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

J U N E . 2 0 1 5

freescale™

External Use

Layerscape LS2085A Software and Tools Enablement

CodeWarrior Development Studio

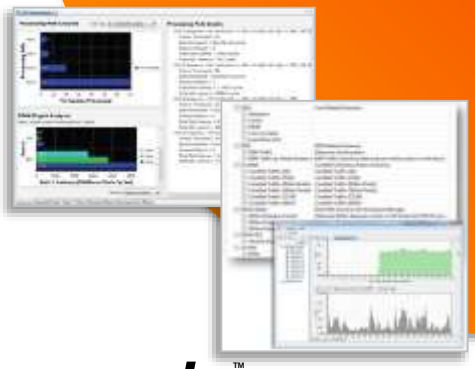
Software Development Tools

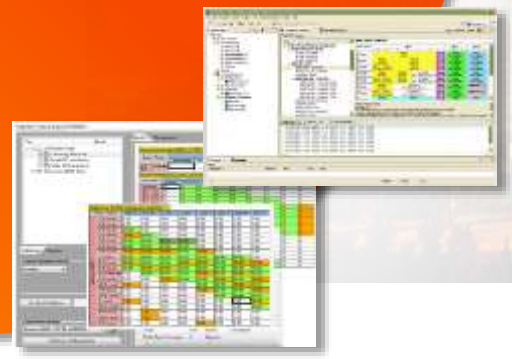
QorIQ Linux SDK

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

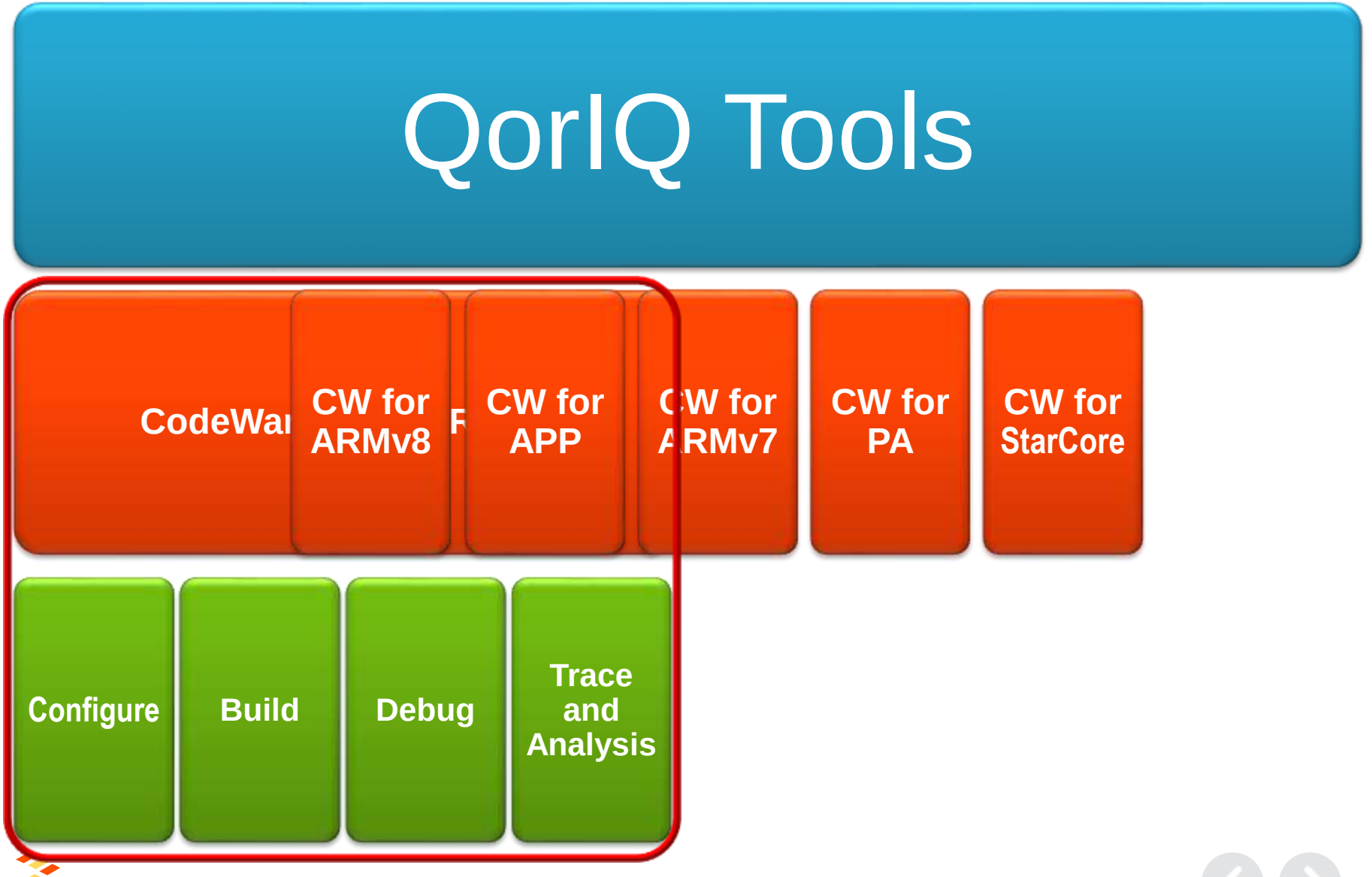
Agenda

- 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

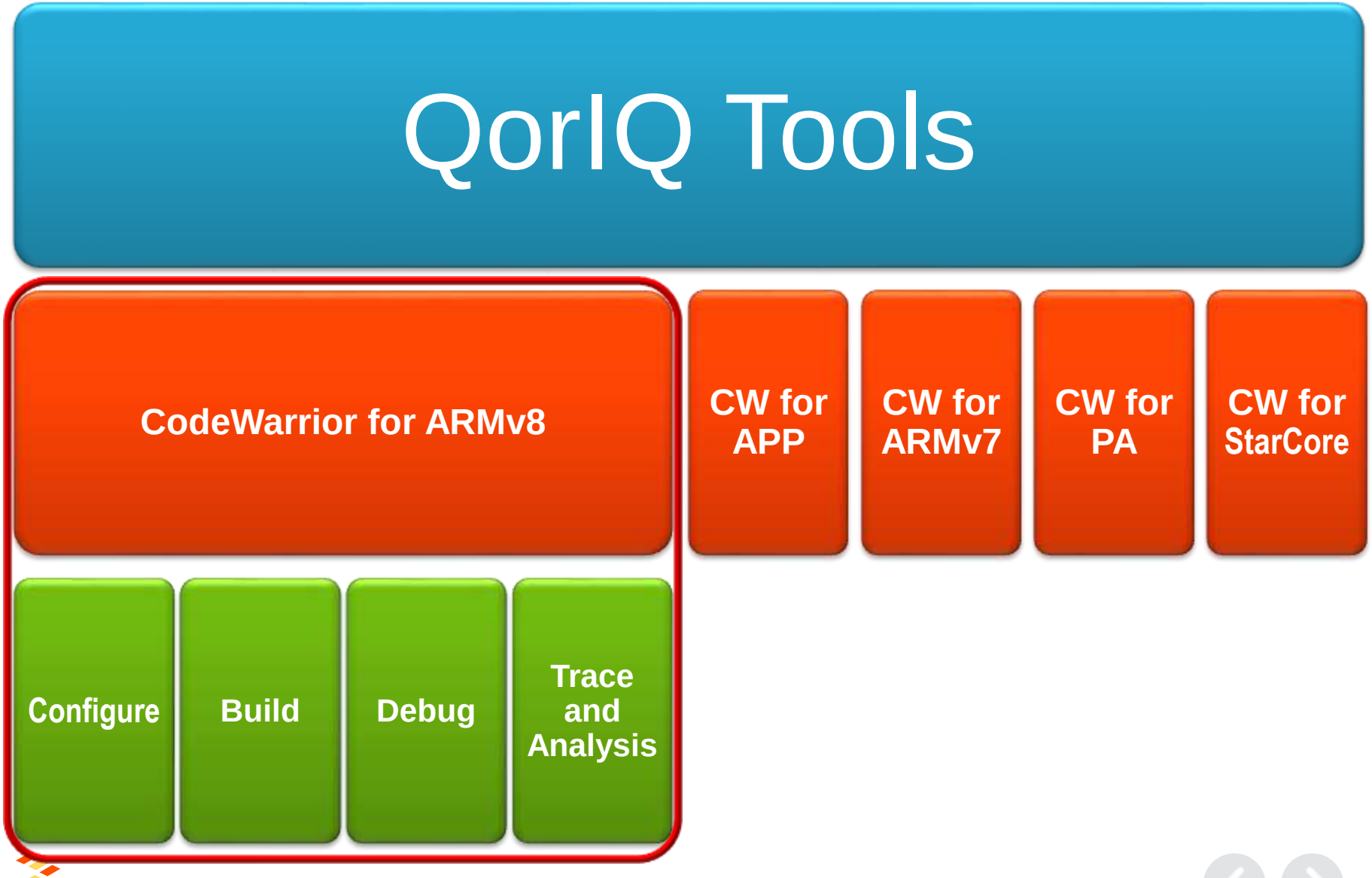
Layerscape Tools

CodeWarrior for ARMv8 ISA
CW-TAP

CodeWarrior Family



CodeWarrior Family



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

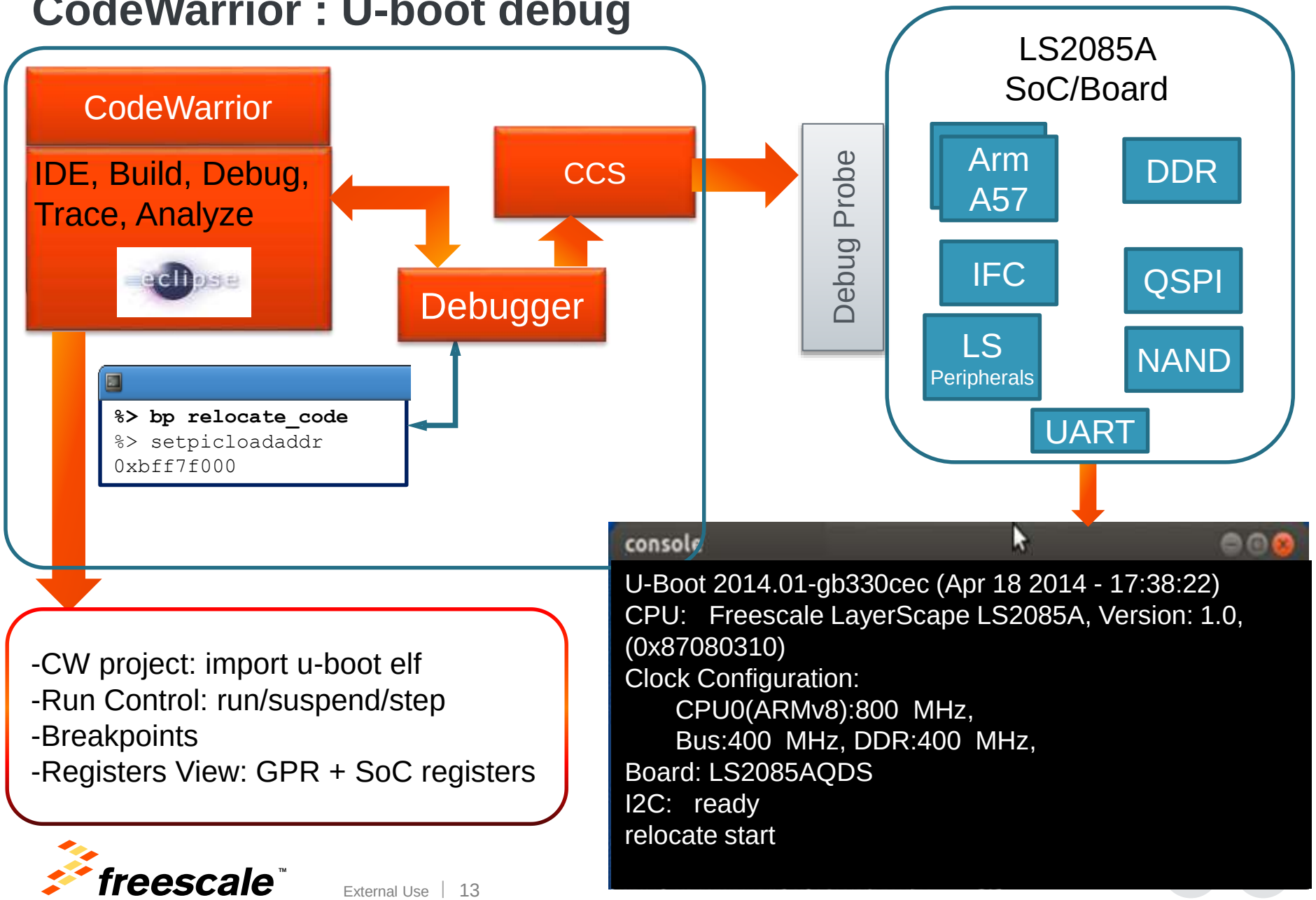
CodeWarrior Aids Debug Through Multiple Phases (3)

- 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



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

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)

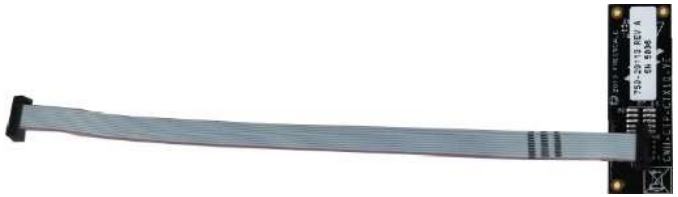
- Reminder : no USB-TAP support

CodeWarrior TAP

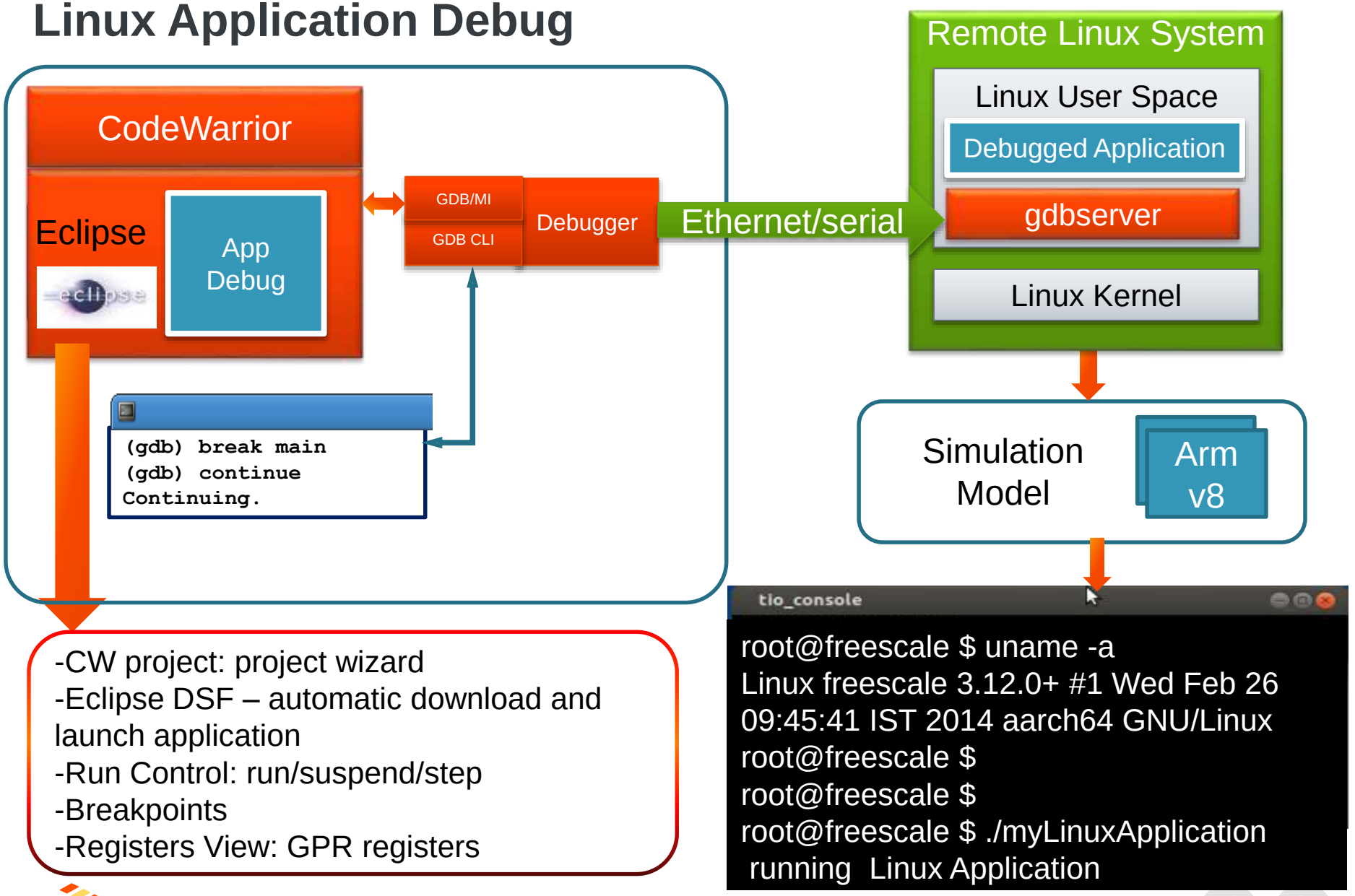
- JTAG debugging and CodeWarrior run control requires :

