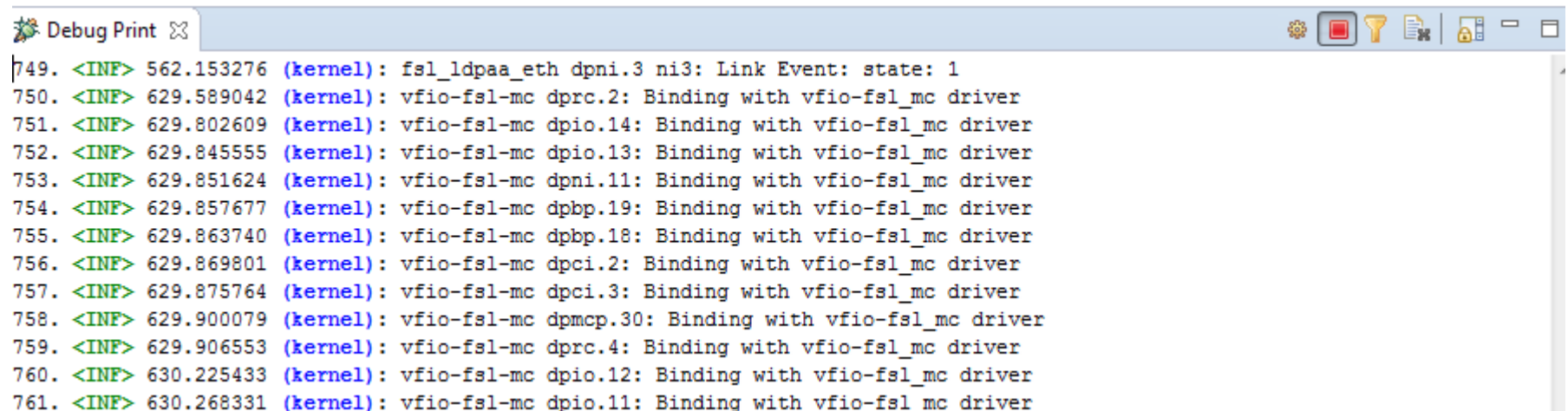


NADK reflector: startup

Step 1: setup the environment according with nadk reflector requirements:

```
root@ls2085ardb:/usr/nadk/nadk-static/bin# ./bind_dprc.sh
#1)    Allow unsafe interrupts
#1.1)  dprc.2 container driver override
#1.2)  Binding dprc.2 to VFIO driver
#1.1)  dprc container driver override
#1.2)  Bind dprc.4 to VFIO driver2 4  vfio
```

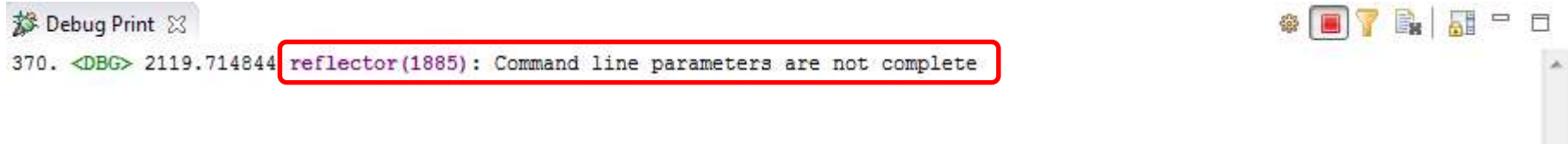
Step 2: Debug Print client will catch all logs during setup:



```
749. <INF> 562.153276 (kernel): fsl_ldpaa_eth dpni.3 ni3: Link Event: state: 1
750. <INF> 629.589042 (kernel): vfio-fsl-mc dprc.2: Binding with vfio-fsl_mc driver
751. <INF> 629.802609 (kernel): vfio-fsl-mc dpio.14: Binding with vfio-fsl_mc driver
752. <INF> 629.845555 (kernel): vfio-fsl-mc dpio.13: Binding with vfio-fsl_mc driver
753. <INF> 629.851624 (kernel): vfio-fsl-mc dpni.11: Binding with vfio-fsl_mc driver
754. <INF> 629.857677 (kernel): vfio-fsl-mc dpbp.19: Binding with vfio-fsl_mc driver
755. <INF> 629.863740 (kernel): vfio-fsl-mc dpbp.18: Binding with vfio-fsl_mc driver
756. <INF> 629.869801 (kernel): vfio-fsl-mc dpci.2: Binding with vfio-fsl_mc driver
757. <INF> 629.875764 (kernel): vfio-fsl-mc dpci.3: Binding with vfio-fsl_mc driver
758. <INF> 629.900079 (kernel): vfio-fsl-mc dpmcp.30: Binding with vfio-fsl_mc driver
759. <INF> 629.906553 (kernel): vfio-fsl-mc dprc.4: Binding with vfio-fsl_mc driver
760. <INF> 630.225433 (kernel): vfio-fsl-mc dpio.12: Binding with vfio-fsl_mc driver
761. <INF> 630.268331 (kernel): vfio-fsl-mc dpio.11: Binding with vfio-fsl_mc driver
```

NADK reflector: startup (cont'd)

If nadk-reflector is started incorrectly the Client will catch that:

A screenshot of a terminal window titled "Debug Print". The terminal shows a command prompt "370. <DBG> 2119.714844" followed by the output "reflector(1885): Command line parameters are not complete". The output text is highlighted with a red rectangular box. The terminal window has standard Linux window controls (minimize, maximize, close) and a taskbar with various icons at the top right.

```
370. <DBG> 2119.714844 reflector(1885): Command line parameters are not complete
```

STEP3: redirect nadk-reflector standard output to Server by loading the appropriate library and start the application:

```
root@ls2085ardb:/usr/nadk/nadk-static/bin# LD_PRELOAD=/usr/nadk/nadk-  
static/bin/linux.armv8.debugprint/lib/Libls.Linux.debugprint.Lib.so ./reflector -g  
dprp.4 -d 10
```

At this point the Linux console should look like:

```
main 863-NOTICE-port => 0 - dpni-9 being created  
main 863-NOTICE-port => 1 - dpni-8 being created  
main 868-NOTICE-port => 0 - dpconc-1 being created  
main 908-NOTICE-Setting number of threads equalsto online CPUs  
  
main 928-NOTICE-Number of threads = 8  
nadk_reflector_configure_all_devices 374-NOTICE-port => 1/2 - dpni-9 being created  
nadk_reflector_configure_all_devices 472-NOTICE-Port Id = dpni-9
```


NADK reflector: Debug Print results

Reflector application messages

```
763. <INF> 630.333594 (kernel): vfio-fsl-mc dpio.9: Binding with vfio-fsl_mc driver
764. <INF> 630.396739 (kernel): vfio-fsl-mc dpio.8: Binding with vfio-fsl_mc driver
765. <INF> 630.439465 (kernel): vfio-fsl-mc dpio.7: Binding with vfio-fsl_mc driver
766. <INF> 630.482192 (kernel): vfio-fsl-mc dpio.6: Binding with vfio-fsl_mc driver
767. <INF> 630.524901 (kernel): vfio-fsl-mc dpio.5: Binding with vfio-fsl_mc driver
768. <INF> 630.530873 (kernel): vfio-fsl-mc dpni.9: Binding with vfio-fsl_mc driver
769. <INF> 630.536838 (kernel): vfio-fsl-mc dpni.8: Binding with vfio-fsl_mc driver
770. <INF> 630.542936 (kernel): vfio-fsl-mc dpbp.6: Binding with vfio-fsl_mc driver
771. <INF> 630.548907 (kernel): vfio-fsl-mc dpseci.1: Binding with vfio-fsl_mc driver
772. <INF> 630.555046 (kernel): vfio-fsl-mc dpcon.1: Binding with vfio-fsl_mc driver
773. <INF> 630.579580 (kernel): vfio-fsl-mc dpmcp.31: Binding with vfio-fsl mc driver
774. <DBG> 872.572601 reflector(1850):
775. <DBG> 872.572604 reflector(1850): Initialization starts with DPRC = dprp.4
776. <DBG> 872.572608 reflector(1850):
777. <DBG> 872.572609 reflector(1850): Starting reflector, data_mem_size: 0x400000
778. <WRN> 872.597878 (kernel): Bits 55-60 of /proc/PID/pagemap entries are about to stop being page-shift some time soon. See
the linux/Documentation/vm/pagemap.txt for details.
```

new Kernel messages are catch

NADK reflector: Debug Print results (cont'd)

Initiate a ping in order to trigger new messages from nadk-reflector

```
root@ls2085ardb:/usr/nadk/nadk-static/bin# ping 8.8.8.10 -c 10
PING 8.8.8.10 (8.8.8.10) 56(84) bytes of data.
64 bytes from 8.8.8.10: icmp_seq=1 ttl=64 time=0.204 ms
64 bytes from 8.8.8.10: icmp_seq=2 ttl=64 time=0.113 ms
64 bytes from 8.8.8.10: icmp_seq=3 ttl=64 time=0.104 ms
64 bytes from 8.8.8.10: icmp_seq=4 ttl=64 time=0.101 ms
64 bytes from 8.8.8.10: icmp_seq=5 ttl=64 time=0.102 ms
64 bytes from 8.8.8.10: icmp_seq=6 ttl=64 time=0.100 ms
64 bytes from 8.8.8.10: icmp_seq=7 ttl=64 time=0.099 ms
64 bytes from 8.8.8.10: icmp_seq=8 ttl=64 time=0.099 ms
64 bytes from 8.8.8.10: icmp_seq=9 ttl=64 time=0.101 ms
64 bytes from 8.8.8.10: icmp_seq=10 ttl=64 time=0.098 ms

--- 8.8.8.10 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 8998ms
rtt min/avg/max/mdev = 0.098/0.112/0.204/0.031 ms
```