  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



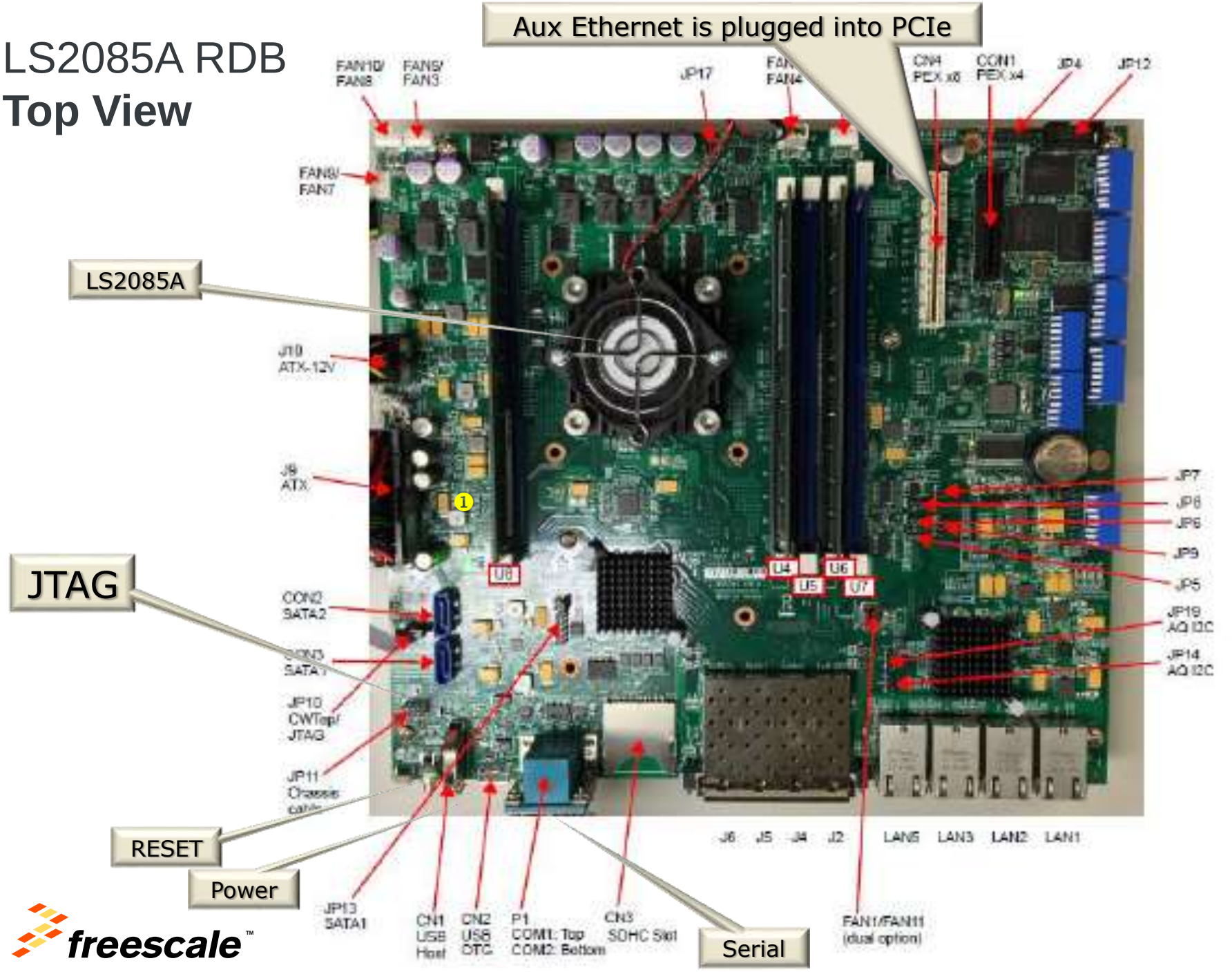
Linux Application Debug



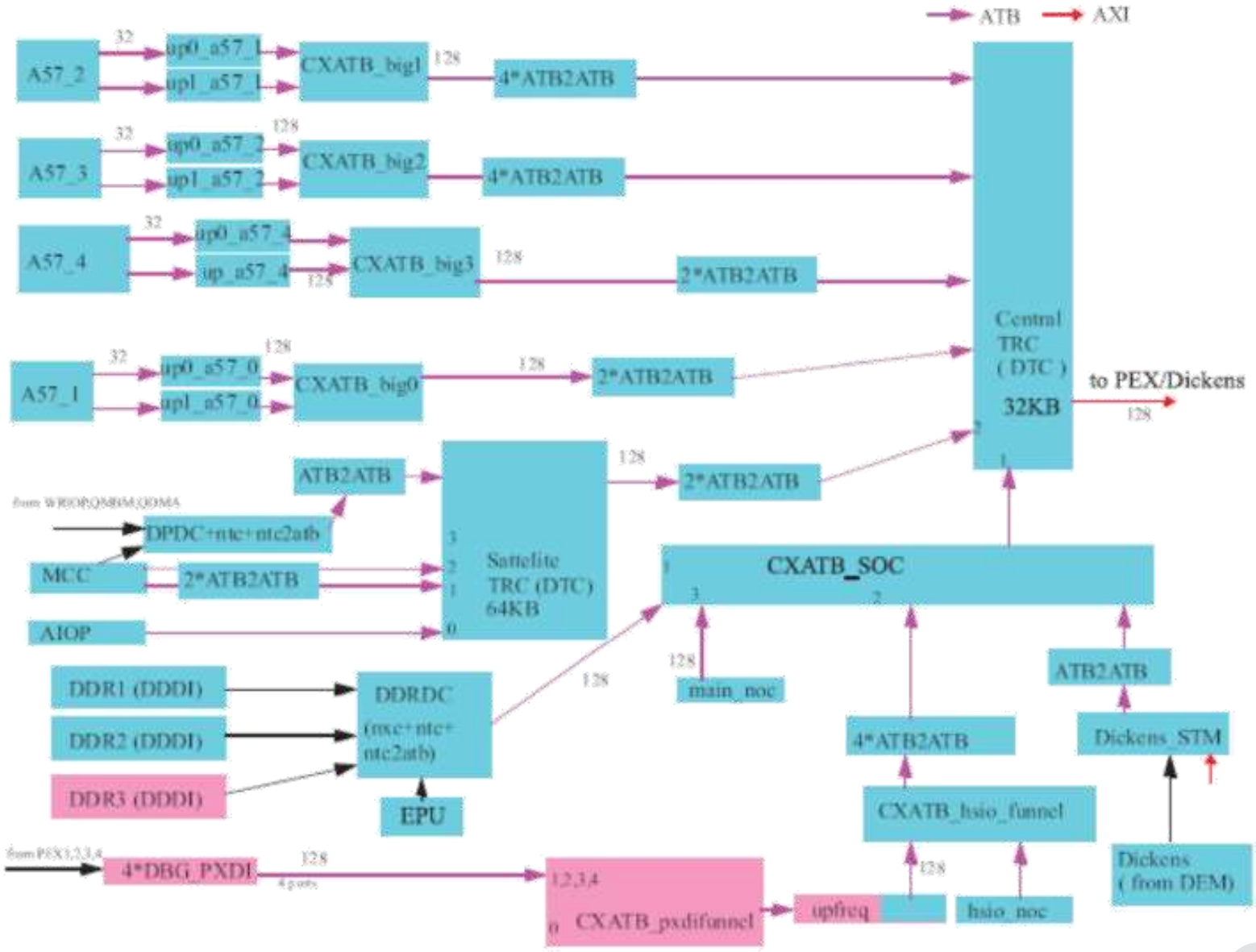
-CW project: project wizard
-Eclipse DSF – automatic download and launch application
-Run Control: run/suspend/step
-Breakpoints
-Registers View: GPR registers

Introducing the LS2085A RDB

LS2085A RDB
Top View



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)

USB<->Serial

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.

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.
```

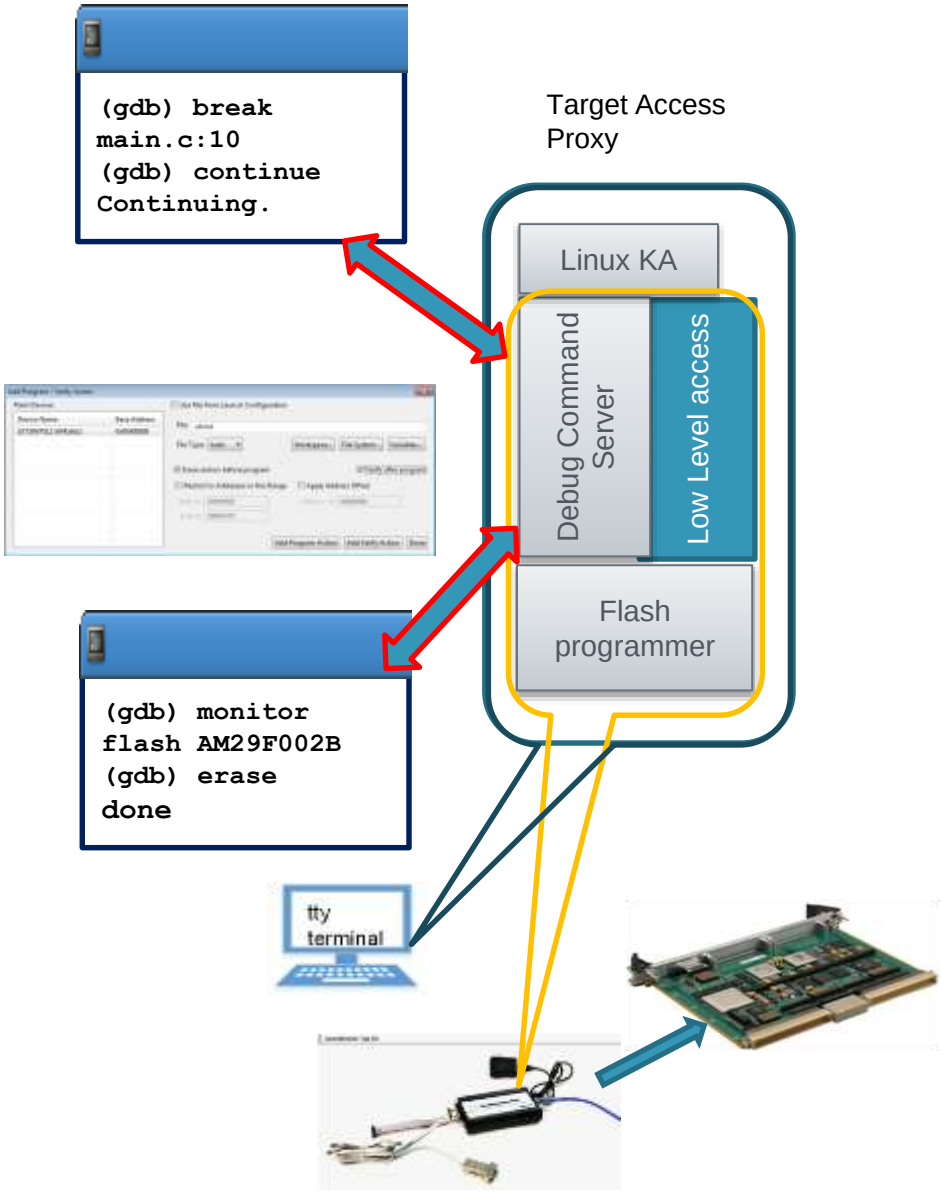
- Establishing network connectivity in your network environment :

```
=> 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

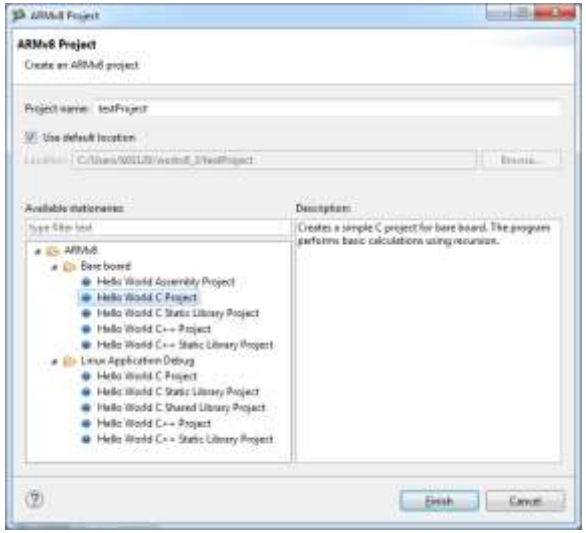
- Target interface to real hardware / simulator
- Lightweight debugger engine accessible from both GUI and command line
- Compatible with the GNU debugger front-end
- Standard set of memory/register access commands + monitor extensions
- Simultaneous connectivity with multiple clients
- Single- and Multi-core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



Activity

Create a CodeWarrior Bareboard Project

Activity - CodeWarrior Bareboard Debugging
Bareboard Debugging with *CodeWarrior TAP* Connection



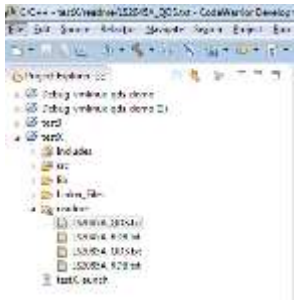
- Activity Summary:

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***.

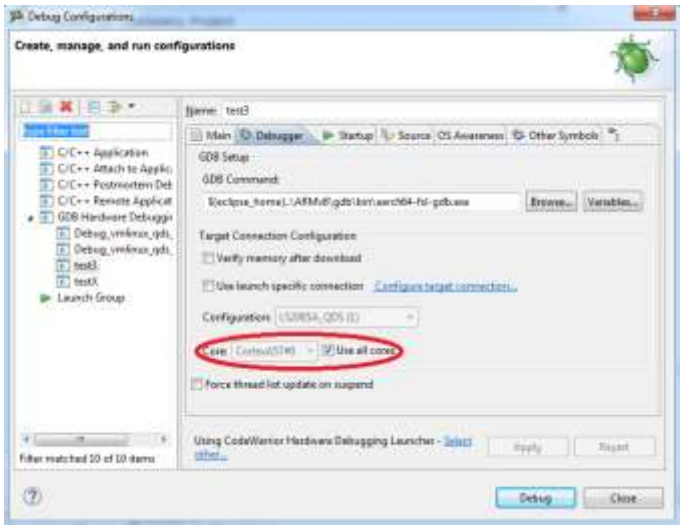


- You will find out about:
  - ***Boot*** options switch configurations
  - Default ***RCW***
  - ***Memory Map*** with accessible areas
  - ***SMP*** debug configuration
  - ***Console I/O*** or ***UART I/O*** selection
  - Other useful info

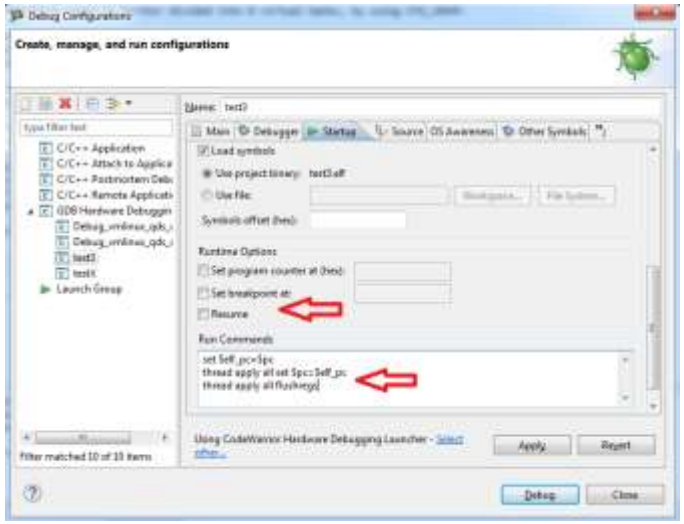
Create bareboard project – Activity (cont.)

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



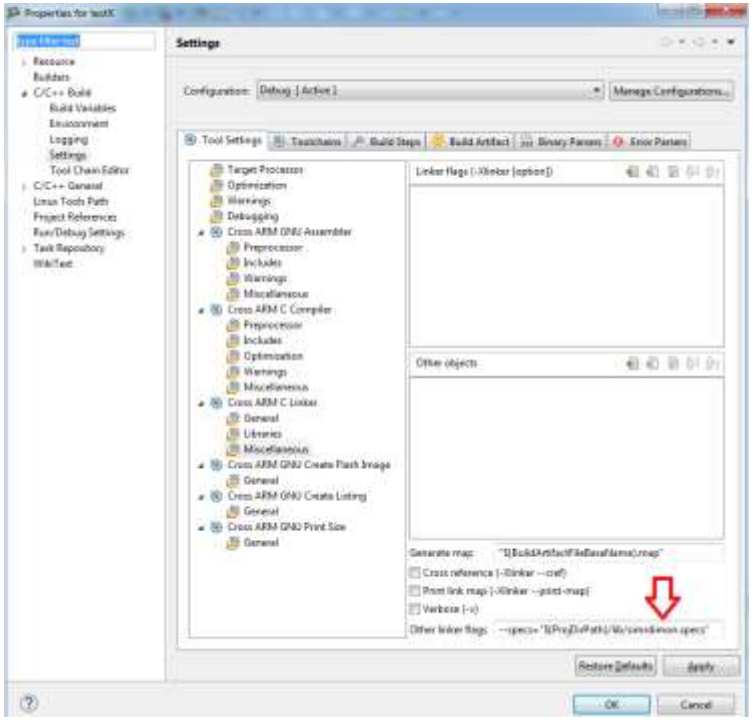
- Enter the following commands in the *Run Commands* section:
  - set $self_pc=$pc
  - thread apply all set $pc=$self_pc
  - thread apply all flushregs

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



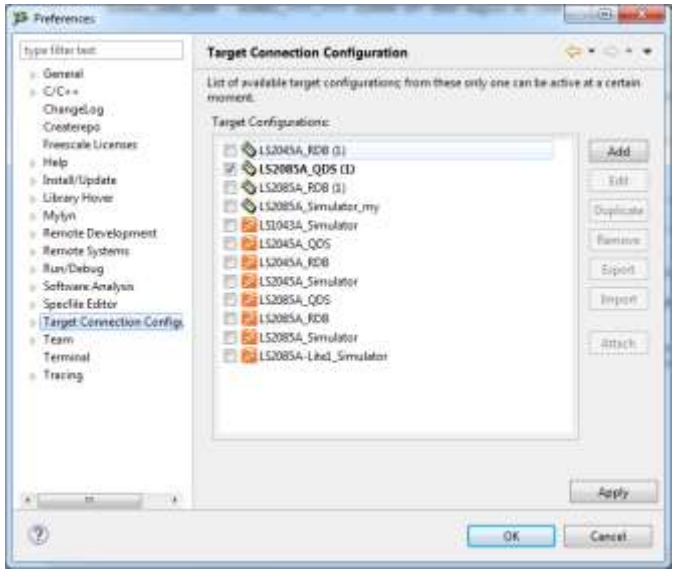
Activity

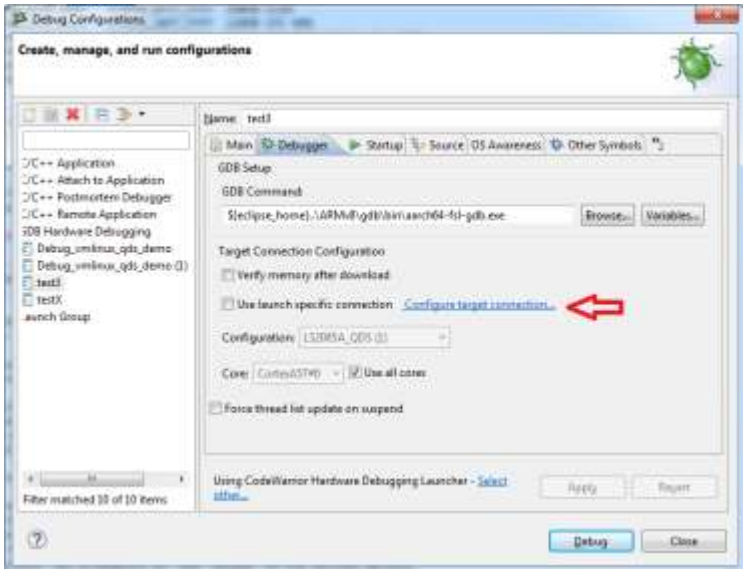
Establish a CodeWarrior Connection to the Target

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…*

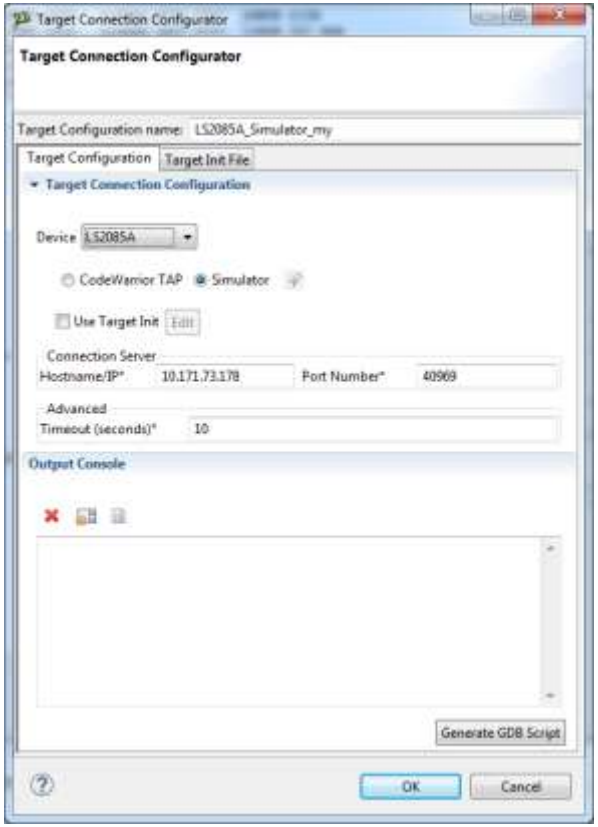




Hardware Selection and Configuration Activity

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 ( )



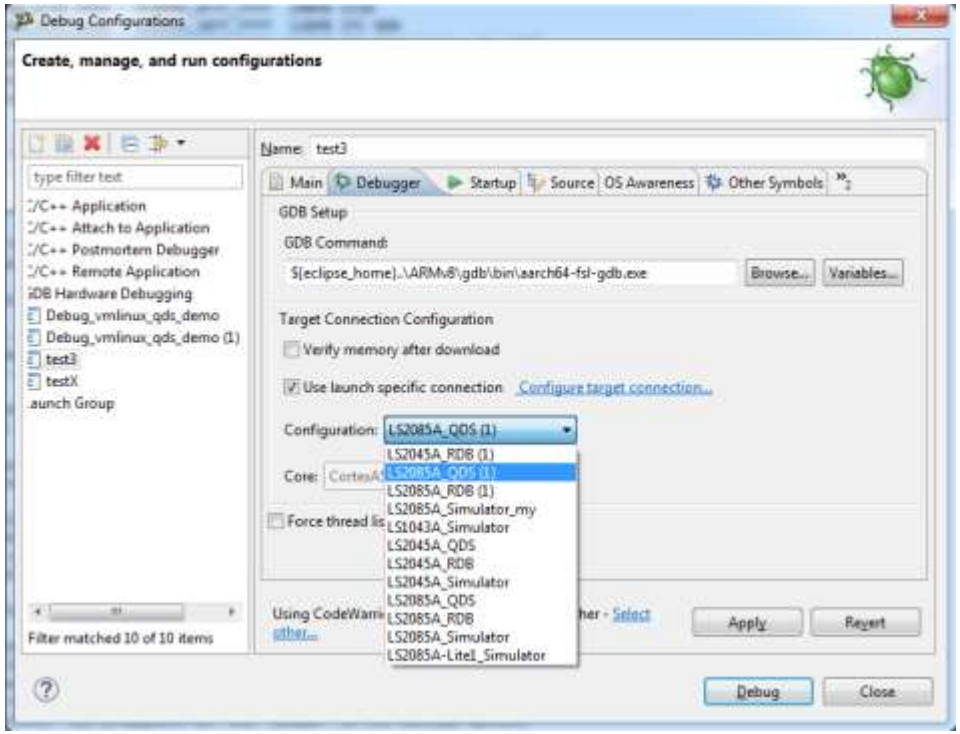
- You can change:
  - **device** type
  - **connection** settings
  - **debug probe** type

- 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), expect 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

cwflash.py

```
######################################################################
#########  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
```

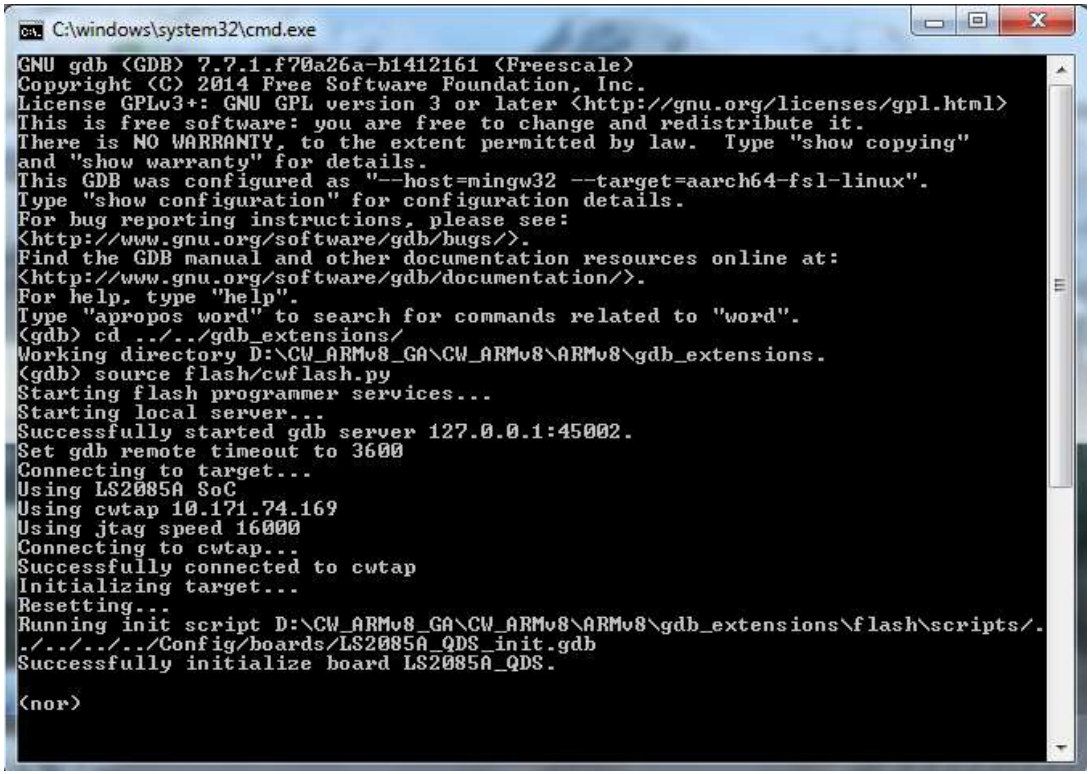
Flashing u-boot image to the Target Activity (5)

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



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

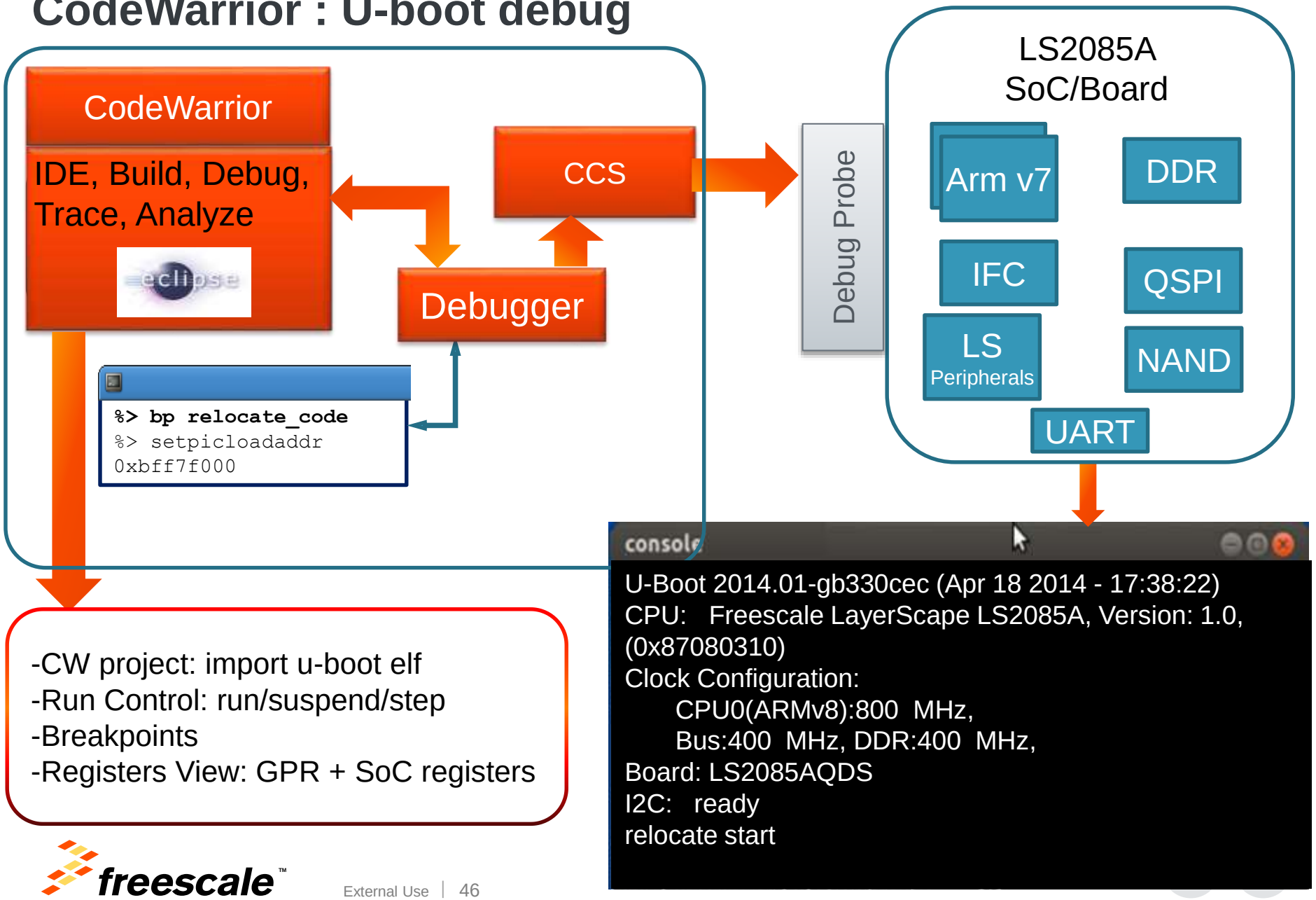
Using CW Flash command line interface (cont.)

- 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**

U-Boot Debug

CodeWarrior : U-boot debug



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

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

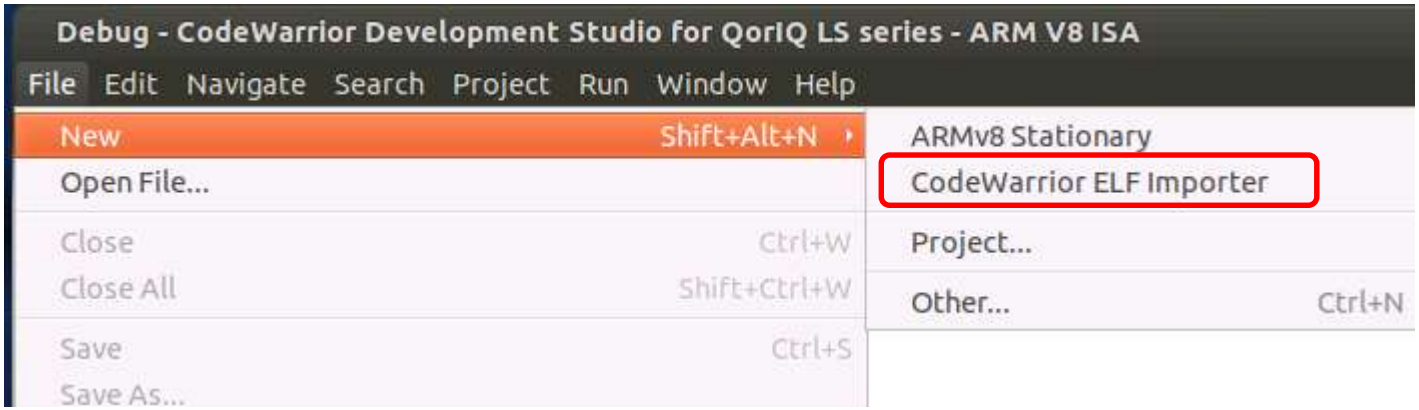
- Prerequisite
  - U-boot image (optionally with symbolic information, useful for source level debug)

Activity

UBOOT- Debug

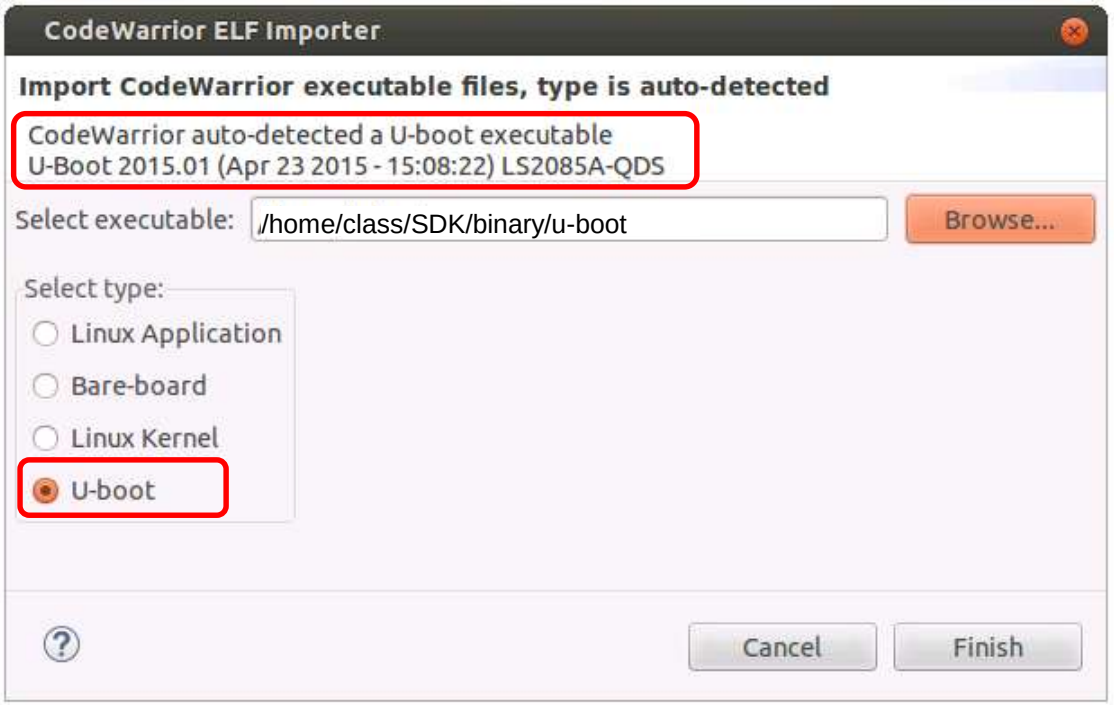
U-boot debug – Create project – Activity

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



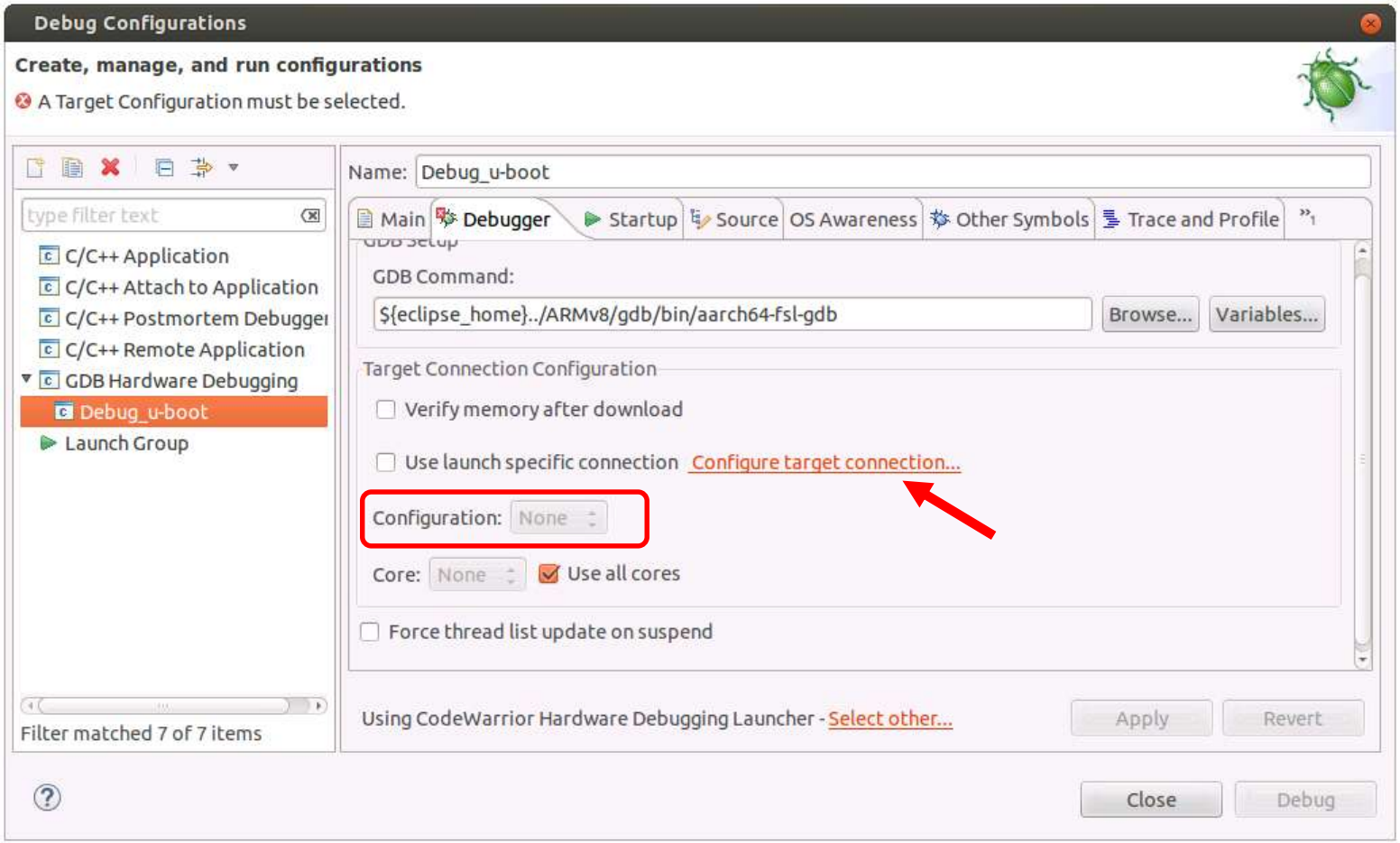
U-boot debug – Create project – Activity

- 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



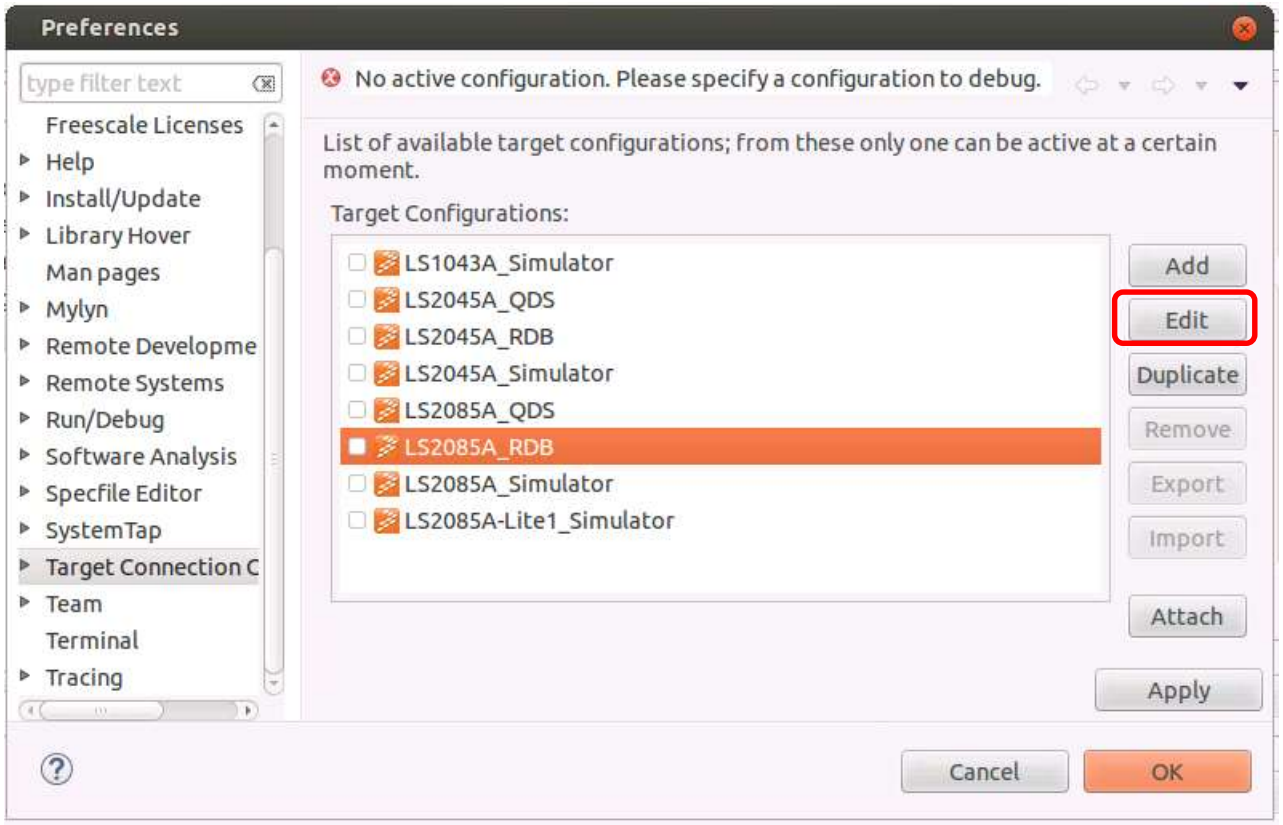
U-boot debug – Create Target Connection – Activity