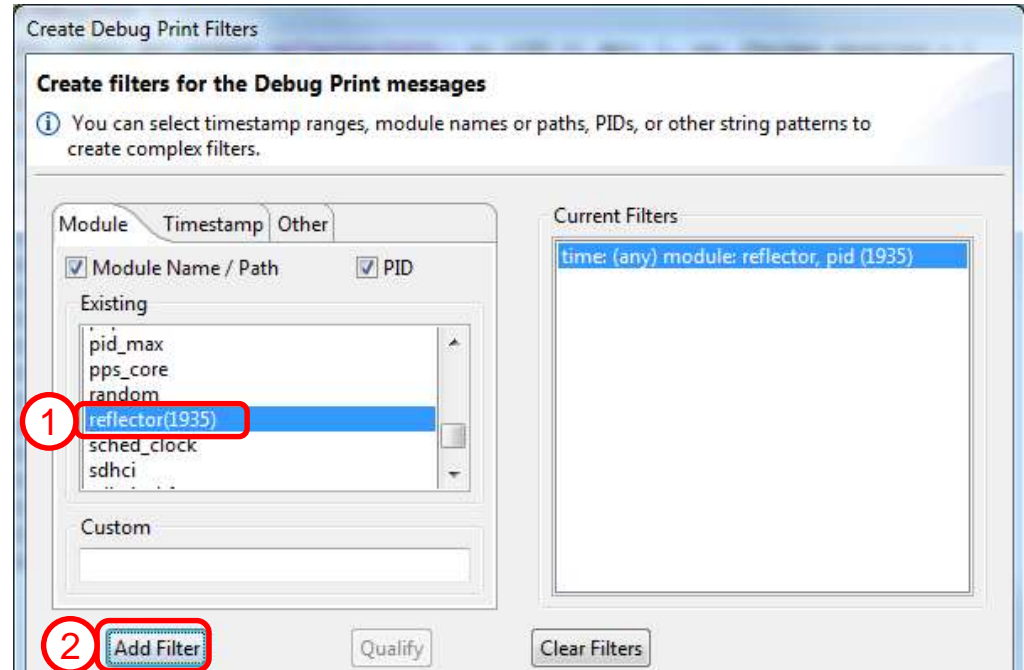
NADK reflector: Debug Print results (cont'd)

Debug Print Client console should catch all nadk-reflector messages

```
Debug Print
1196. <DBG> 794.386210 reflector(1935):
1197. <DBG> 794.386211 reflector(1935): Starting reflector, data_mem_size: 0x400000
1198. <DBG> 795.668816 reflector(1935):
1199. <DBG> 795.668818 reflector(1935): Number of threads = 8
1200. <DBG> 795.668842 reflector(1935): Hit Ctrl-C (or send SIGINT) to terminate.
1201. <DBG> 987.959648 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 1
1202. <DBG> 987.959748 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 2
1203. <DBG> 988.958604 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 3
1204. <DBG> 988.958658 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 4
1205. <DBG> 989.957894 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 5
1206. <DBG> 989.957947 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 6
1207. <DBG> 990.957893 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 7
1208. <DBG> 990.957943 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 8
1209. <DBG> 991.957892 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 9
1210. <DBG> 991.957944 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 10
1211. <DBG> 992.957891 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 11
1212. <DBG> 992.957942 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 12
1213. <DBG> 993.957891 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 13
1214. <DBG> 993.957942 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 14
1215. <DBG> 994.957893 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 15
1216. <DBG> 994.957944 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 16
1217. <DBG> 995.957892 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 17
1218. <DBG> 995.957944 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 18
1219. <DBG> 996.957891 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 19
1220. <DBG> 996.957941 reflector(1935): => [CPU 0] dev: 1, vq: 2Packet received = 20
```

NADK reflector: Debug Print results (cont'd)

Customize the Debug Print Client to display only relevant information: messages for nadk-reflector



```
Debug Print
1201. <DBG> 987.959648 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 1
1202. <DBG> 987.959748 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 2
1203. <DBG> 988.958604 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 3
1204. <DBG> 988.958658 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 4
1205. <DBG> 989.957894 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 5
1206. <DBG> 989.957947 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 6
1207. <DBG> 990.957893 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 7
1208. <DBG> 990.957943 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 8
1209. <DBG> 991.957892 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 9
1210. <DBG> 991.957944 reflector(1935) : => [CPU 0] dev: 1, vq: 2Packet received = 10
```



Debug Print Considerations

- Debug Print Client can **show up messages from Kernel, Modules and User Applications** in a **easy straightforward** fashion allowing **filtering based on source/timestamps/keywords**
- Attaching like use cases to a running application is not supported since the **Debug Print redirect library must be loaded before application is getting started**
- Debug Print Server and Client can be started at any time

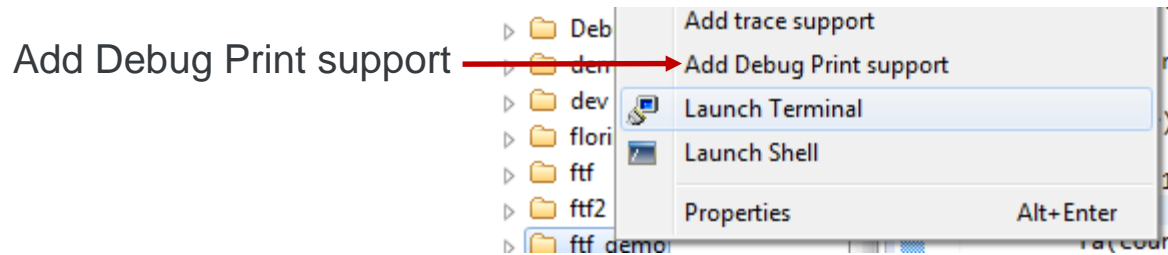
Activity

Linux Debug Print



Linux Debug Print

1. Create a new folder into your target file system.
2. Add Debug Print support (right click on your newly created folder):



3. Run `ls` in Terminal to check the files in your folder:

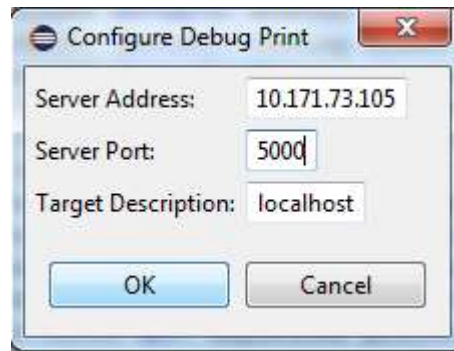
```
root@ls2085aqds:~/dbgPrint# ls
linux.armv8.debugprint
root@ls2085aqds:~/dbgPrint# █
```

Linux Debug Print

4. Launch Debug Print server (default it will start on port 5000):
./linux.armv8.debugprint/bin/ls.target.server

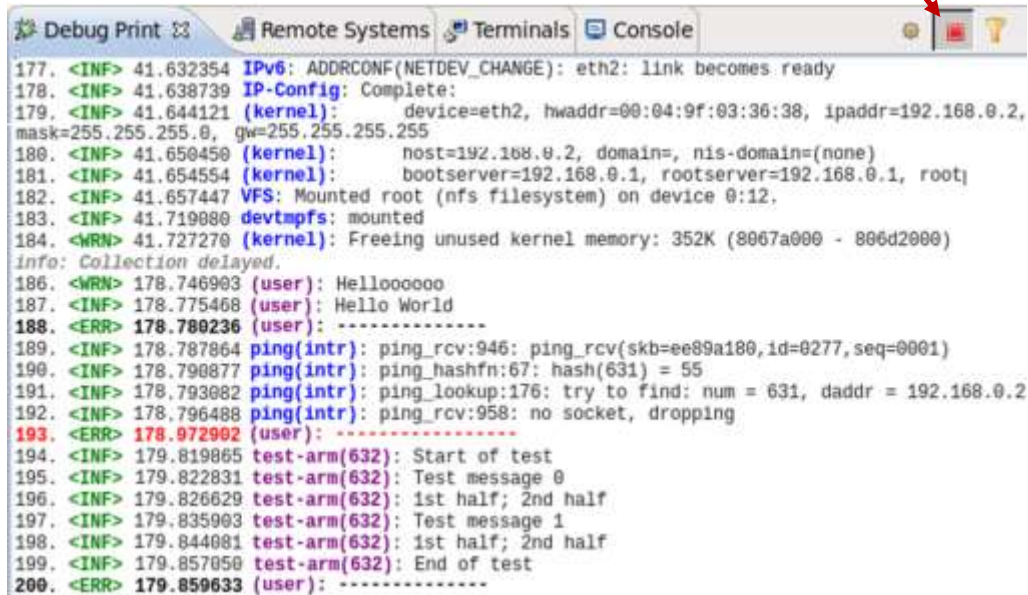
```
root@ls2085aqds:~/dbgPrint# ./linux.armv8.debugprint/bin/ls.target.server
Using port 5000
Initializing
█
```

5. Open Debug Print View (Window->Show View->Other->Software Analysis->Debug Print)
6. Click on view's Configuration button and put your target settings