- 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

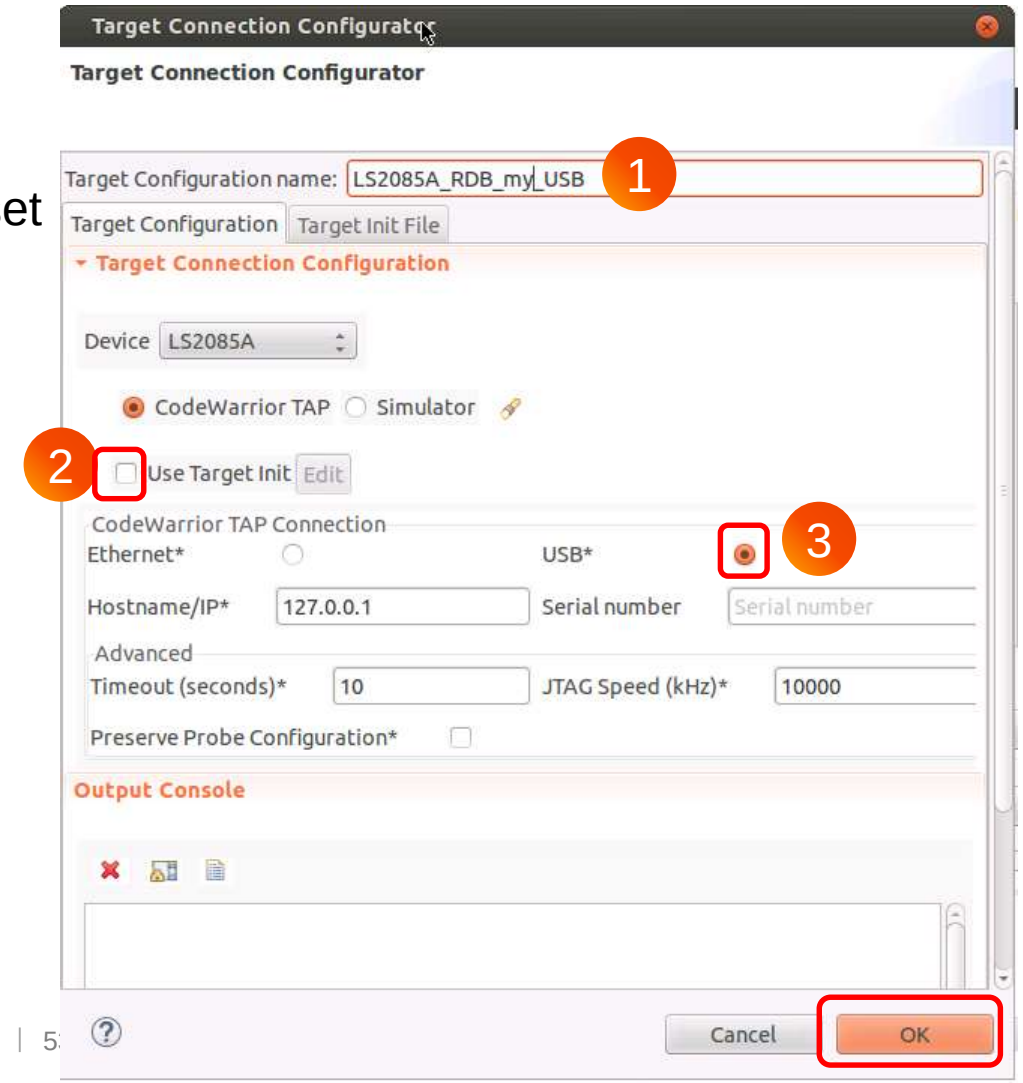
- 1. Edit the name (different than the template name)
- 2. Un-check “Use Target Init” checkbox

- 3. Select USB

or

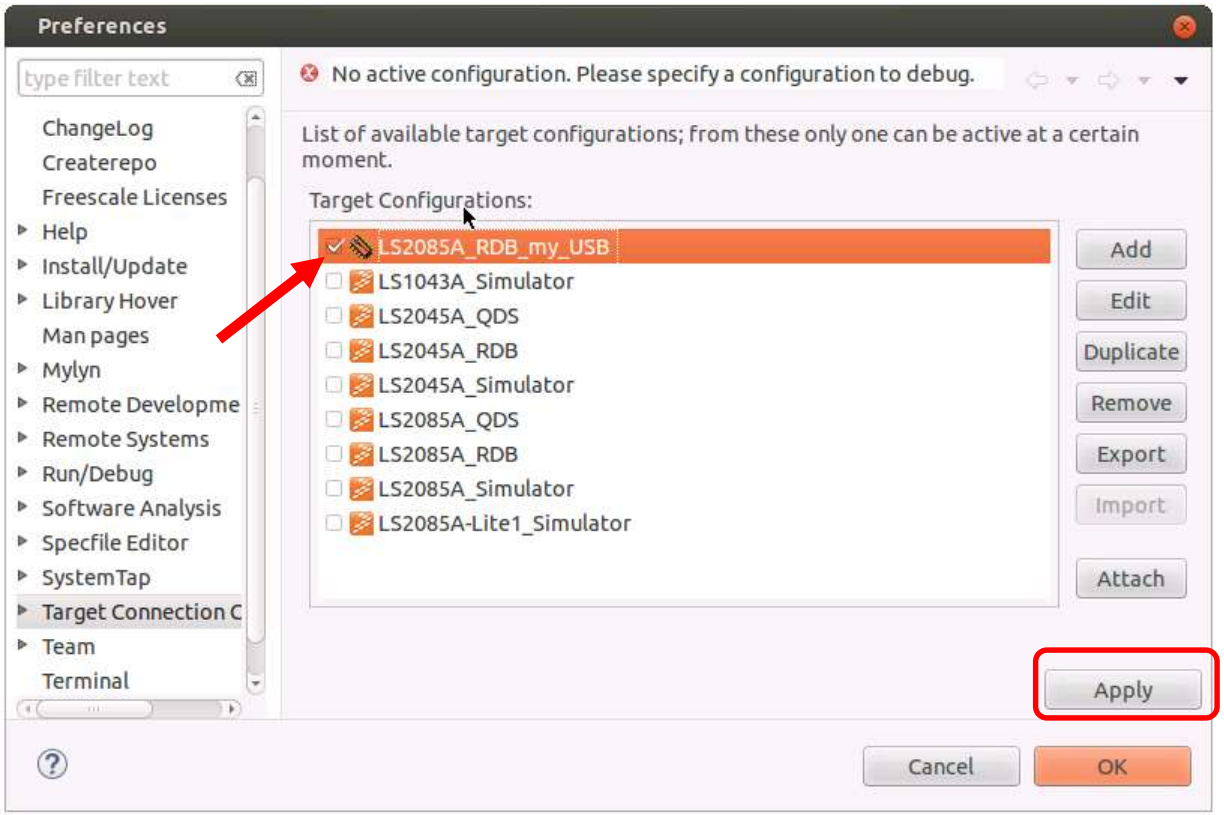
- 3. Select Ethernet and set the correct CWTAP IP

- Press OK



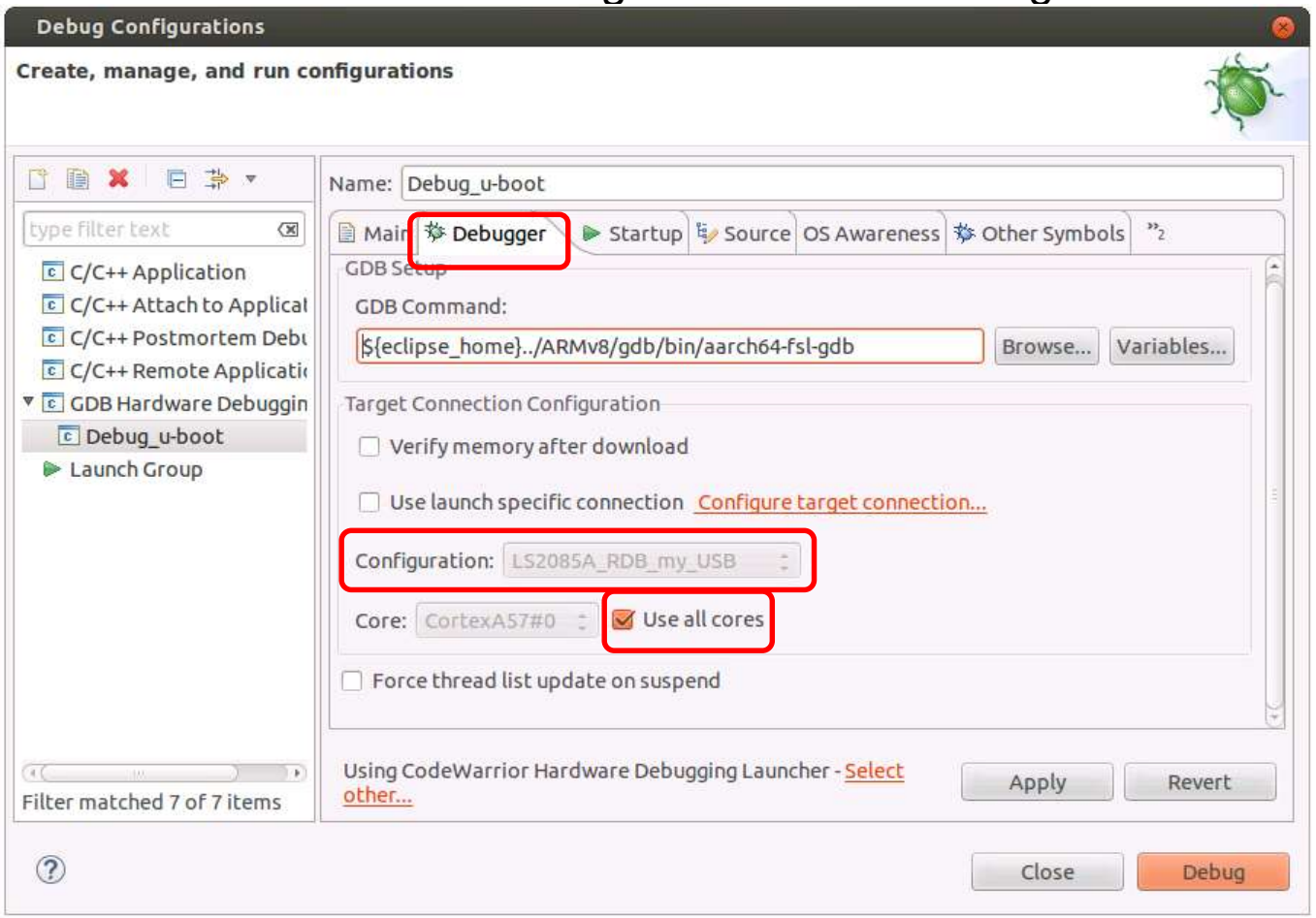
U-boot debug – Create Target Connection – Activity

- 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

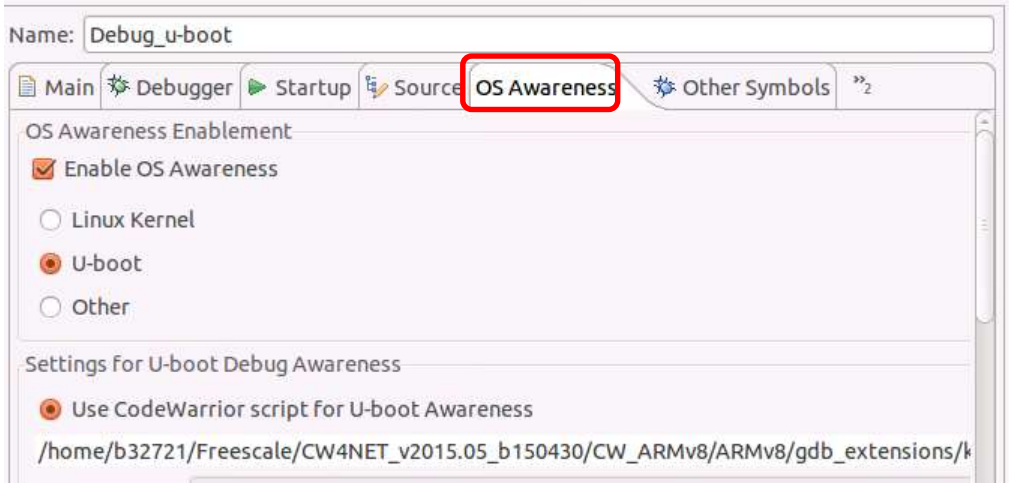
- **Debugger Tab**
  - Configuration automatically set to the default Target Connection
  - Uncheck “Use all cores” to do single core u-boot debug



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



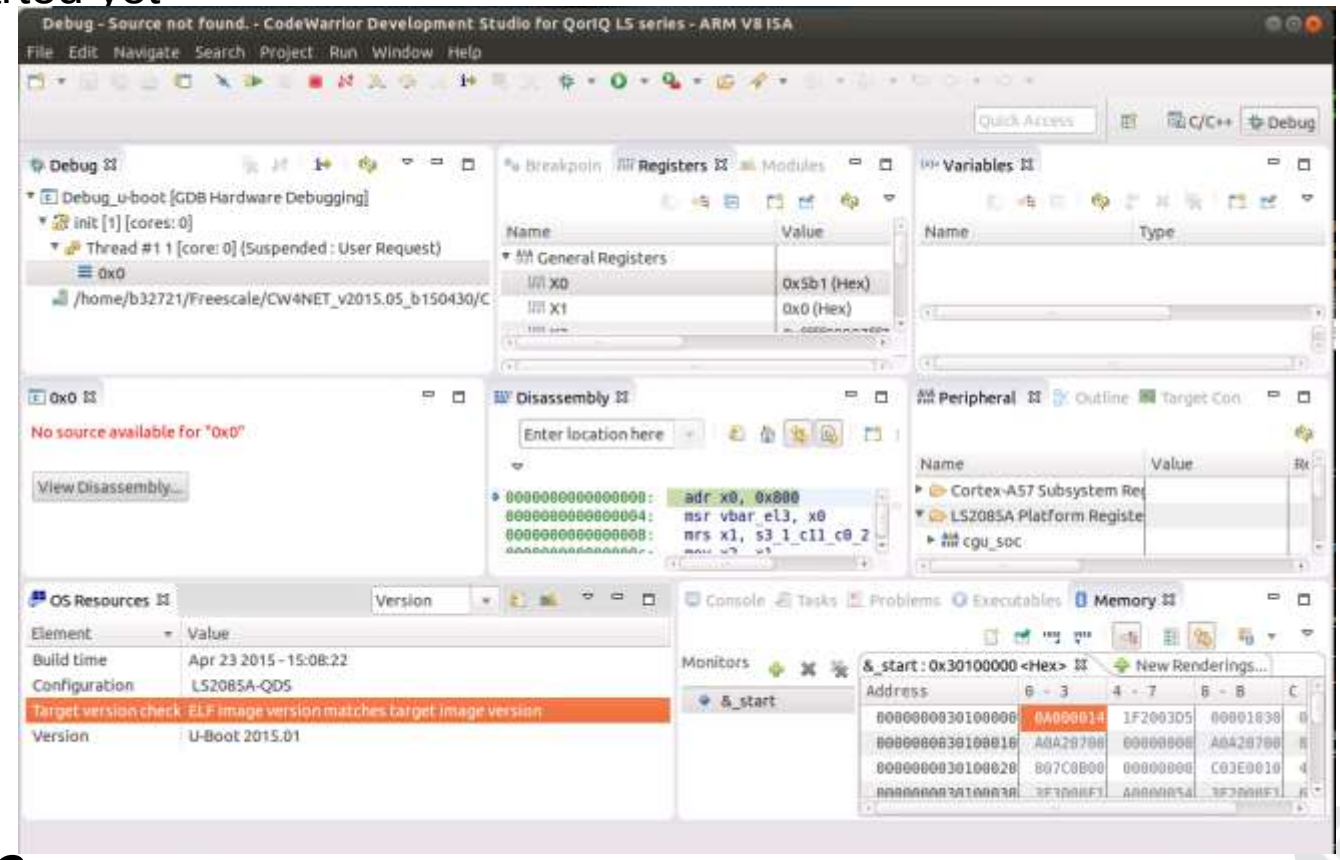
U-boot debug – Debug Overview – Activity

- **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

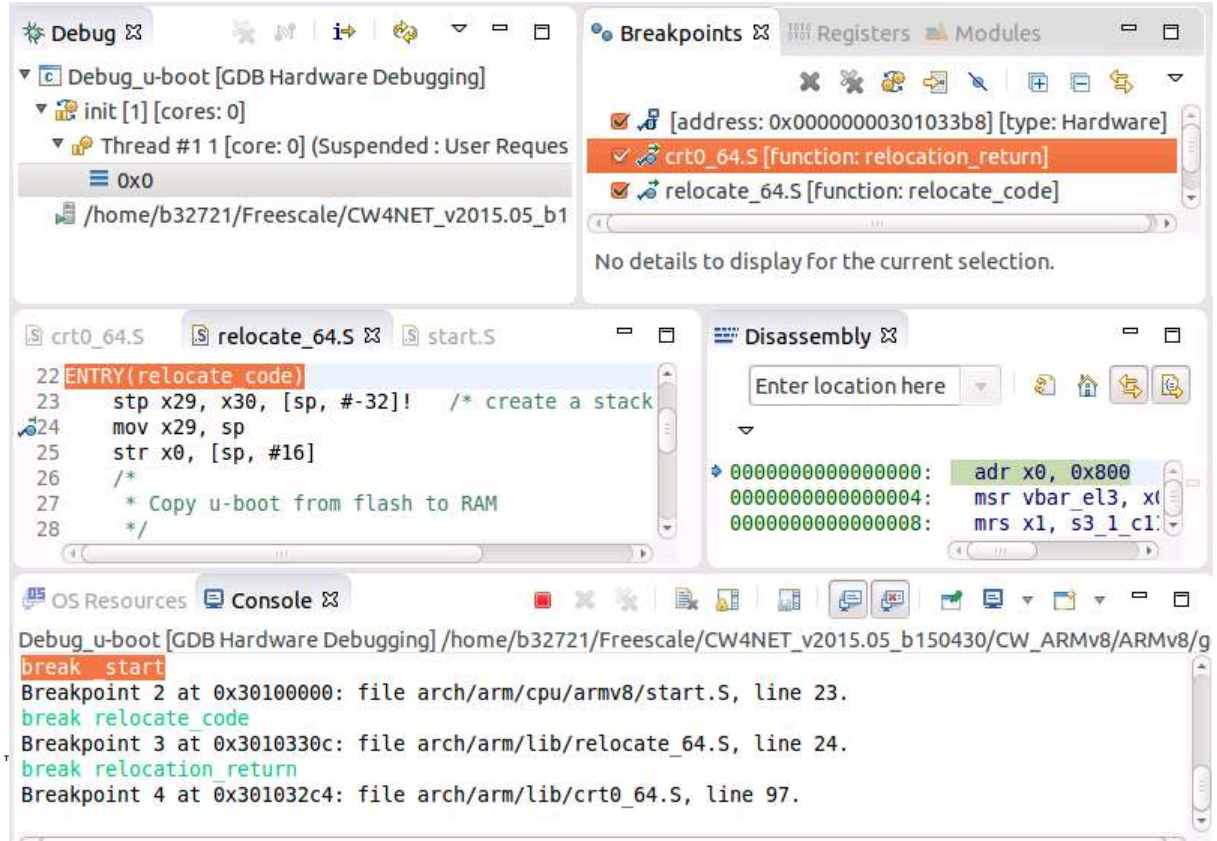
U-boot debug – Debug – Activity

- 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



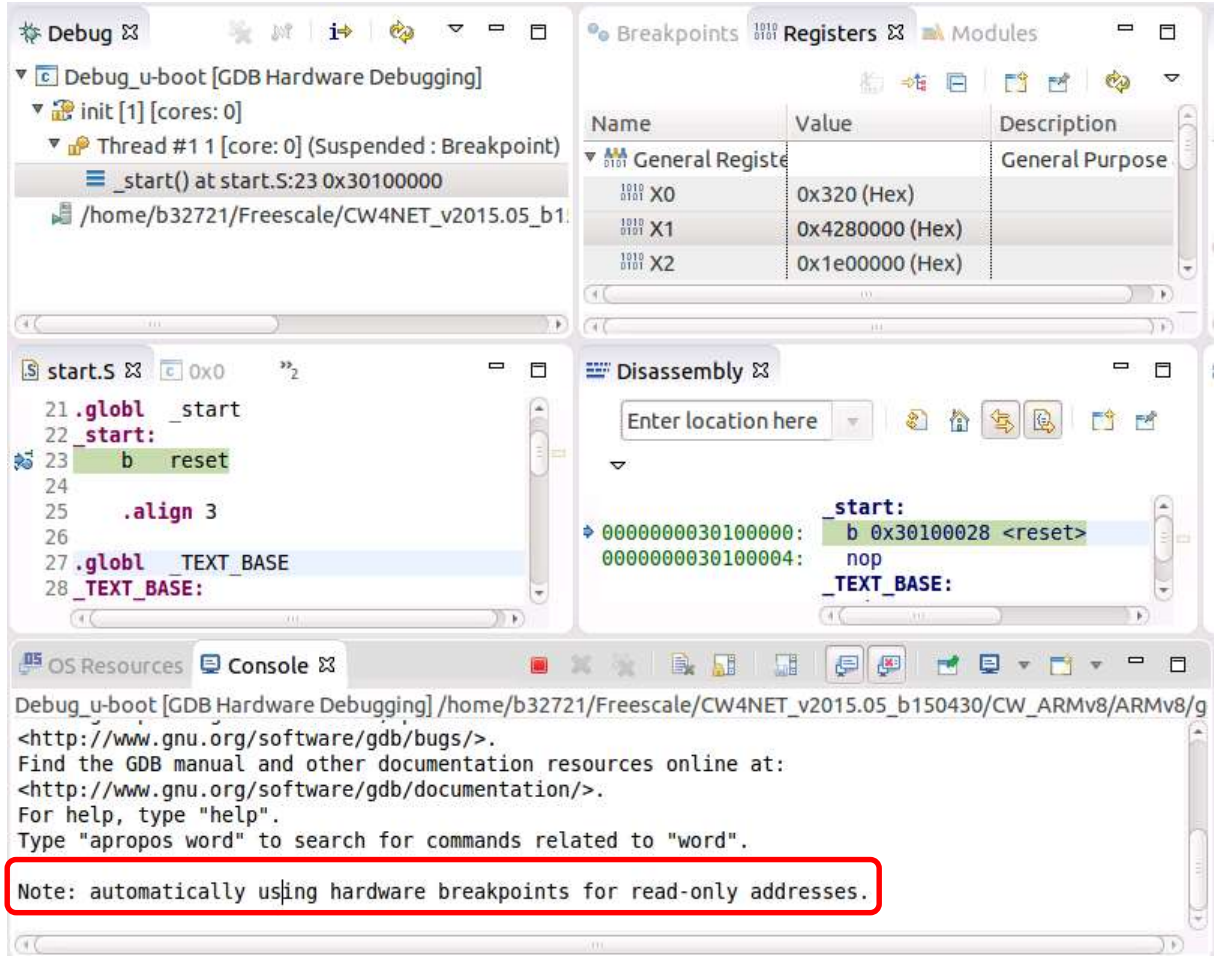
U-boot debug – Debug – Activity

- 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



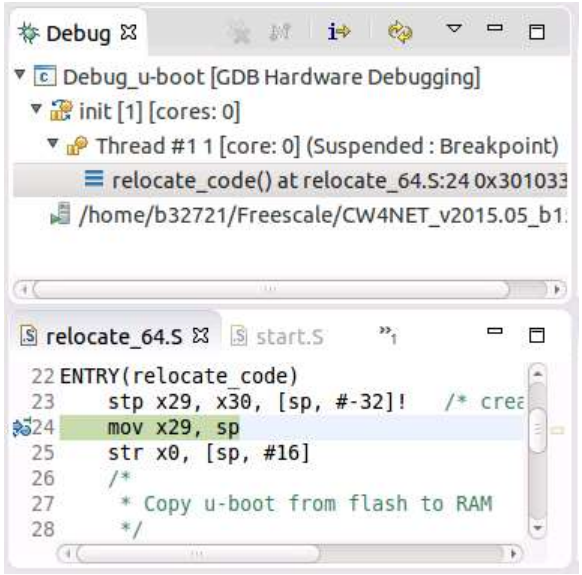
U-boot debug – Debug – Activity

- 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.



U-boot debug – Debug – Activity

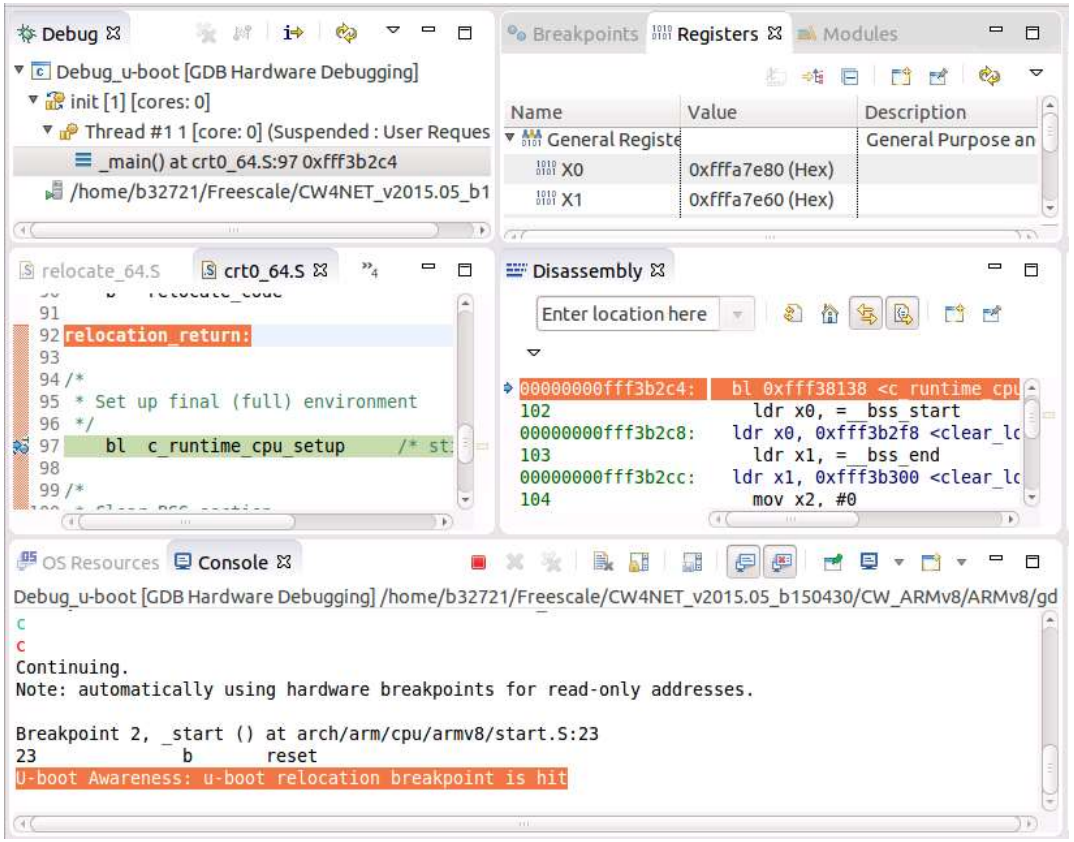
- Start relocation breakpoint hit
- U-boot running from Flash (address 0x3010xxxx) prepare for running from DDRAM
- Booting messages on u-boot console




```
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
```

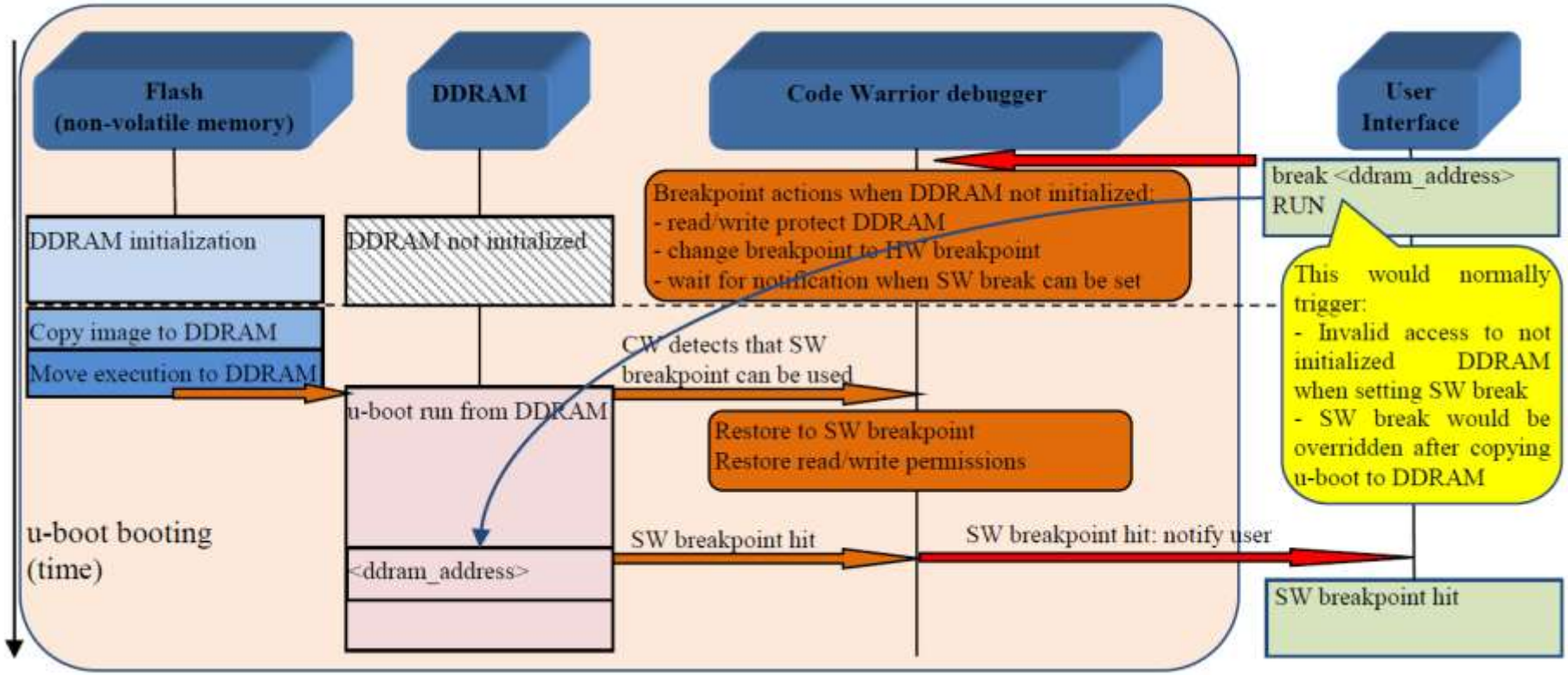
U-boot debug – Debug – Activity

- 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



U-boot debug – Debug – Activity

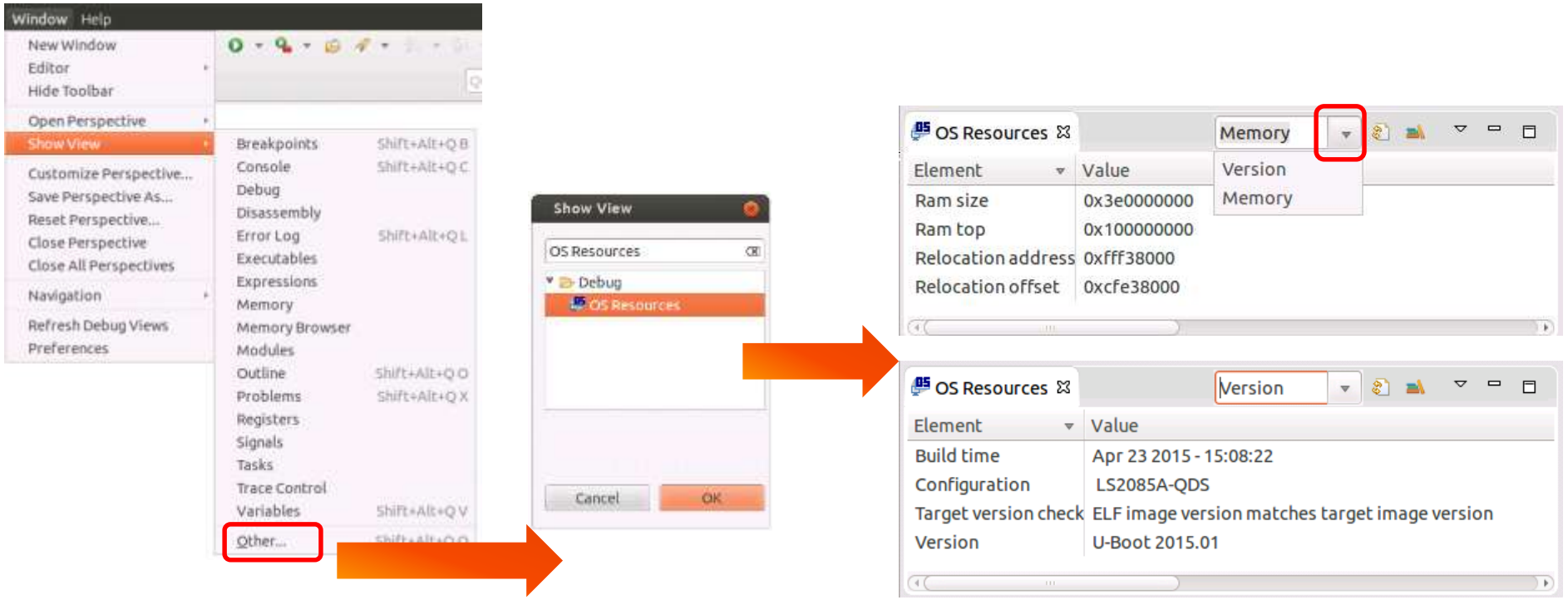
- U-boot Awareness allows setting SW breakpoints to DDRAM before DDRAM initialization and before u-boot relocation to DDRAM



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

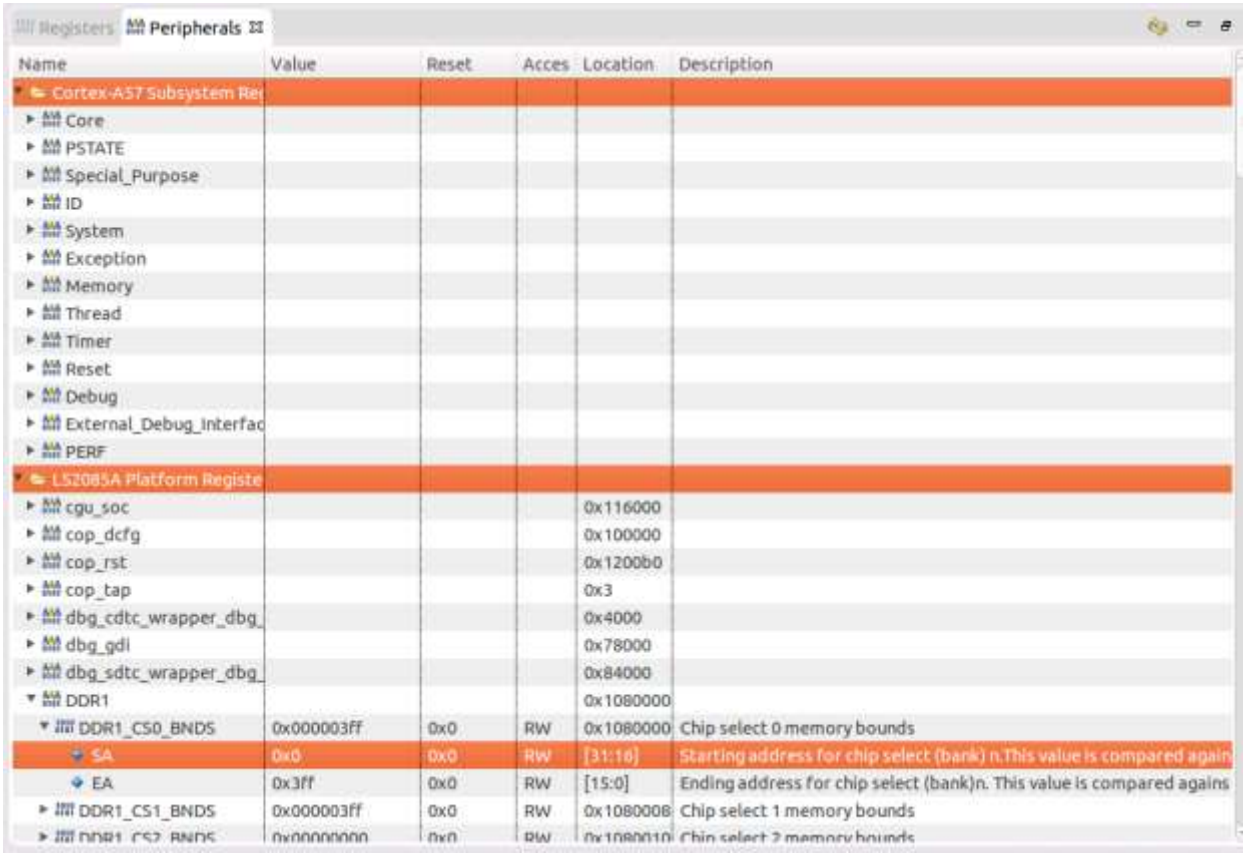


U-boot debug – Register / Peripheral – Activity

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

Registers | Peripherals

Name	Value
General Registers	
X0	0xfffa7e80 (Hex)
X1	0xfffa7e60 (Hex)
X2	0x40 (Hex)
X3	0x3f (Hex)
X4	0xfff9c100 (Hex)
X5	0xe7ff069f (Hex)
X6	0xffffffd0 (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)