Linux Debug Print

7. Start Debug Print message collection from viewer:

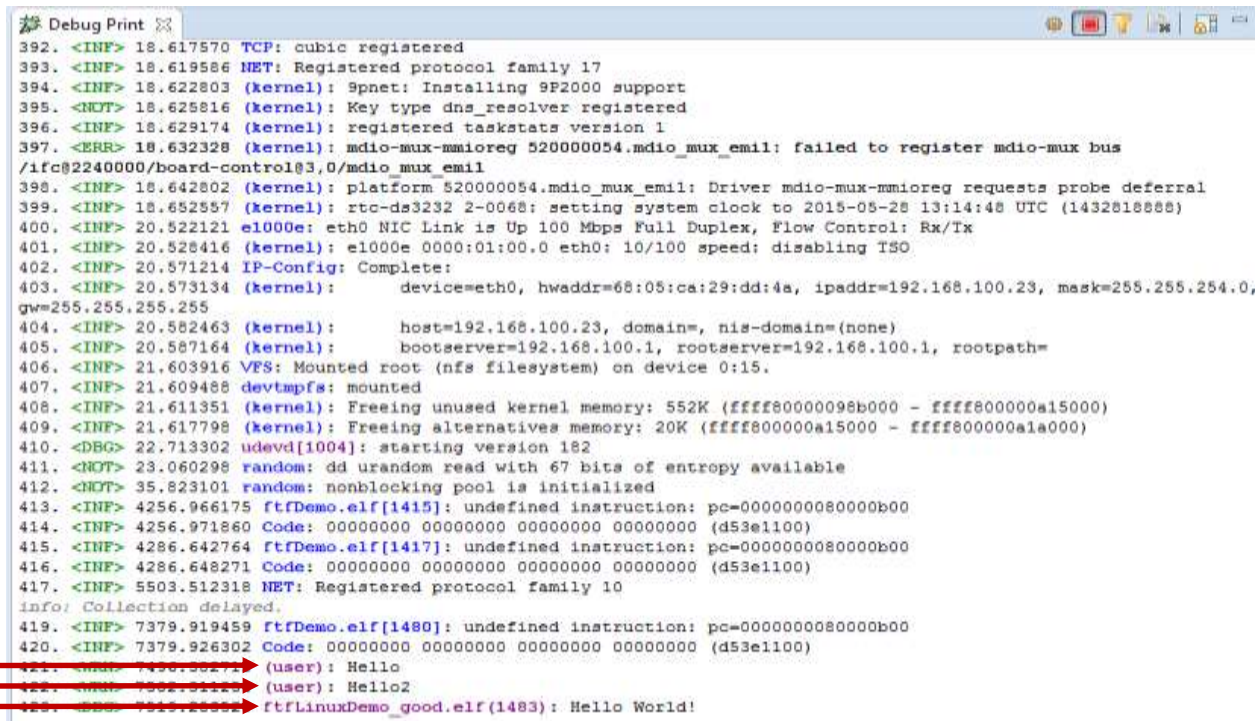


```
Debug Print Remote Systems Terminals Console
177. <INF> 41.632354 IPv6: ADDRCONF(NETDEV_CHANGE): eth2: link becomes ready
178. <INF> 41.638739 IP-Config: Complete:
179. <INF> 41.644121 (kernel): device=eth2, hwaddr=00:04:9f:03:36:38, ipaddr=192.168.0.2,
mask=255.255.255.0, gw=255.255.255.255
180. <INF> 41.650450 (kernel): host=192.168.0.2, domain=, nis-domain=(none)
181. <INF> 41.654554 (kernel): bootserver=192.168.0.1, rootserver=192.168.0.1, root|
182. <INF> 41.657447 VFS: Mounted root (nfs filesystem) on device 0:12.
183. <INF> 41.719880 devtmpfs: mounted
184. <WRN> 41.727270 (kernel): Freeing unused kernel memory: 352K (8067a000 - 806d2000)
info: Collection delayed.
186. <WRN> 178.746903 (user): Helloooooo
187. <INF> 178.775468 (user): Hello World
188. <ERR> 178.780236 (user): -----
189. <INF> 178.787864 ping(intr): ping_rcv:946: ping_rcv(skb=ee89a180,id=0277,seq=0001)
190. <INF> 178.790877 ping(intr): ping_hashfn:67: hash(631) = 55
191. <INF> 178.793082 ping(intr): ping_lookup:176: try to find: num = 631, daddr = 192.168.0.2
192. <INF> 178.796488 ping(intr): ping_rcv:958: no socket, dropping
193. <ERR> 178.972902 (user): -----
194. <INF> 179.819865 test-arm(632): Start of test
195. <INF> 179.822831 test-arm(632): Test message 0
196. <INF> 179.826629 test-arm(632): 1st half; 2nd half
197. <INF> 179.835903 test-arm(632): Test message 1
198. <INF> 179.844081 test-arm(632): 1st half; 2nd half
199. <INF> 179.857050 test-arm(632): End of test
200. <ERR> 179.859633 (user): -----
```

Linux Debug Print

8. From a separate Terminal launch some user applications that contains some *printf()* calls or *echo* some messages:

```
# echo Hello > /dev/kmsg
# echo Hello2 > /dev/kmsg
# LD_PRELOAD=/home/root/linux.armv8.debugprint/lib/libls.linux.
debugprint.lib.so ./ftfLinuxDemo.elf
```

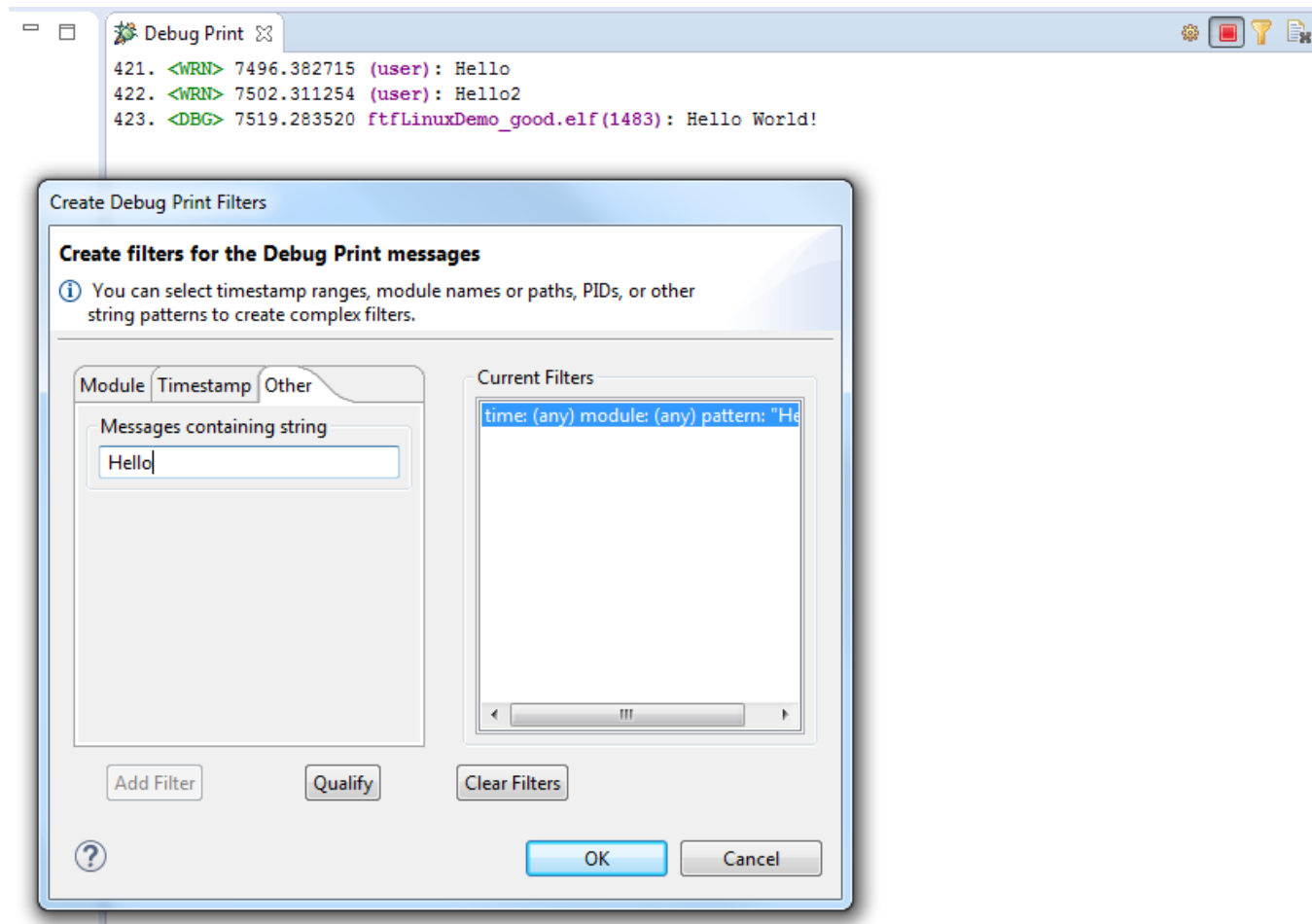


```
392. <INF> 18.617570 TCP: cubic registered
393. <INF> 18.619586 NET: Registered protocol family 17
394. <INF> 18.622803 (kernel): Spnet: Installing 9P2000 support
395. <NOT> 18.625816 (kernel): Key type dns_resolver registered
396. <INF> 18.629174 (kernel): registered taskstats version 1
397. <ERR> 18.632328 (kernel): mdio-mux-mmio-reg 520000054.mdio_mux_emil: failed to register mdio-mux bus
/1fc@2240000/board-control@3,0/mdio_mux_emil
398. <INF> 18.642802 (kernel): platform 520000054.mdio_mux_emil: Driver mdio-mux-mmio-reg requests probe deferral
399. <INF> 18.652557 (kernel): rtc-ds3232 2-0068: setting system clock to 2015-05-28 13:14:48 UTC (1432818888)
400. <INF> 20.522121 e1000e: eth0 NIC Link is Up 100 Mbps Full Duplex, Flow Control: Rx/Tx
401. <INF> 20.528416 (kernel): e1000e 0000:01:00:0 eth0: 10/100 speed; disabling TSO
402. <INF> 20.571214 IP-Config: Complete:
403. <INF> 20.573134 (kernel): device=eth0, hwaddr=68:05:ca:29:dd:4a, ipaddr=192.168.100.23, mask=255.255.254.0,
gw=255.255.255.255
404. <INF> 20.582463 (kernel): host=192.168.100.23, domain=, nis-domain=(none)
405. <INF> 20.587164 (kernel): bootserver=192.168.100.1, rootserver=192.168.100.1, rootpath=
406. <INF> 21.603916 VFS: Mounted root (nfs filesystem) on device 0:15.
407. <INF> 21.609488 devtmpfs: mounted
408. <INF> 21.611351 (kernel): Freeing unused kernel memory: 552K (ffff80000098b000 - ffff800000a15000)
409. <INF> 21.617798 (kernel): Freeing alternatives memory: 20K (ffff800000a15000 - ffff800000a1a000)
410. <DBG> 22.713302 udevd[1004]: starting version 182
411. <NOT> 23.060298 random: dd urandom read with 67 bits of entropy available
412. <NOT> 35.823101 random: nonblocking pool is initialized
413. <INF> 4256.966175 ftfDemo.elf[1415]: undefined instruction: pc=0000000080000b00
414. <INF> 4256.971860 Code: 00000000 00000000 00000000 00000000 (d53e1100)
415. <INF> 4286.642764 ftfDemo.elf[1417]: undefined instruction: pc=0000000080000b00
416. <INF> 4286.648271 Code: 00000000 00000000 00000000 00000000 (d53e1100)
417. <INF> 5503.512318 NET: Registered protocol family 10
info: Collection delayed.
419. <INF> 7379.919459 ftfDemo.elf[1480]: undefined instruction: pc=0000000080000b00
420. <INF> 7379.926302 Code: 00000000 00000000 00000000 00000000 (d53e1100)
421. <DBG> 7496.88271 (user): Hello
422. <DBG> 7502.91119 (user): Hello2
423. <DBG> 7619.88882 ftfLinuxDemo_good.elf(1483): Hello World!
```