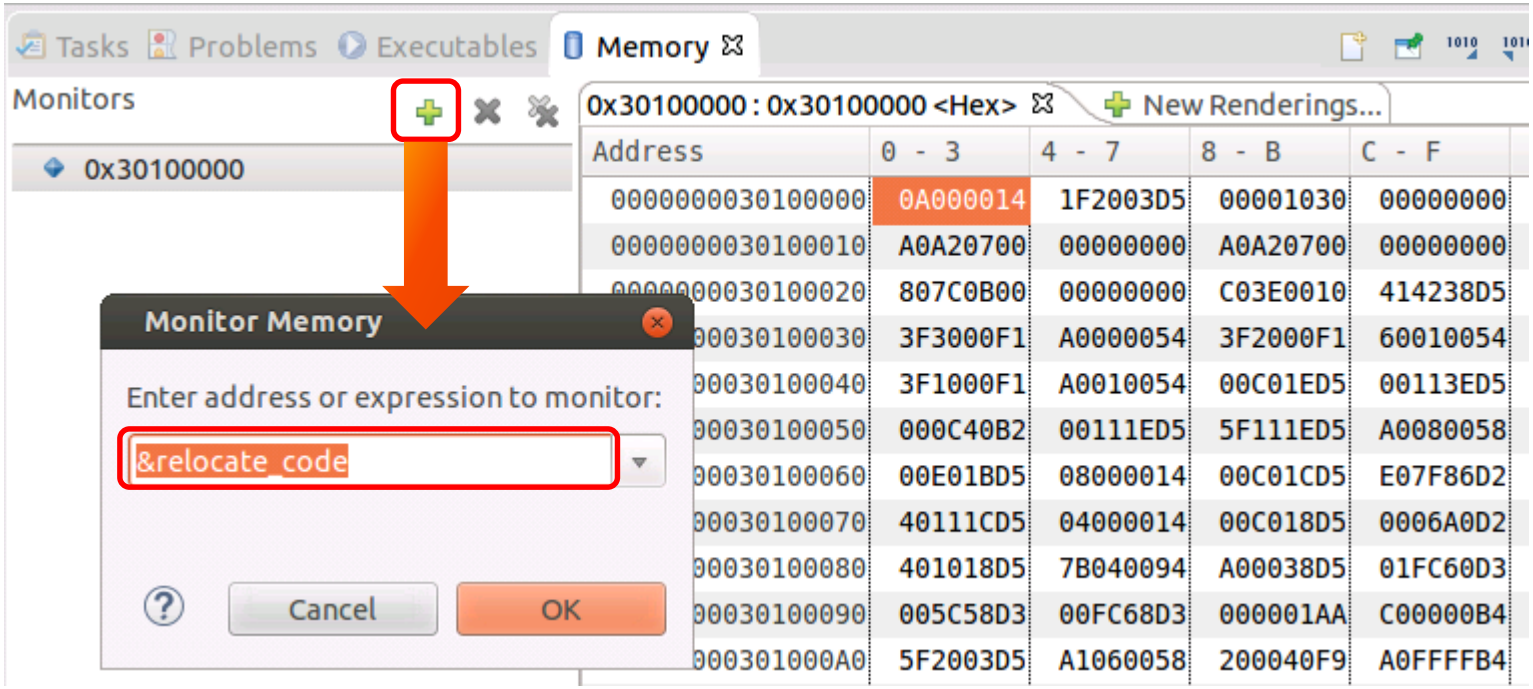


Registers | Peripherals

Name	Value	Reset	Acces	Location	Description
Cortex-A57 Subsystem Re					
Core					
PSTATE					
Special_Purpose					
ID					
System					
Exception					
Memory					
Thread					
Timer					
Reset					
Debug					
External_Debug_Interfac					
PERF					
LS2085A Platform Registe					
cgu_soc				0x116000	
cop_dcfg				0x100000	
cop_rst				0x1200b0	
cop_tap				0x3	
dbg_cdtc_wrapper_dbg_				0x4000	
dbg_gdi				0x78000	
dbg_sdtc_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 again
EA	0x3ff	0x0	RW	[15:0]	Ending address for chip select (bank)n. This value is compared agains
DDR1_CS1_BNDS	0x000003ff	0x0	RW	0x1080008	Chip select 1 memory bounds
DDR1_CS2_BNDS	0x00000000	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



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

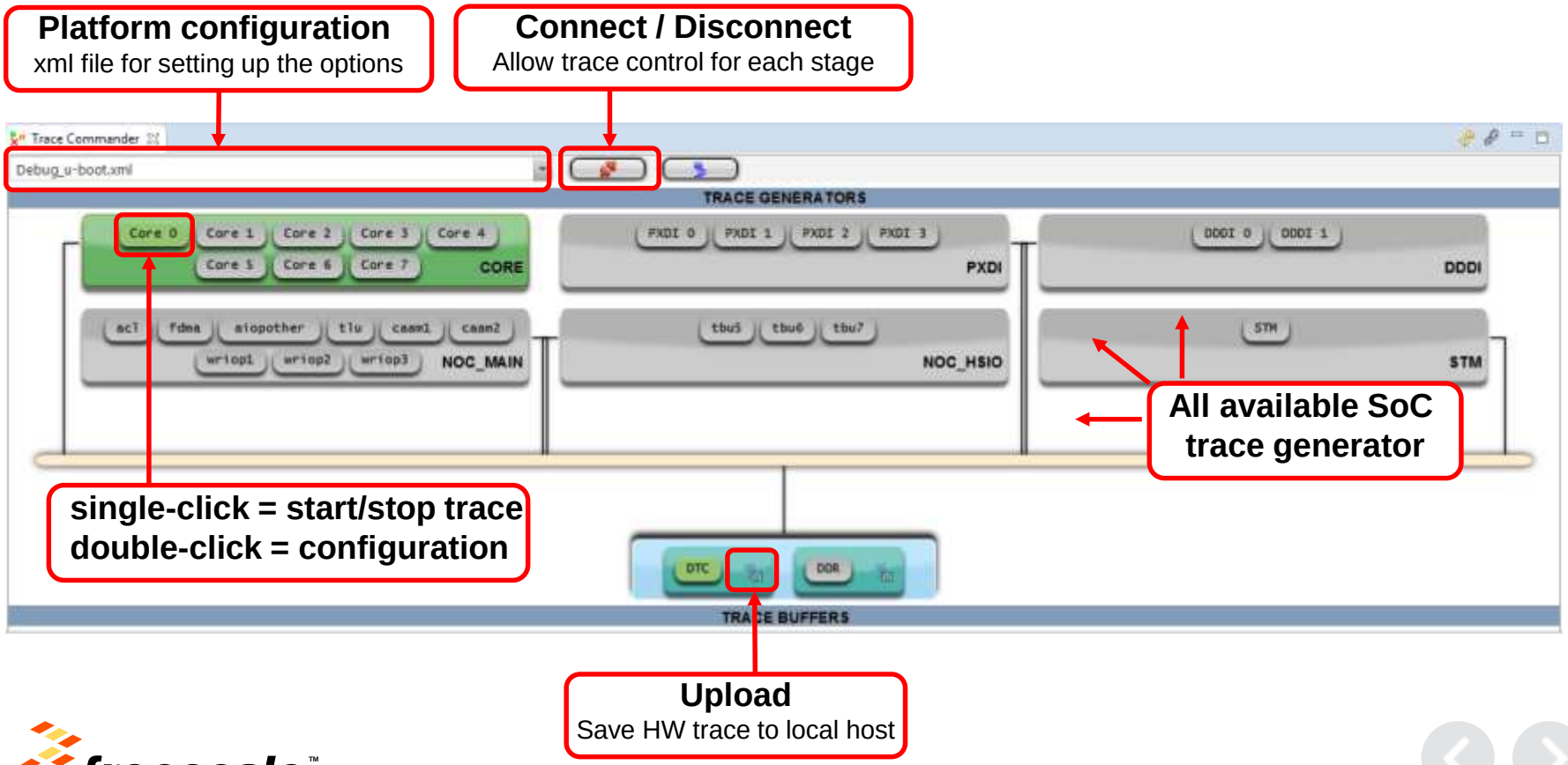
CodeWarrior TAP over ETH/USB + Serial

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

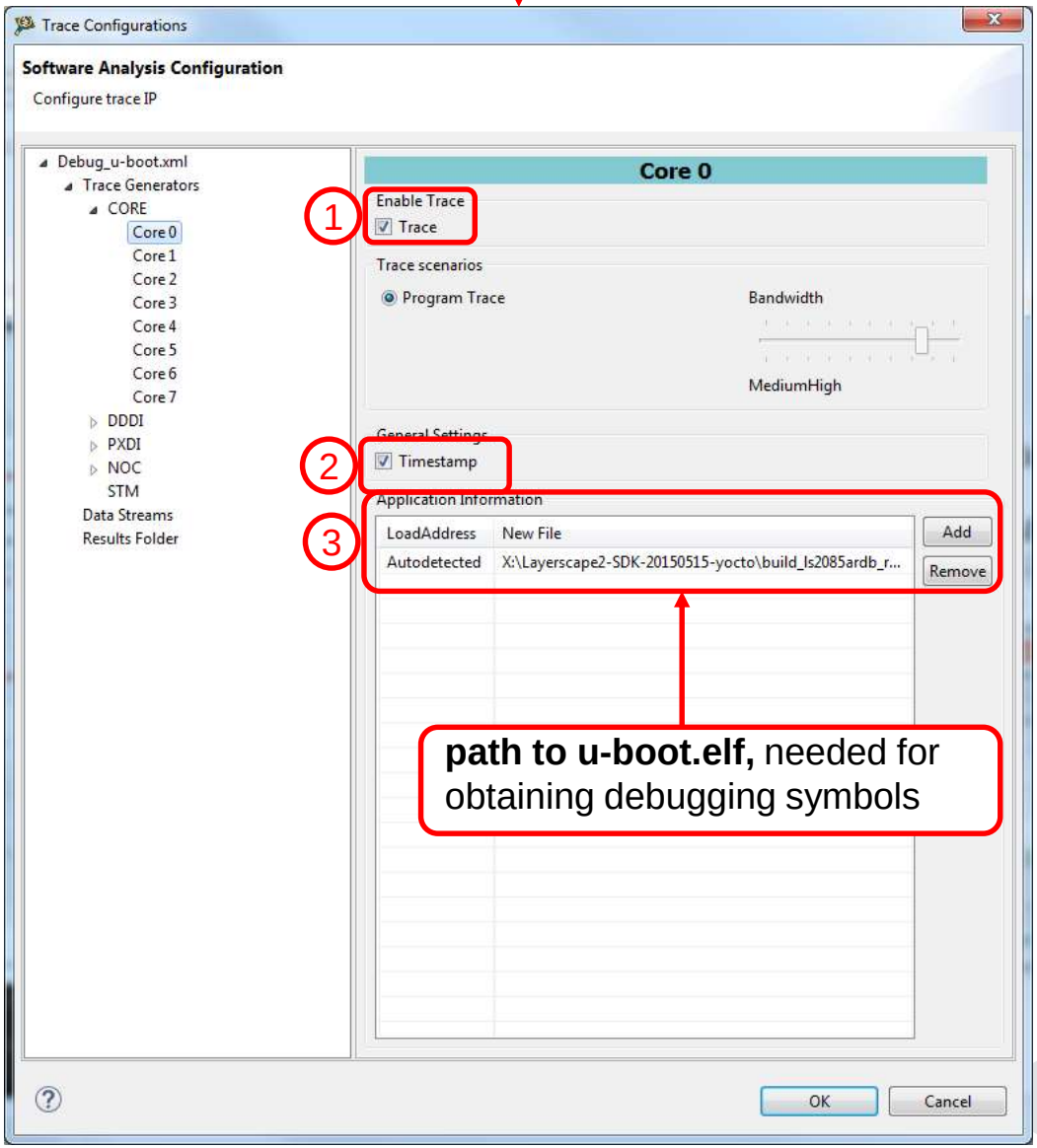
u-boot tracing: Software Analysis Overview

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



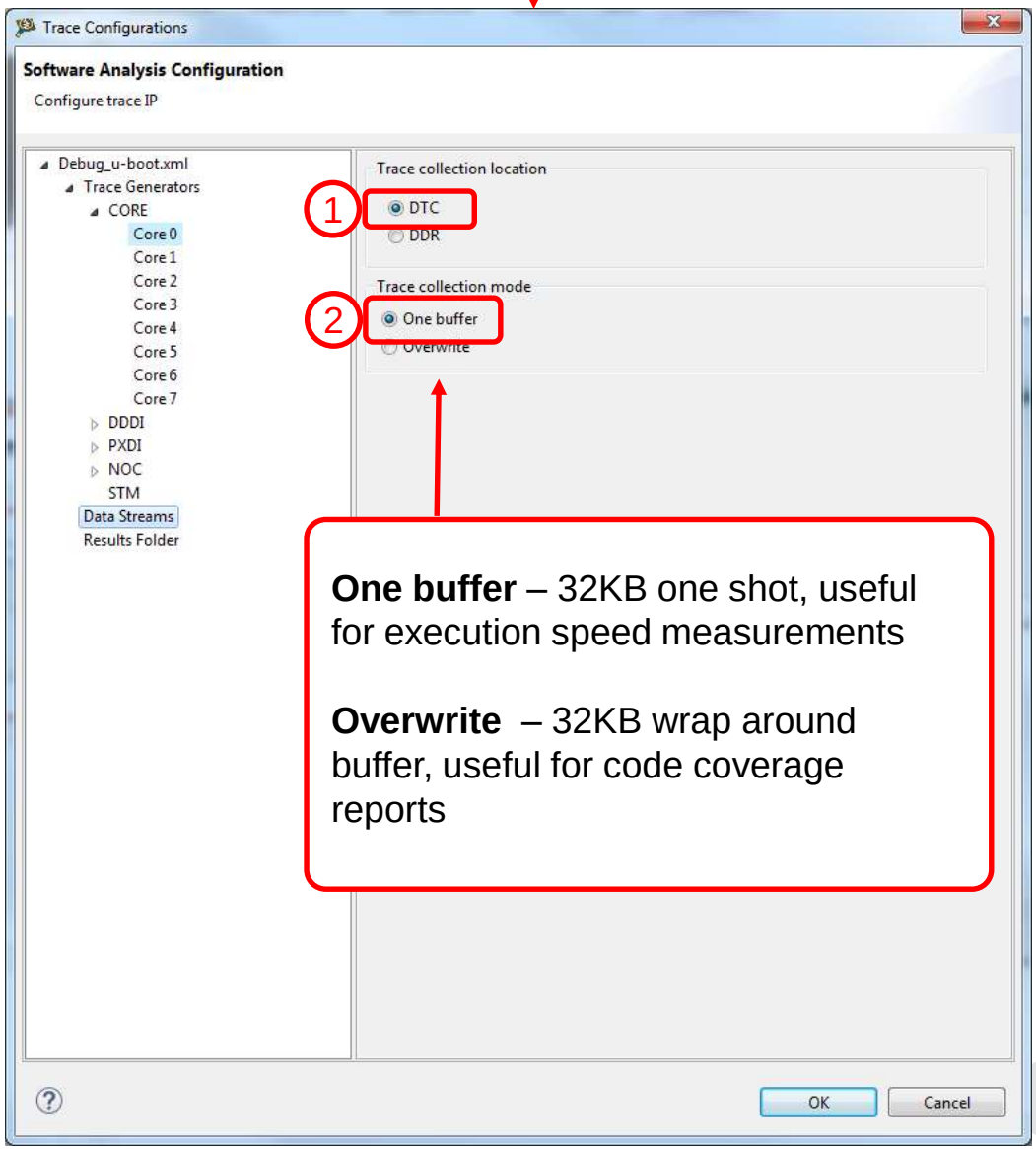
u-boot tracing: Software setup (cont'd)

double-click to configure trace



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

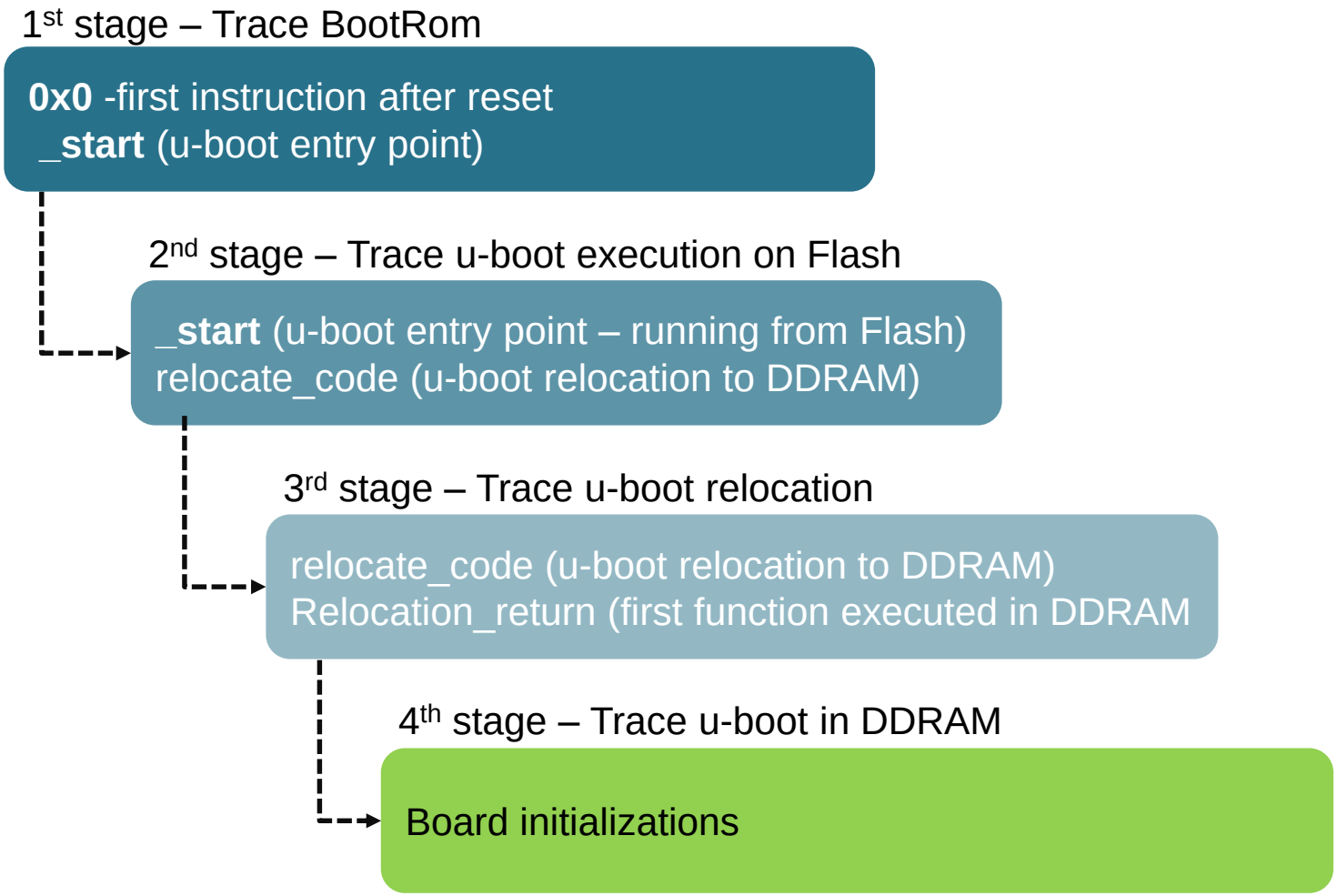
Select the destination for PTM stream and operation mode