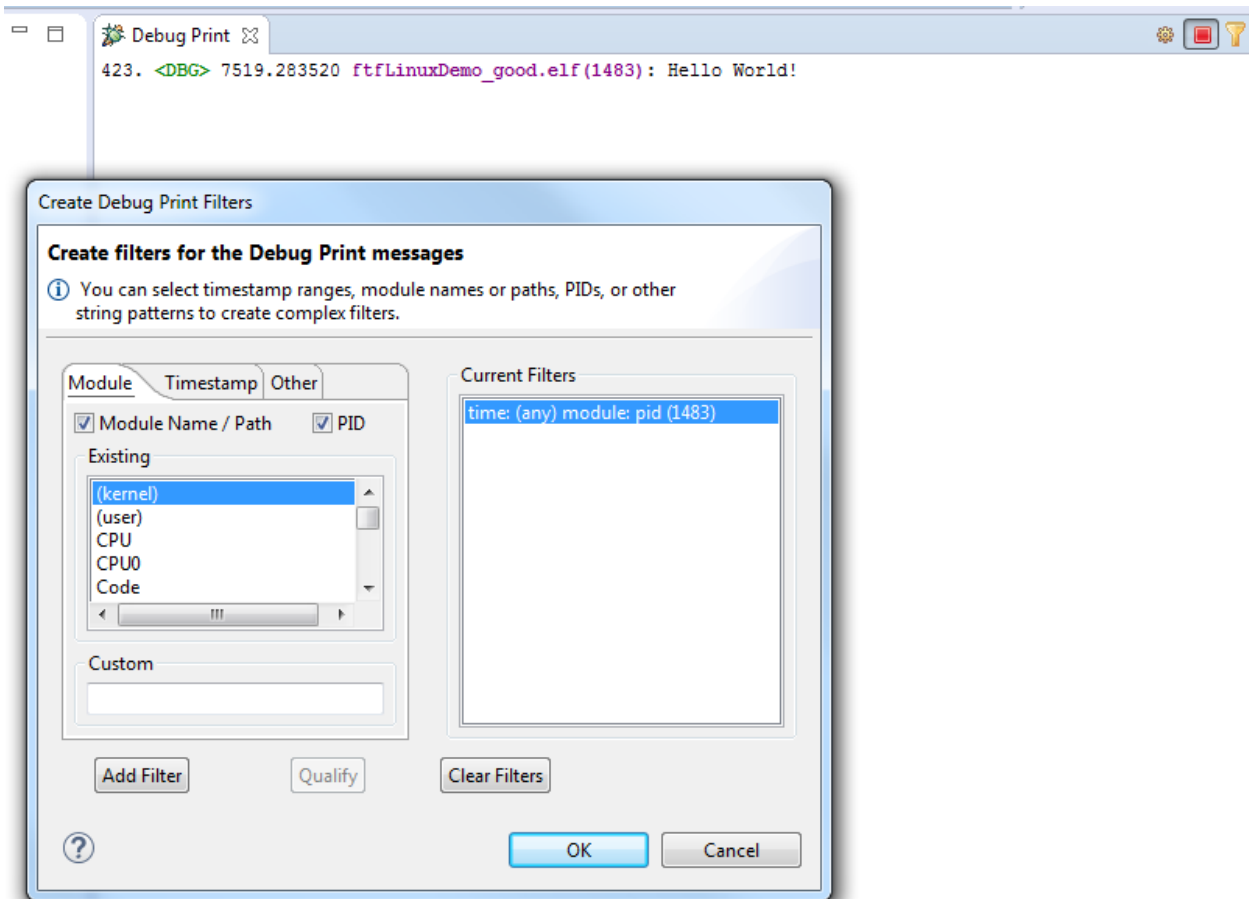
Linux Debug Print

9. You can filter the output (messages containing string “Hello”):



Linux Debug Print

10. You can filter the output (messages only from application with PID 1483):



Linux Application Trace



Linux probeless trace

- Based on a software probe
 - Linux cross-compiled application
 - CW and SDK component
- Advantages
 - Speed
 - contains only what is needed
 - Speed
 - all services are hosted on target machine
 - Nonintrusive
 - no need to instrument the target application
 - Simple API
 - can be effortlessly integrated into any testing framework
 - Data-driven
 - the configurator and probe can be easily tuned up using *xml* files

Linux probeless trace – Hardware setup

LS 2085A-QDS



Linux standalone application included in **CodeWarrior** and **Freescale SDK**

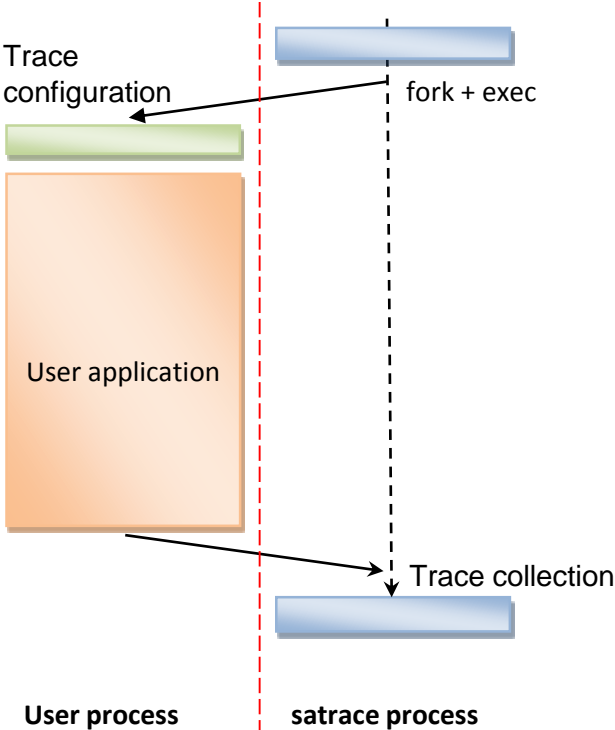
Ethernet cable + `linux.armv8.satrace`



Hardware Probe using JTAG
(E.g. CodeWarrior USB TAP)



Linux user space application trace (*command line*)




Activity

Linux application trace



Linux probeless trace (*CodeWarrior*)

1. Create and build a Linux C/C++ application
2. You may use a code like this: 
3. Build the project
4. Create or use an existing RSE connection
5. Verify that you can see the target file system

```
main.c
#include <stdio.h>

int rec(int x)
{
    if (x < 3)
        x = rec(++x);

    return x;
}

int fb(int m, int n)
{
    register int x = 3, y = 7;

    m += x*y;
    n = m+y;

    return (x*m+n*y);
}

int fa(int k)
{
    register int x = 0, y = 1;

    x = k+1;
    y -= x;

    k = fb(x,y);

    return k;
}

int main(void)
{
    int counter = 0;
    int i;

    printf("Hello World!\n");

    counter = rec(counter);

    fa(counter);

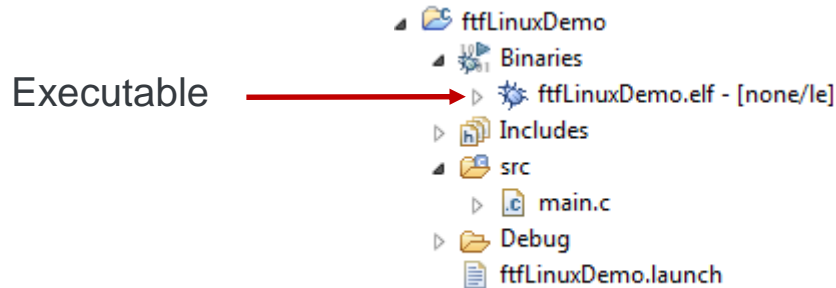
    for(i=0;i<100;i++) {
        counter++;
        fa(counter);
    }

    return 0;
}
```



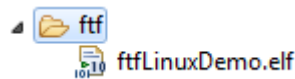
Linux probeless trace (*CodeWarrior*)

6. Go to Linux C/C++ project and copy the executable file



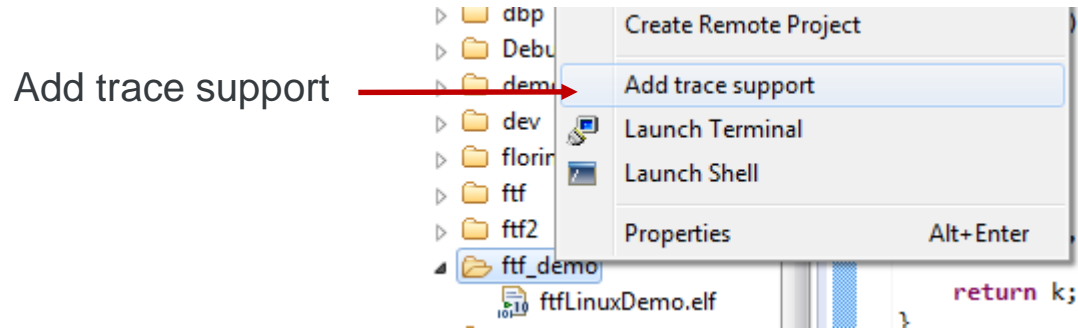
7. Create a demo folder into target file system

8. Paste the executable on target file system (into previously created folder)

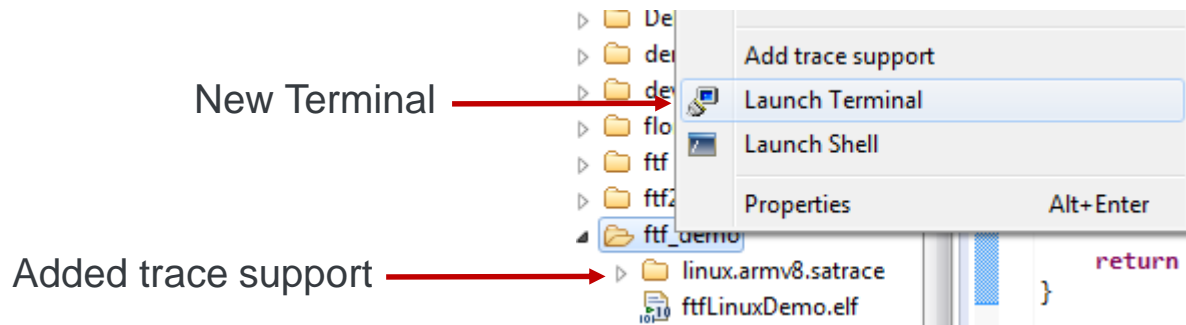


Linux probeless trace (*CodeWarrior*)

9. Add trace support (right click on your newly created folder):

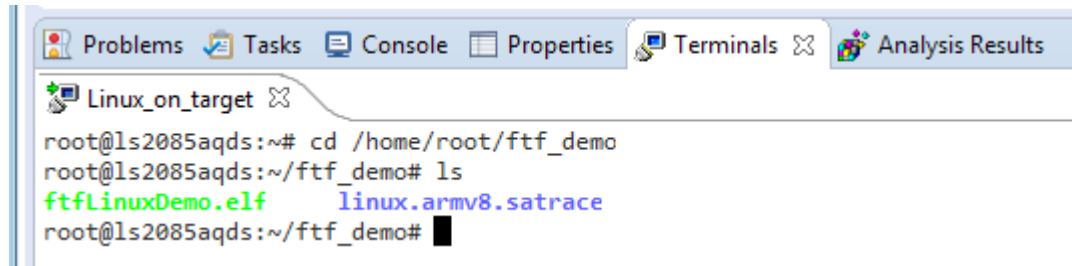


10. Launch a Terminal



Linux probeless trace (CodeWarrior)

11. Run `ls` in Terminal to check the files in your folder:



```
Problems Tasks Console Properties Terminals Analysis Results
Linux_on_target
root@ls2085aqds:~# cd /home/root/ftf_demo
root@ls2085aqds:~/ftf_demo# ls
ftfLinuxDemo.elf  linux.armv8.satrace
root@ls2085aqds:~/ftf_demo#
```

12. Launch application with trace support. Make sure you pass `-v` flag in order to see all stages in verbose mode:

```
./linux.armv8.satrace/bin/ls.linux.satrace -v ./ftfLinuxDemo.elf
```

Linux probeless trace (CodeWarrior)

13. The results will be:

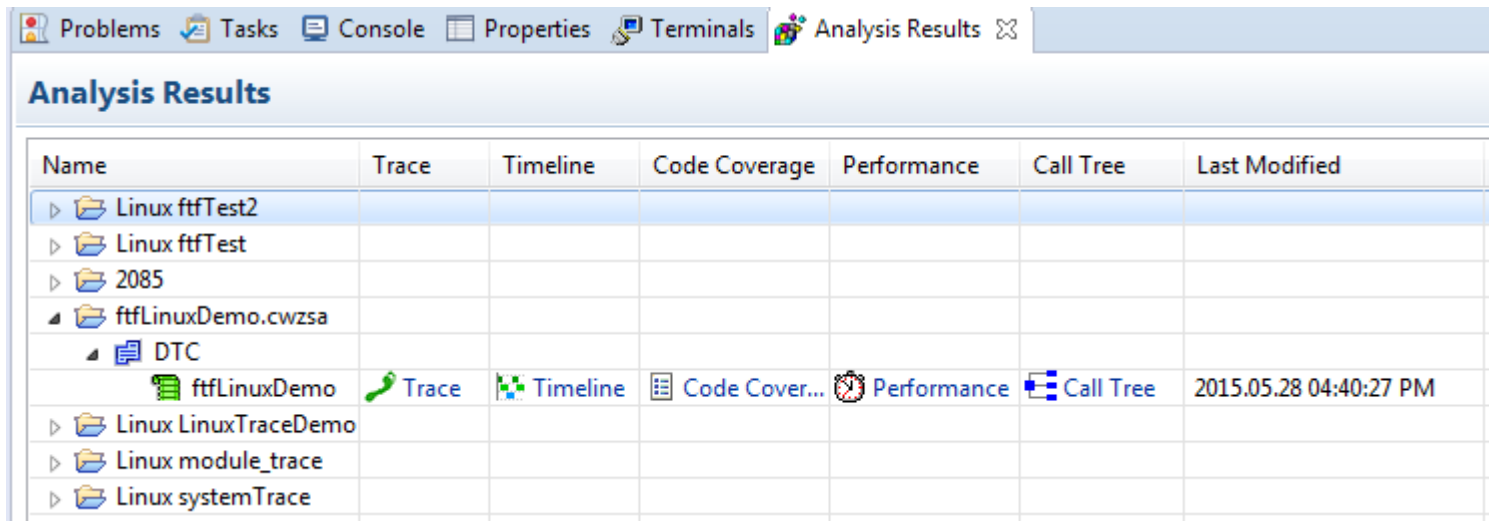
```
root@ls2085aqds:~/ftf_demo# ./linux.armv8.satrace/bin/ls.linux.satrace -v ./ftfLinuxDemo.elf
User space trace
Application : `./ftfLinuxDemo.elf`
Arguments   :
Starting `./ftfLinuxDemo.elf`
Hello World!
User application exit status : 0
Master process
Relocation file : `/home/root/ftf_demo/ftfLinuxDemo.rlog`
Trace file      : `/home/root/ftf_demo/ftfLinuxDemo.dat`
Collecting trace ...
Archive file    : `/home/root/ftf_demo/ftfLinuxDemo.cwzsa`
Creating archive ....
Archiving /home/root/ftf_demo/ftfLinuxDemo.dat
Archiving /home/root/ftf_demo/ftfLinuxDemo.elf
Archiving /home/root/ftf_demo/ftfLinuxDemo.rlog
Archiving /home/root/ftf_demo/linux.armv8.satrace/config/PlatformConfig.xml
Archiving /home/root/ftf_demo/linux.armv8.satrace/bin/ftfLinuxDemo.resultsConfig
Archiving /lib/ld-2.19-2014.04.so
Archiving /lib/libc-2.19-2014.04.so
Archiving /lib/libdl-2.19-2014.04.so
Archiving /lib/libm-2.19-2014.04.so
Archiving /lib/libpthread-2.19-2014.04.so
Archiving /lib/librt-2.19-2014.04.so
root@ls2085aqds:~/ftf_demo# ls
ftfLinuxDemo.cwzsa  ftfLinuxDemo.elf  linux.armv8.satrace
root@ls2085aqds:~/ftf_demo# █
```

Linux probeless trace (CodeWarrior)

14. Refresh the files system view. Notice the newly created *.cwzsa file. Double-click on it to import trace results on host.



15. Notice the Analysis Results view in CodeWarrior. Browse the results and open them.



Linux trace – view results



Analysis Results content

Links

- ✓ Trace Viewer
- ✓ Timeline
- ✓ Code Coverage
- ✓ Hierarchical Performance
- ✓ Call Tree

Platform configuration
used to collect trace

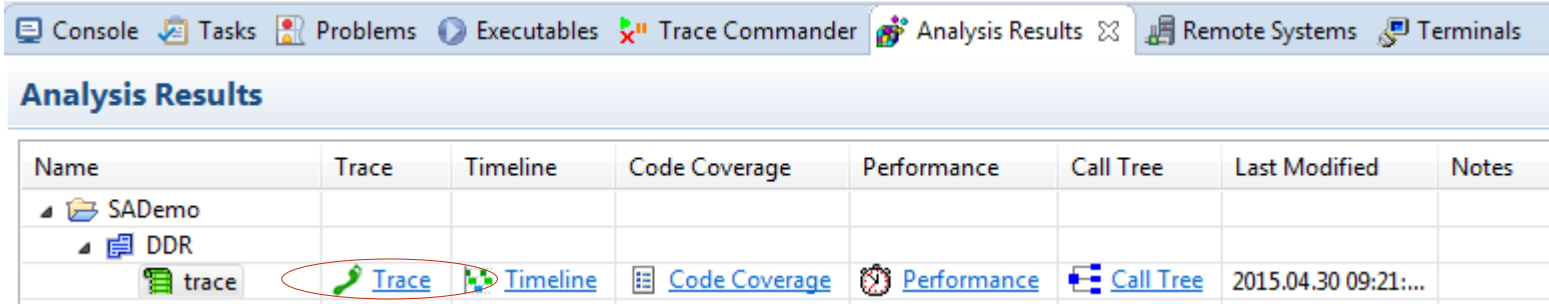
Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
Is2085a							
DDR							
trace	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.04.29 07:13:11 PM	
test_2045							
DTC							
trace	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.04.28 01:23:18 PM	
trace_file.dat							
Imported							
trace_file	Trace	Timeline	Code Coverage	Performance	Call Tree	2015.04.29 07:14:28 PM	

Data source from where the trace was collected:






- DTC
- DDR
- or Imported trace

Trace Viewer

1. From results folder:

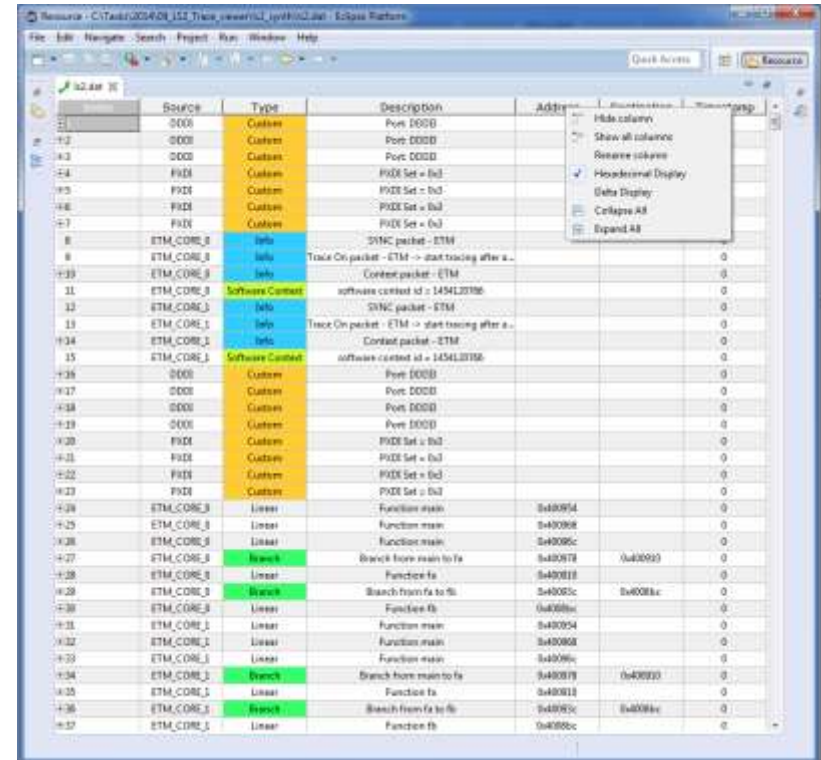


The screenshot shows the Eclipse IDE interface with the 'Analysis Results' view open. The view contains a table with columns: Name, Trace, Timeline, Code Coverage, Performance, Call Tree, Last Modified, and Notes. Under the 'SADemo' folder, there is a sub-folder 'DDR' containing a file named 'trace'. The 'Trace' icon in the 'Trace' column is circled in red.