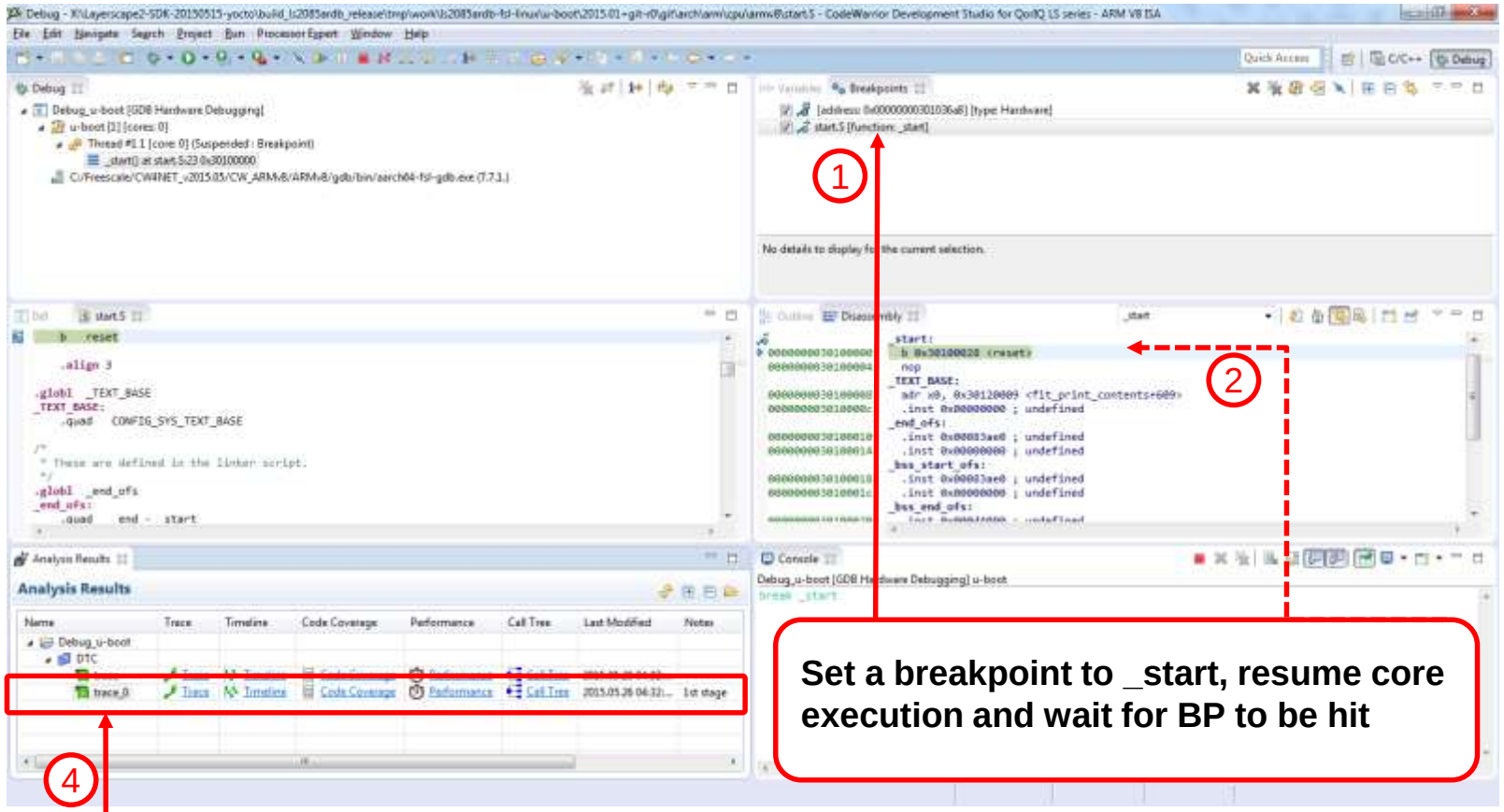
One buffer – 32KB one shot, useful for execution speed measurements

Overwrite – 32KB wrap around buffer, useful for code coverage reports

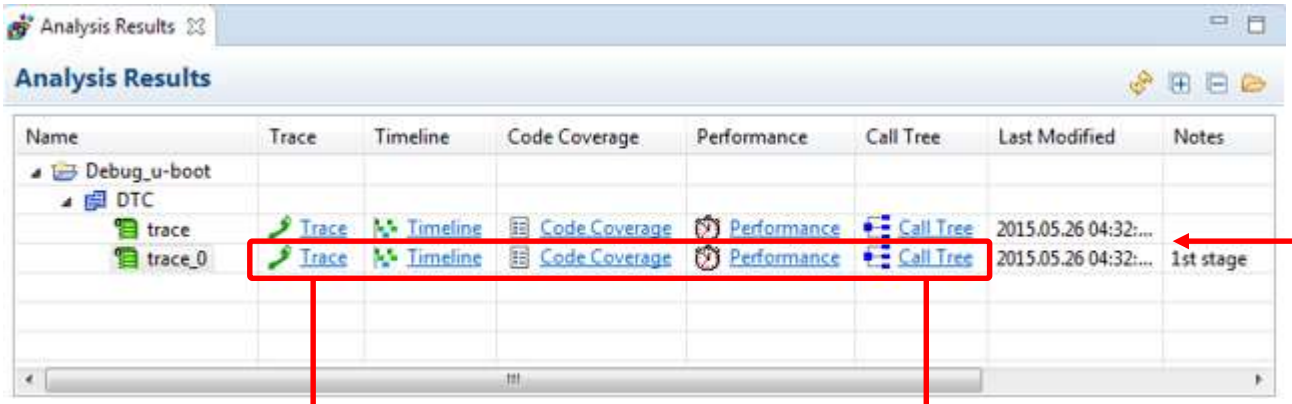
u-boot tracing: actions per stage



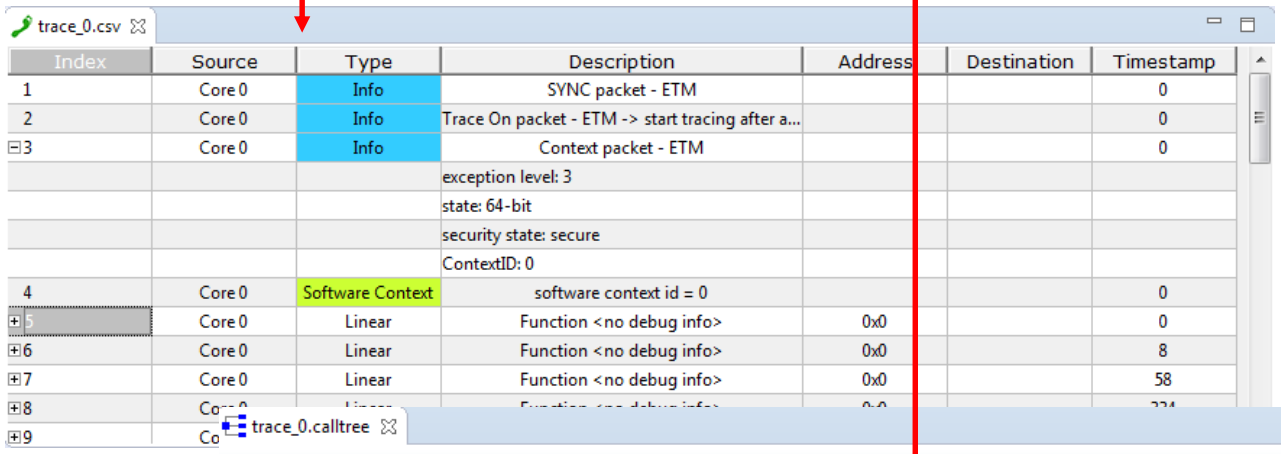
u-boot tracing: 1st stage setup



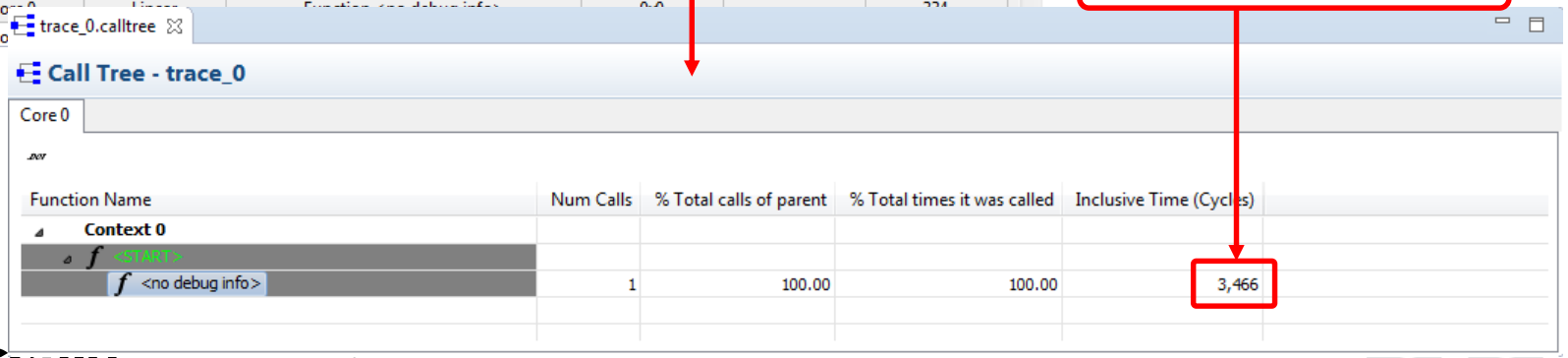
u-boot tracing: 1st stage results



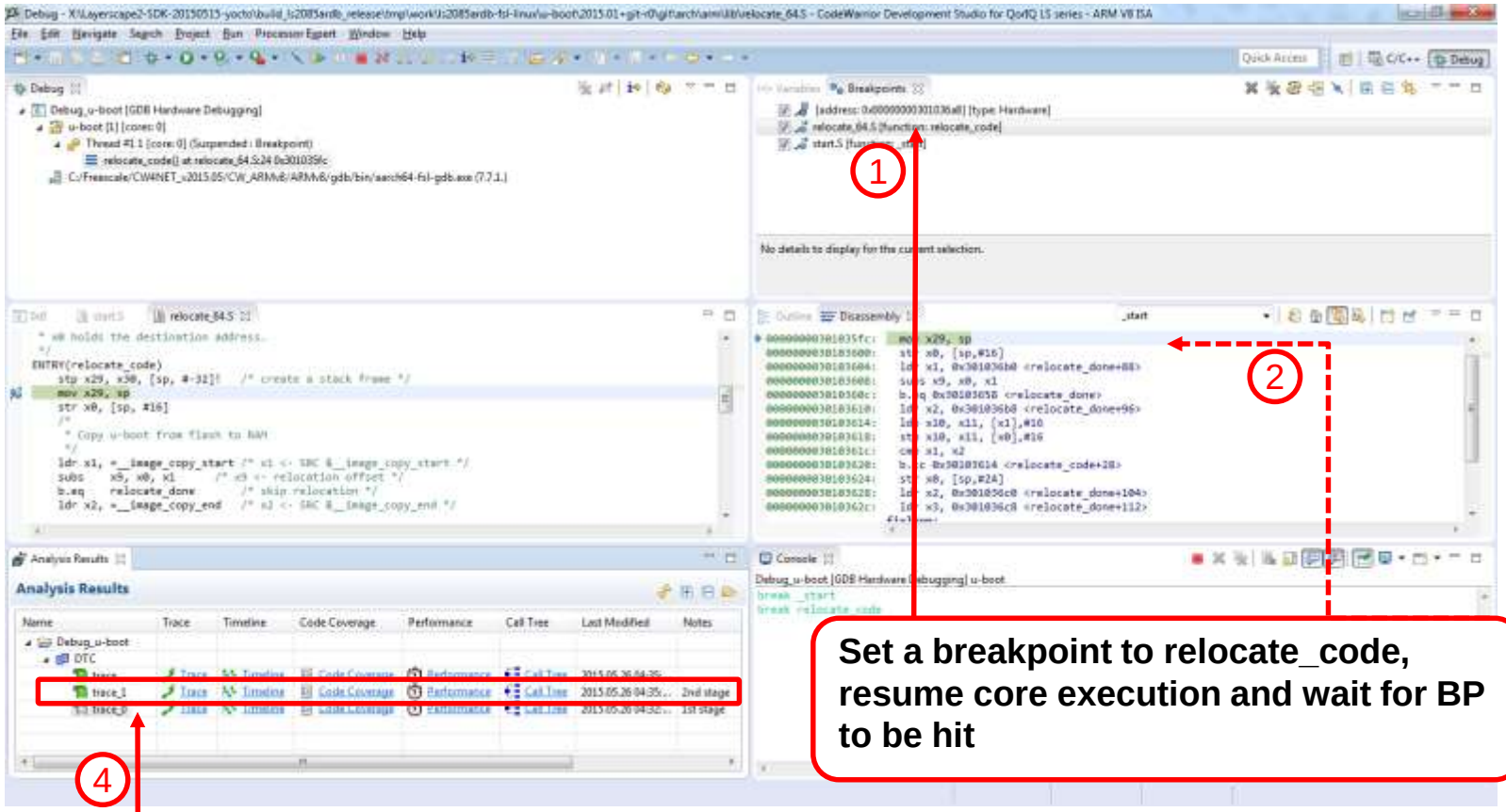
Quick links to results. Note! Not all of them are relevant at this point