Name	Trace	Timeline	Code Coverage	Performance	Call Tree	Last Modified	Notes
▲ SADemo							
▲ DDR							
trace						2015.04.30 09:21:...	

2. Open Trace Viewer:

✓ Accurate information about program flow, DDR transactions, instrumentation trace, NoC transactions and PCI Express debug status.

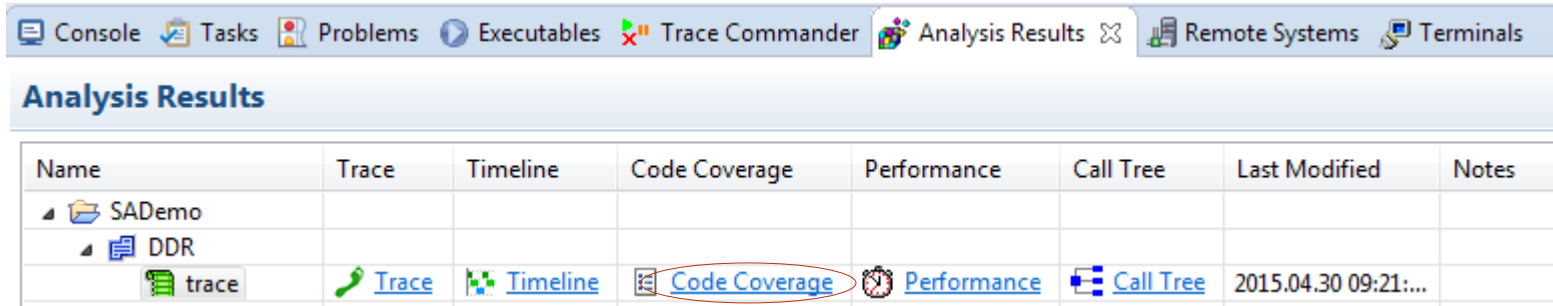


The screenshot shows the Trace Viewer application window. It displays a table with columns: Source, Type, Description, Address, Branch, and Target. The table contains various trace entries, including DDR transactions, ETM packets, and function calls. A context menu is open over the table, showing options like 'Hide columns', 'Show all columns', 'Restore columns', 'Horizontal Display', 'Delta Display', 'Collapse All', and 'Expand All'.

Source	Type	Description	Address	Branch	Target
0000	Custom	Port: D000			
0000	Custom	Port: D000			
0000	Custom	Port: D000			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
ETM_CORE_0	Info	ETM packet - ETM			
ETM_CORE_0	Info	Trace On packet - ETM -> start tracing after a...			
ETM_CORE_0	Info	Context packet - ETM			
ETM_CORE_0	Software Comment	software context id = 145420396			
ETM_CORE_0	Info	ETM packet - ETM			
ETM_CORE_0	Info	Trace On packet - ETM -> start tracing after a...			
ETM_CORE_0	Info	Context packet - ETM			
ETM_CORE_0	Software Comment	software context id = 145420396			
D000	Custom	Port: D000			
D000	Custom	Port: D000			
D000	Custom	Port: D000			
D000	Custom	Port: D000			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
F000	Custom	P000 Set = 0x0			
ETM_CORE_0	Linear	Function main	0x400054		
ETM_CORE_0	Linear	Function main	0x400060		
ETM_CORE_0	Linear	Function main	0x40006c		
ETM_CORE_0	Branch	Branch from main to fa	0x400078	0x400090	
ETM_CORE_0	Linear	Function fa	0x400078		
ETM_CORE_0	Branch	Branch from fa to fb	0x40007c	0x40008c	
ETM_CORE_0	Linear	Function fb	0x40007c		
ETM_CORE_0	Linear	Function main	0x400054		
ETM_CORE_0	Linear	Function main	0x400060		
ETM_CORE_0	Linear	Function main	0x40006c		
ETM_CORE_0	Branch	Branch from main to fa	0x400078	0x400090	
ETM_CORE_0	Linear	Function fa	0x400078		
ETM_CORE_0	Branch	Branch from fa to fb	0x40007c	0x40008c	
ETM_CORE_0	Linear	Function fb	0x40007c		

Code coverage

1. From results folder:

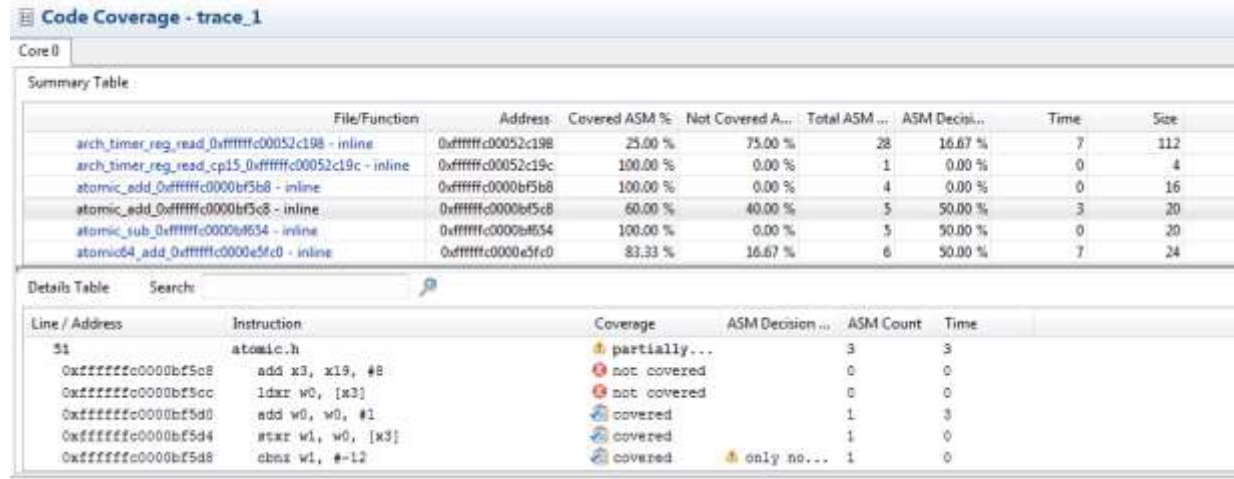


2. Open code coverage viewer:

Split pane with 2 types of info:

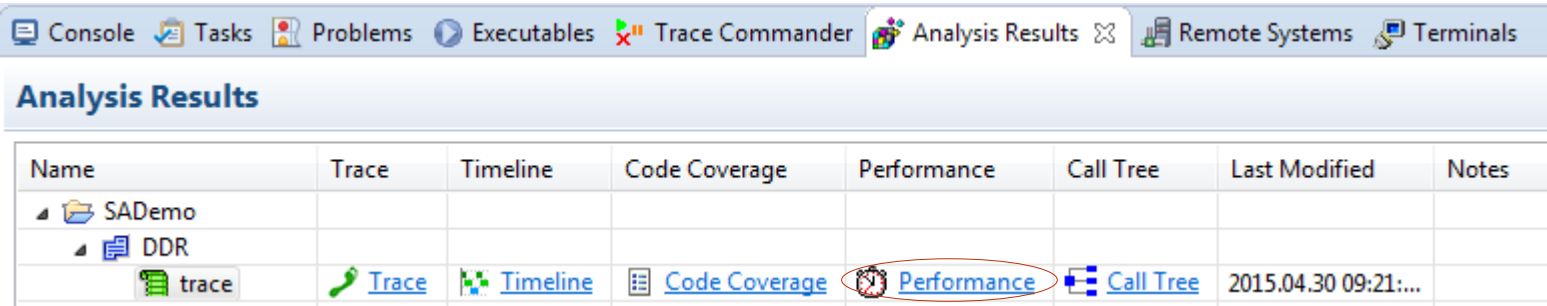
✓ Summary table displaying statistics for each function

✓ Details table displaying line-by-line coverage of selected function



Performance profiler

1. From results folder:

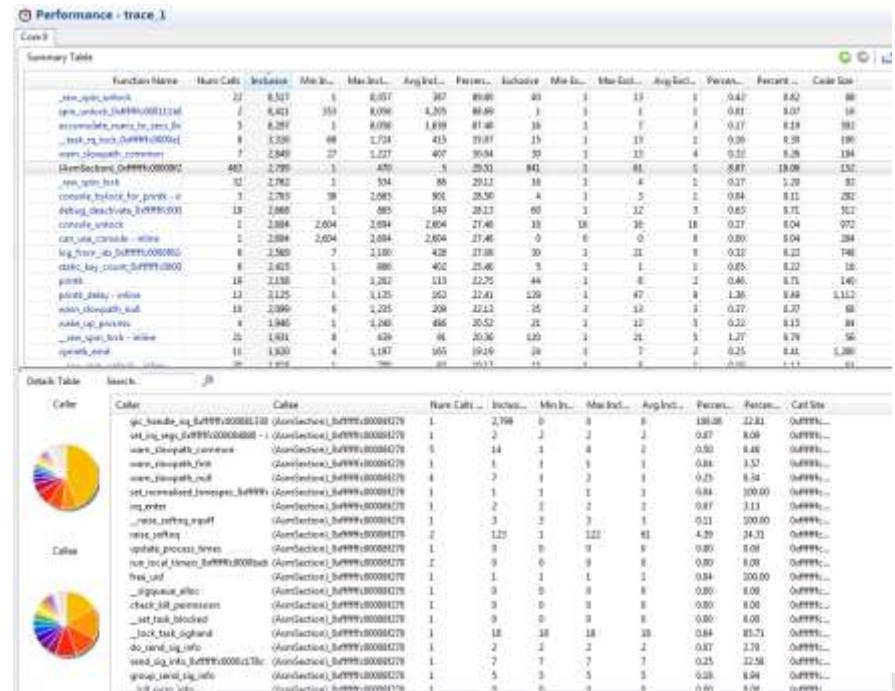


2. Open performance viewer:

✓ Per core analysis

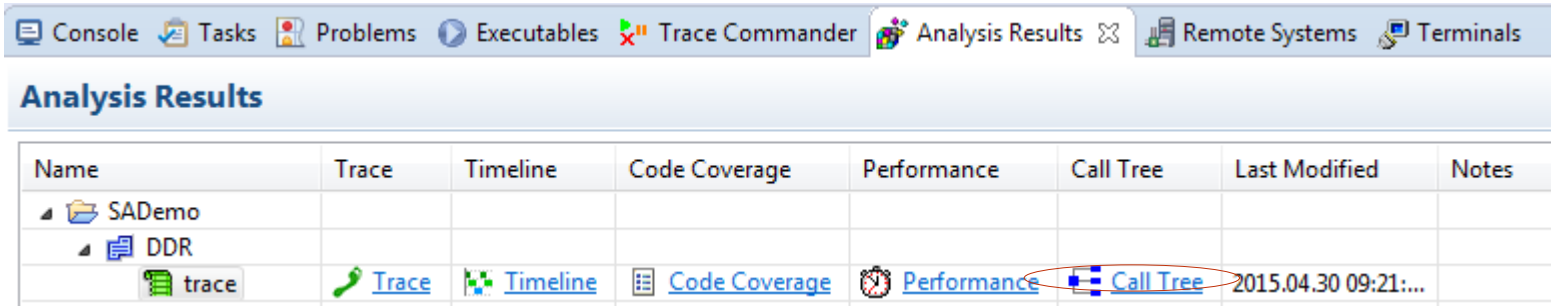
✓ Split pane with 2 types of information:

- Summary table displaying profiling values for functions executed in each context
- Details table displaying performance values for caller and callee



Call tree profiler

1. From results folder:



2. Open call tree viewer:

Shows call tree of executed functions. Two highlighted paths:

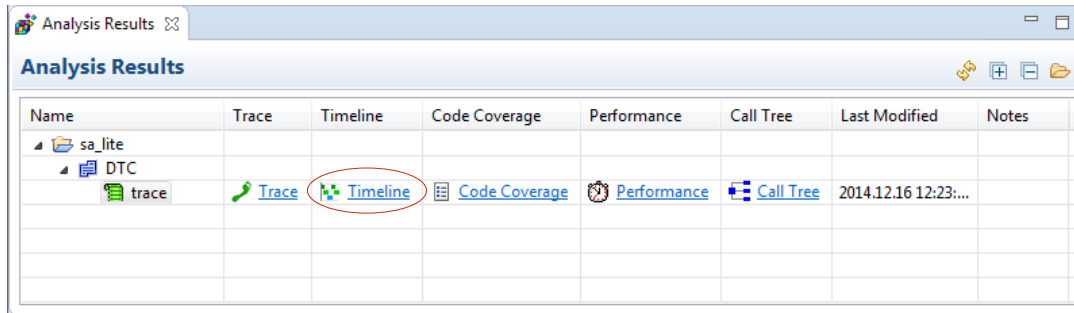
- ✓ Green color shows critical path
- ✓ Grey background shows max stack path

The screenshot shows the 'Call Tree - trace_1' window. It displays a table with the following columns: Function Name, Num Calls, % Total calls of parent, % Total times it was called, and Inclusive Time (Cycles). The table shows a call tree for 'Context 549'. Two paths are highlighted: one in green and one in grey.

Function Name	Num Calls	% Total calls of parent	% Total times it was called	Inclusive Time (Cycles)
Context 549				
f (AsmSection)_0xfffffc00008f270	1	100.00	100.00	9,484
f gic_read_iar - inline	1	16.67	0.21	2,799
f irq_find_mapping	1	16.67	33.33	27
f (AsmSection)_0xfffffc00008f270	1	100.00	0.21	2,799
f set_irq_regs_0xfffffc000084880 - inline	1	16.67	100.00	9,410
f (AsmSection)_0xfffffc00008f270	1	25.00	100.00	25
f __this_cpu_preempt_check	1	20.00	0.21	2,799
f check_preemption_disabled	1	20.00	4.00	600
f current_thread_info_0xfffffc000039	1	100.00	3.03	554
f _my_cpu_offset_0xfffffc00008489c - inl	1	100.00	4.76	0
f __this_cpu_preempt_check	1	20.00	100.00	0
f __this_cpu_preempt_check	1	20.00	4.00	600
f check_preemption_disabled	1	---	---	---

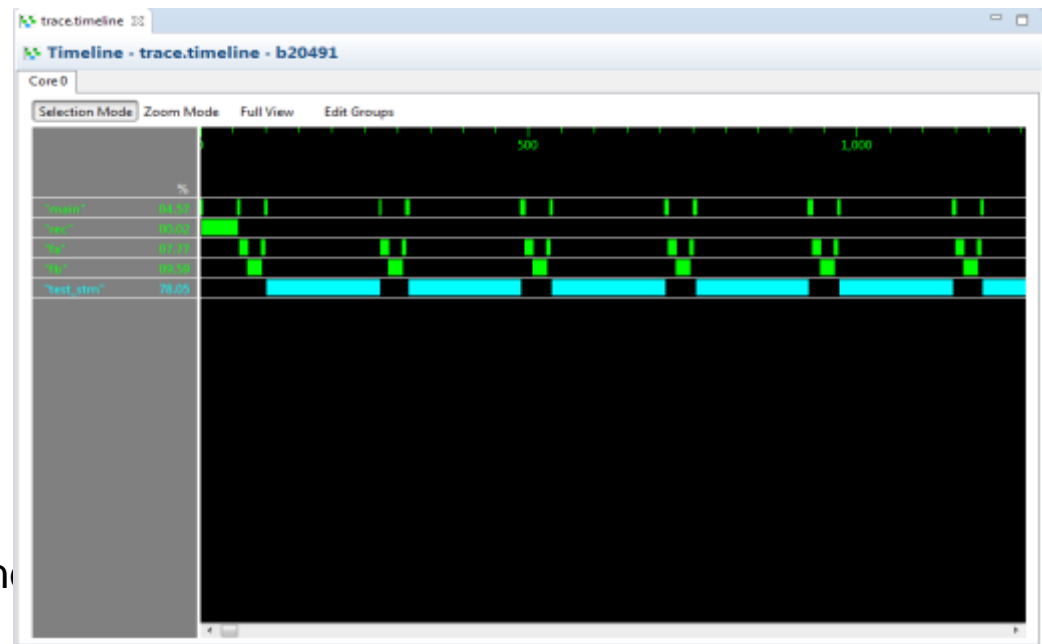
Timeline

1. From Analysis Results view:



2. Open Timeline viewer:

- ✓ Organizes multicore results in tabs
- ✓ Customize the way the data is drawn
- ✓ Execution timeline of functions and custom groups
- ✓ Spot performance problems in code and





Summary

- This course has been a brief introduction into the LS2085 RDB board and the CodeWarrior tools available to debug the board
- Linux kernel debug
- Linux application debug
- Trace
- Digital Networking is introducing a new networking tools suite
 - CodeWarrior Development Studio for QorIQ LS Series – ARMv8 ISA
 - Tools covering Configuration, Build, Debug, and Analysis



Q&A





www.Freescale.com

BACKUP

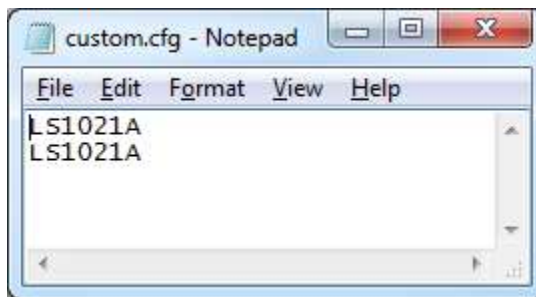


Custom Board Setup

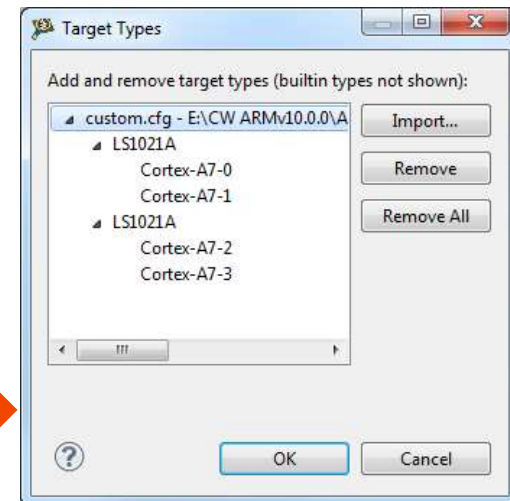
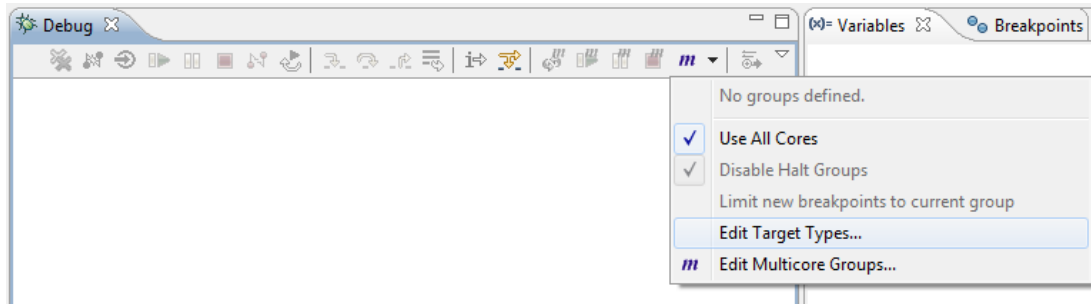


Stock Freescale Boards vs. Customer Designs

- For stock Freescale boards, CW comes with all the necessary initialization / configuration files
- For custom designs, first, using a JTAG configuration file, define the list of devices on the JTAG chain.
 - As example, for a board with two QorIQ LS2085A processors, JTAG configuration file is:

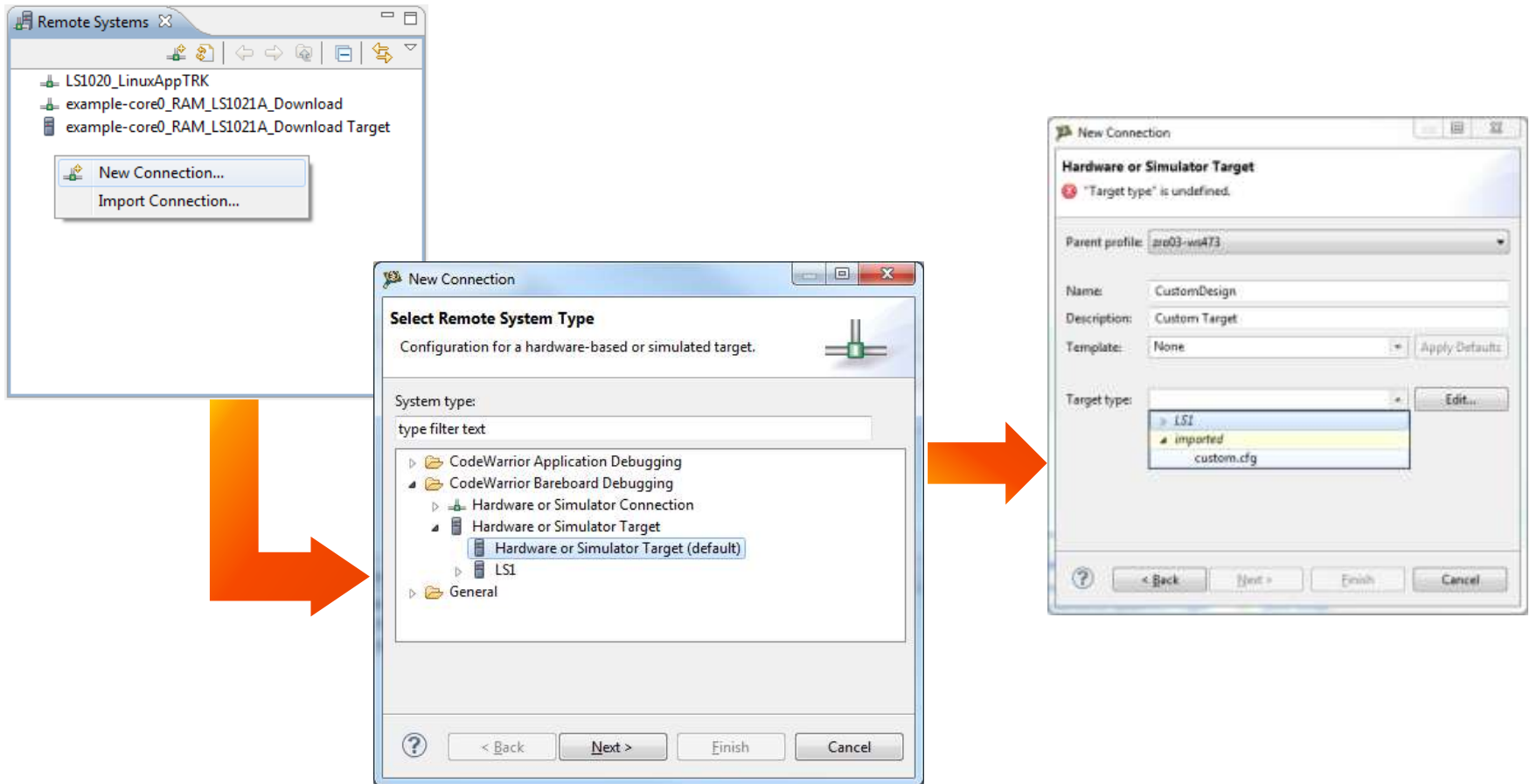


- Import it.



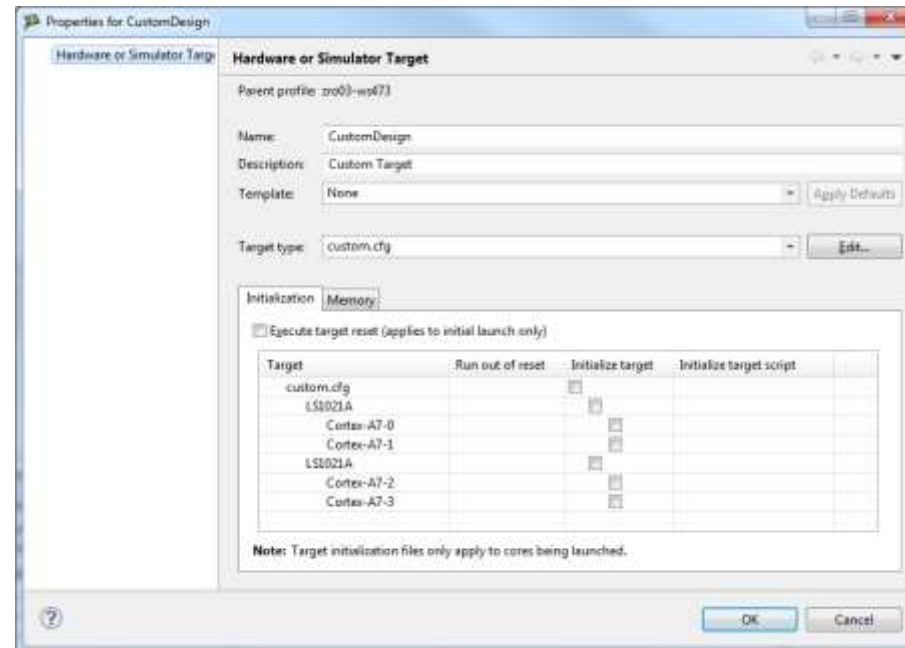
Creating a Custom Target System

- Create a new target system



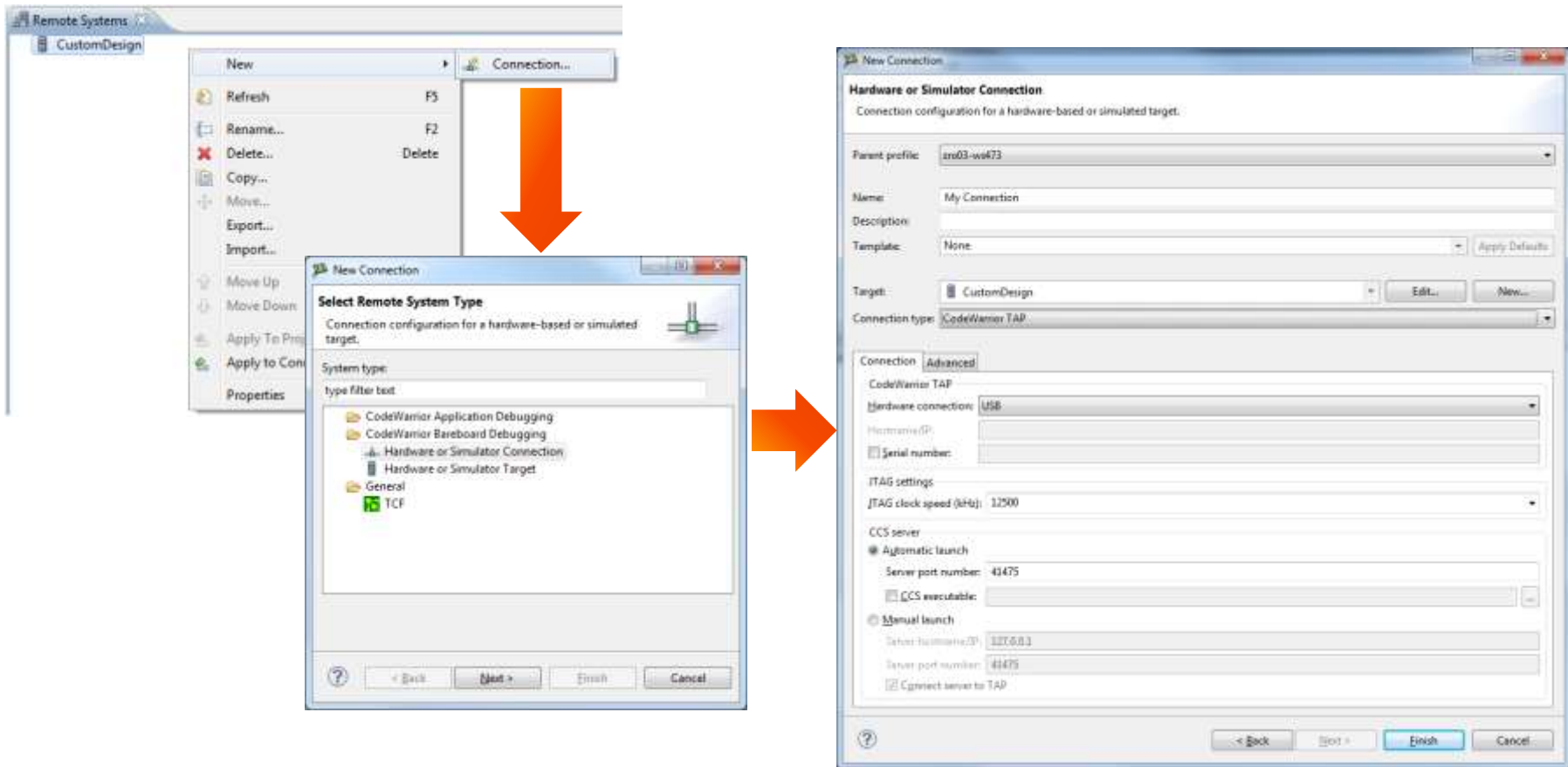
Configure the Newly Created Target

- Check “Execute target reset” if you want to reset the SoC when first core connects.
- Select initialization files (for DDR, internal memories, flash controllers, etc.) and on what core(s) they should be executed.
- Select memory configuration files. Consider that the memory map may differ since, for example, some IP blocks may not be present in the new design.



New Connection & Debug

- Create a new connection and select for it the “Target” system that was just created.
- Apply the new connection to your project(s)





www.Freescale.com