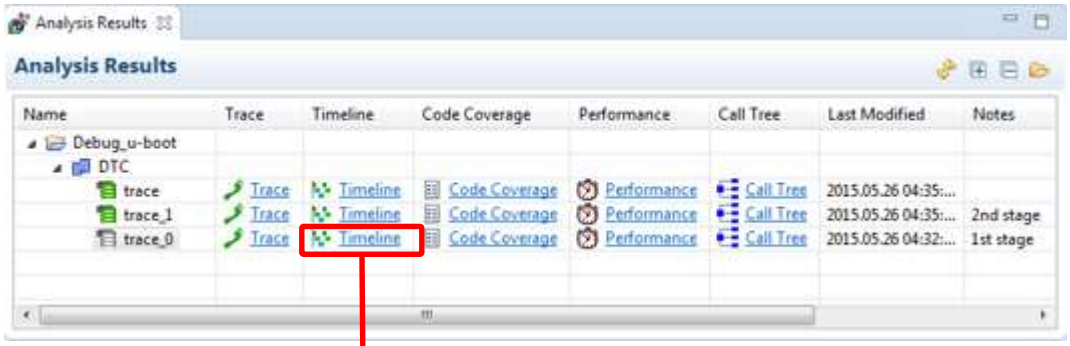
BootRom execution time



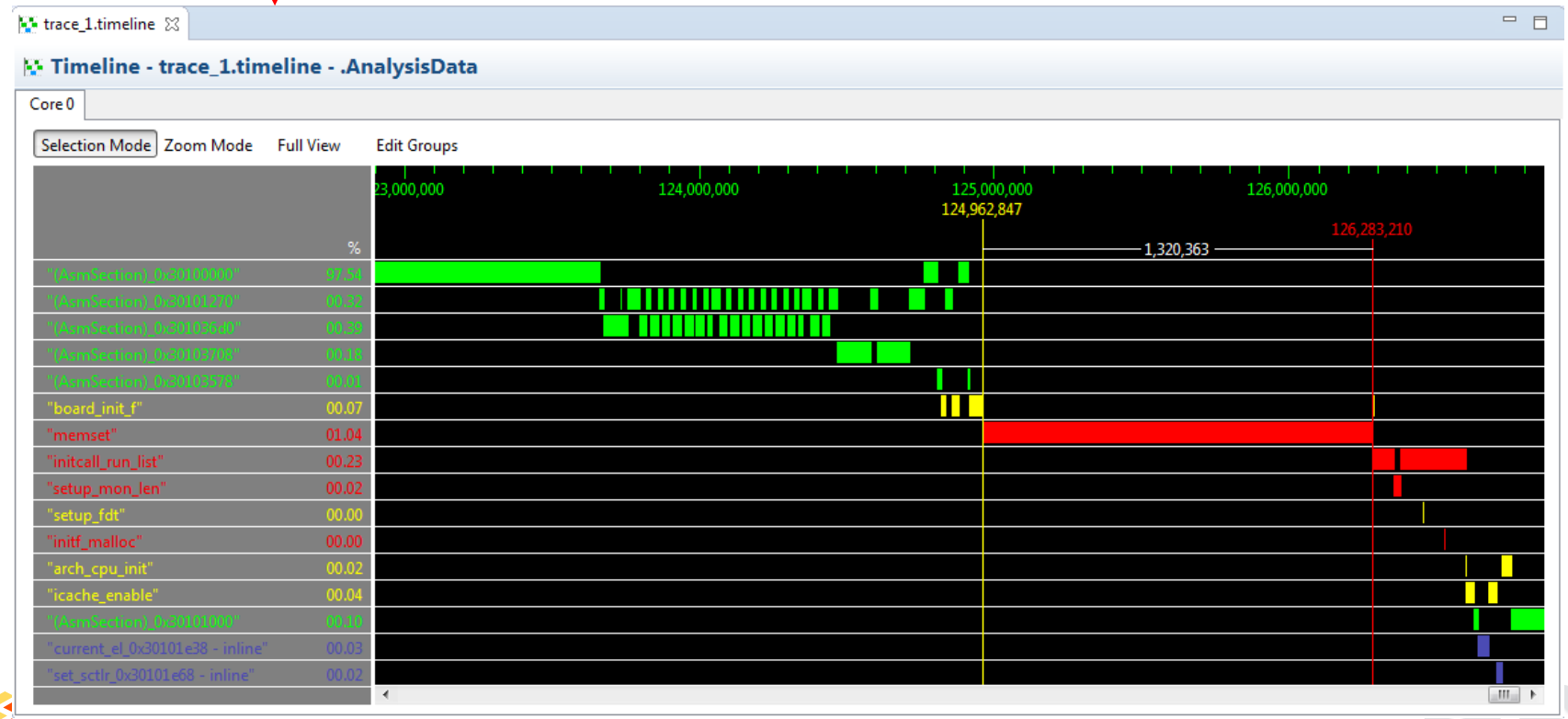
u-boot tracing: 2nd stage setup



u-boot tracing: 2nd stage results

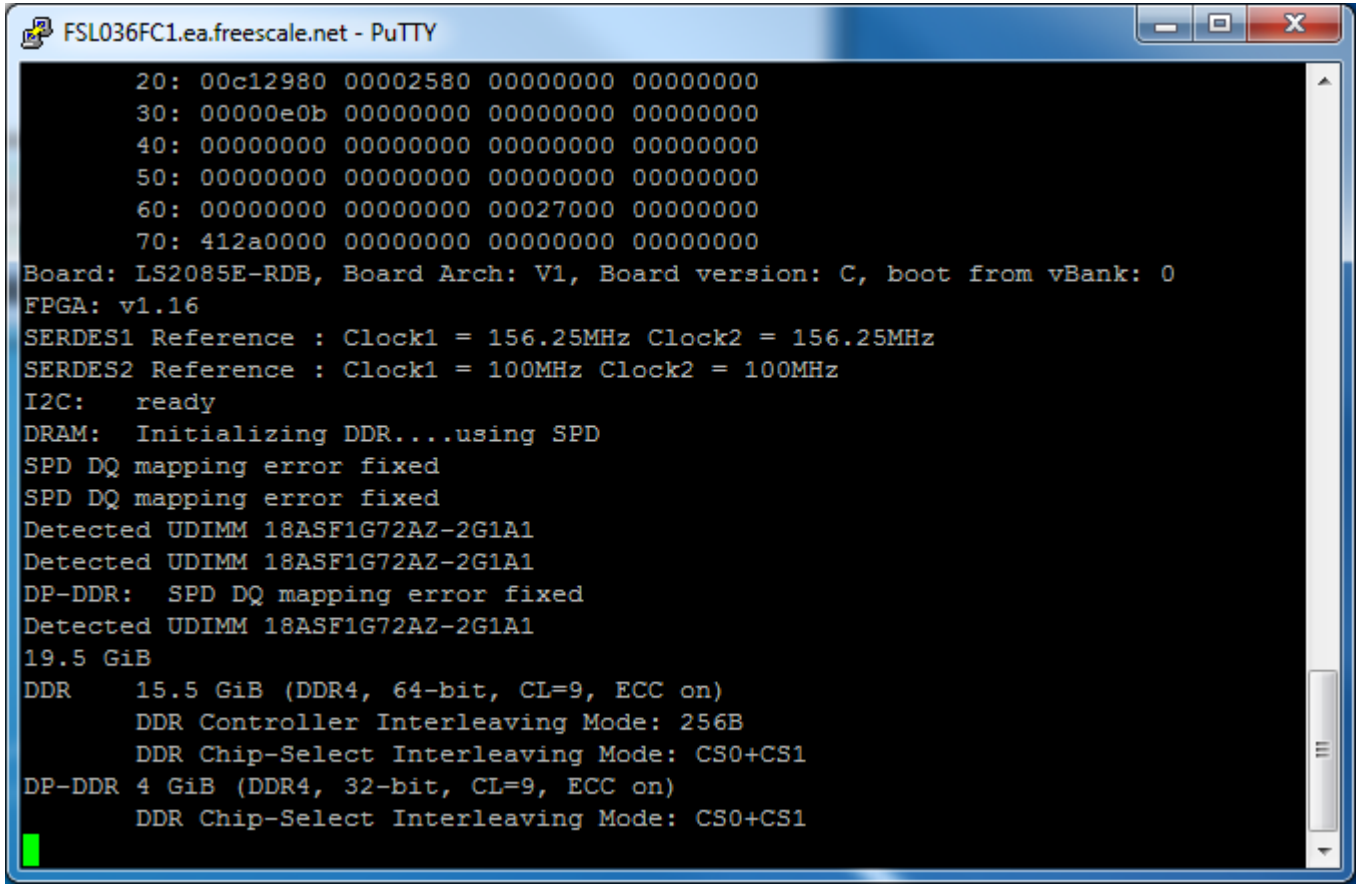


Sequential view of u-boot routines:
- Easy highlighting of special routines, cycle count measurements

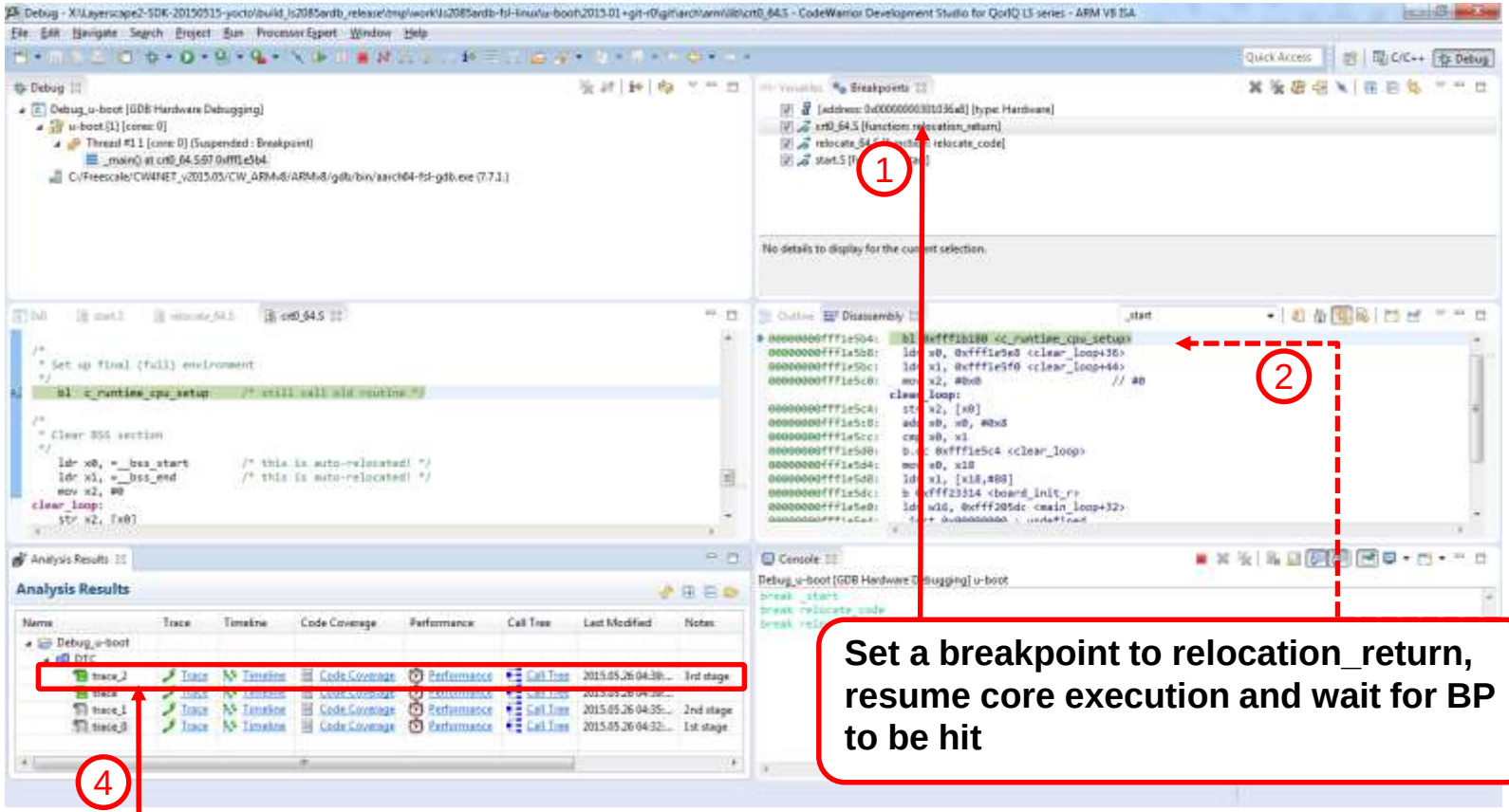


u-boot tracing: 2nd stage results (cont'd)

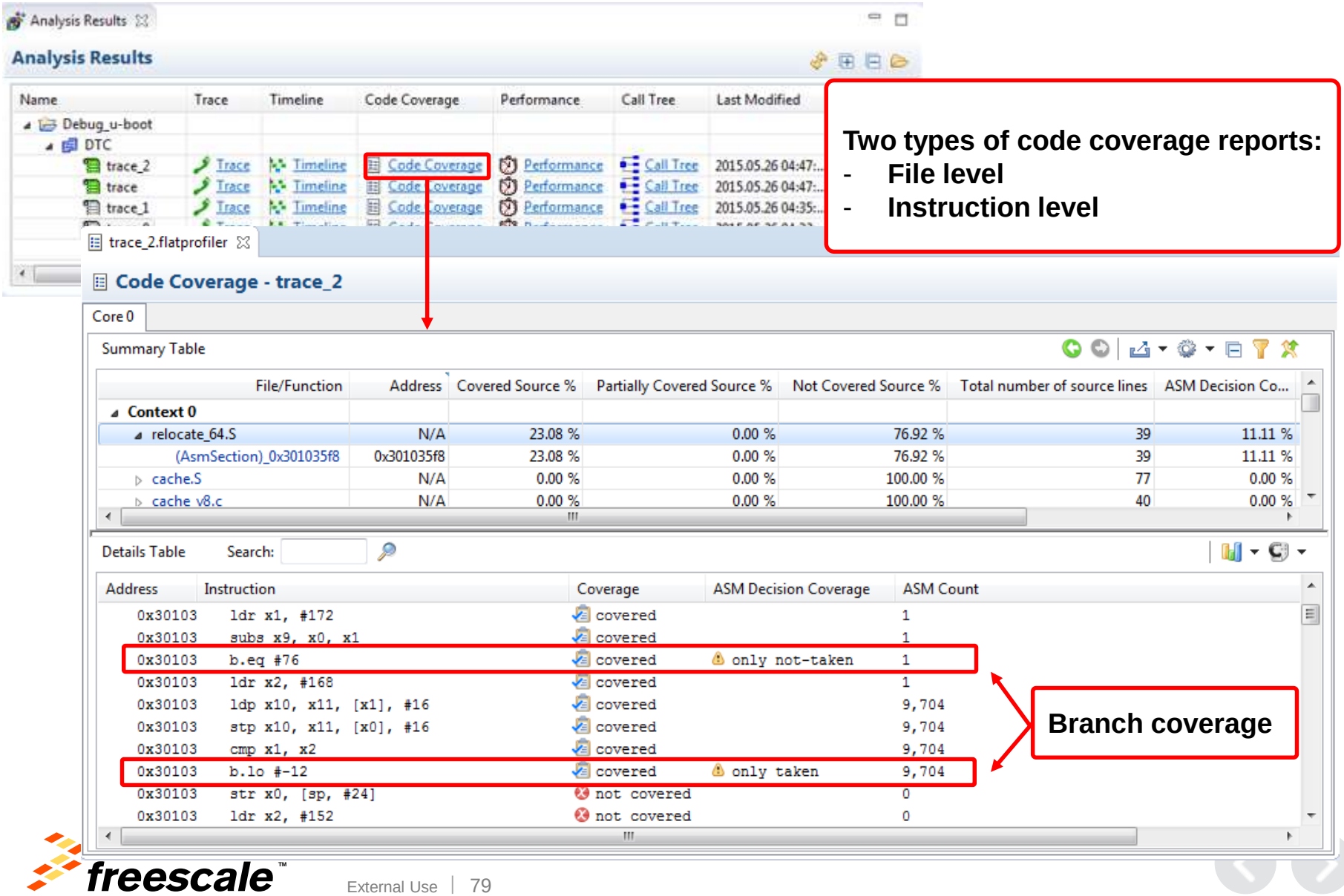
During 2nd stage the u-boot should display booting messages on console



u-boot tracing: 3rd stage setup

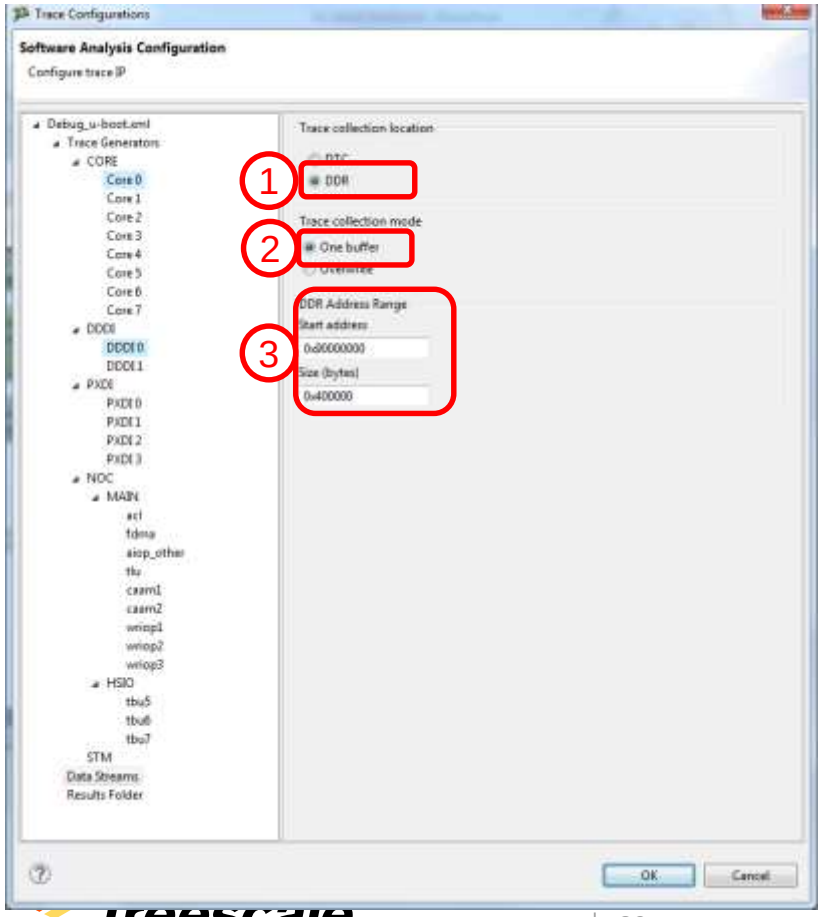


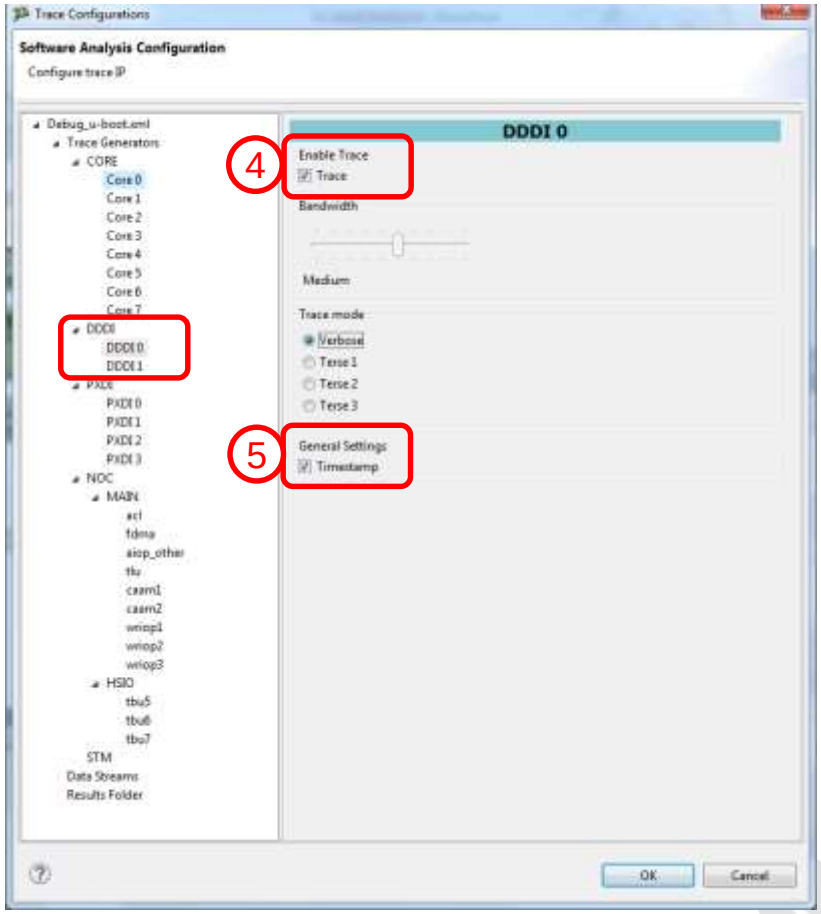
u-boot tracing: 3rd stage results



u-boot tracing: 4th stage setup

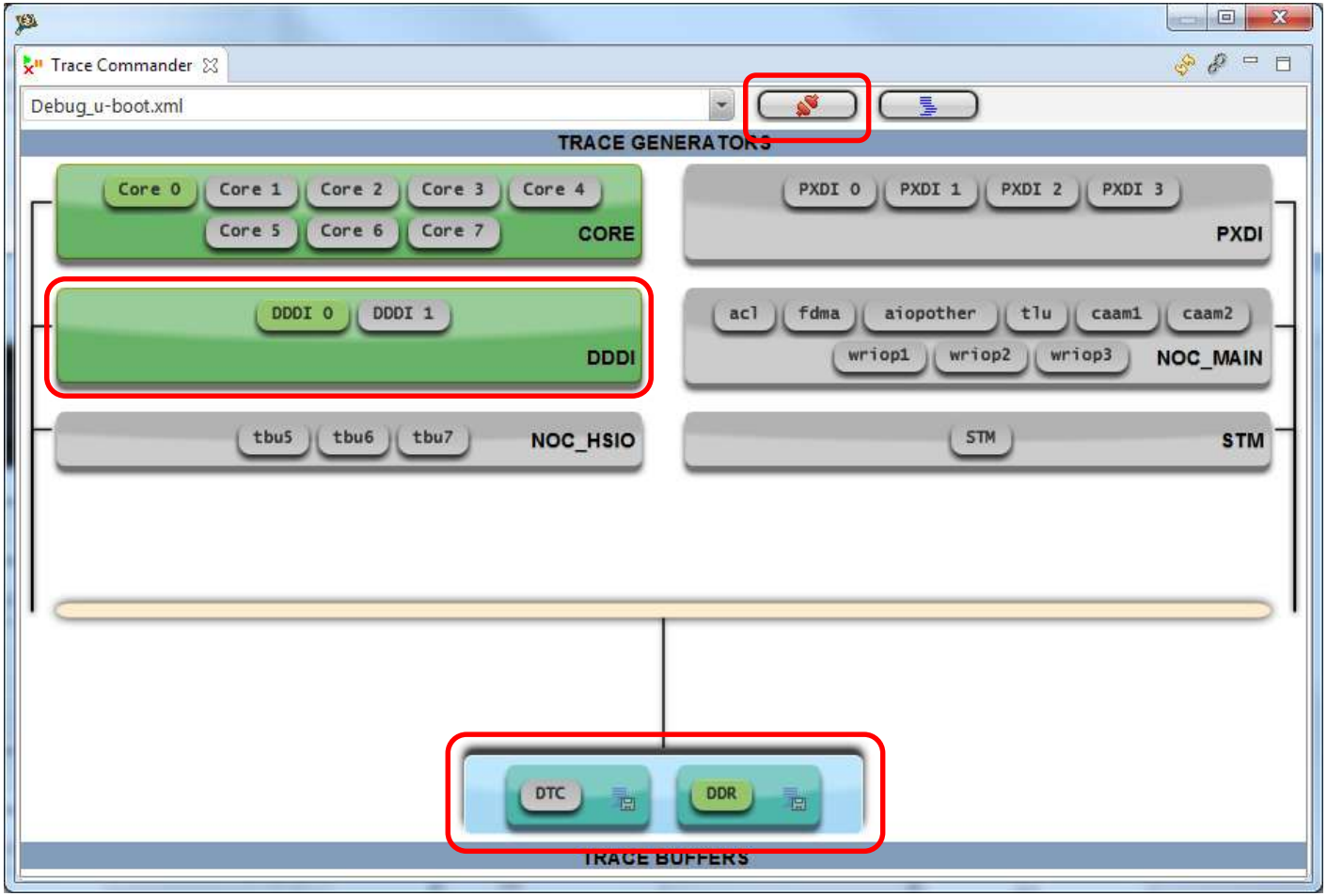
- At this point the DDR is initialized. Move Trace Buffer into DDR in order to remove 32KB size limitation;
- DDR trace event can be captured too



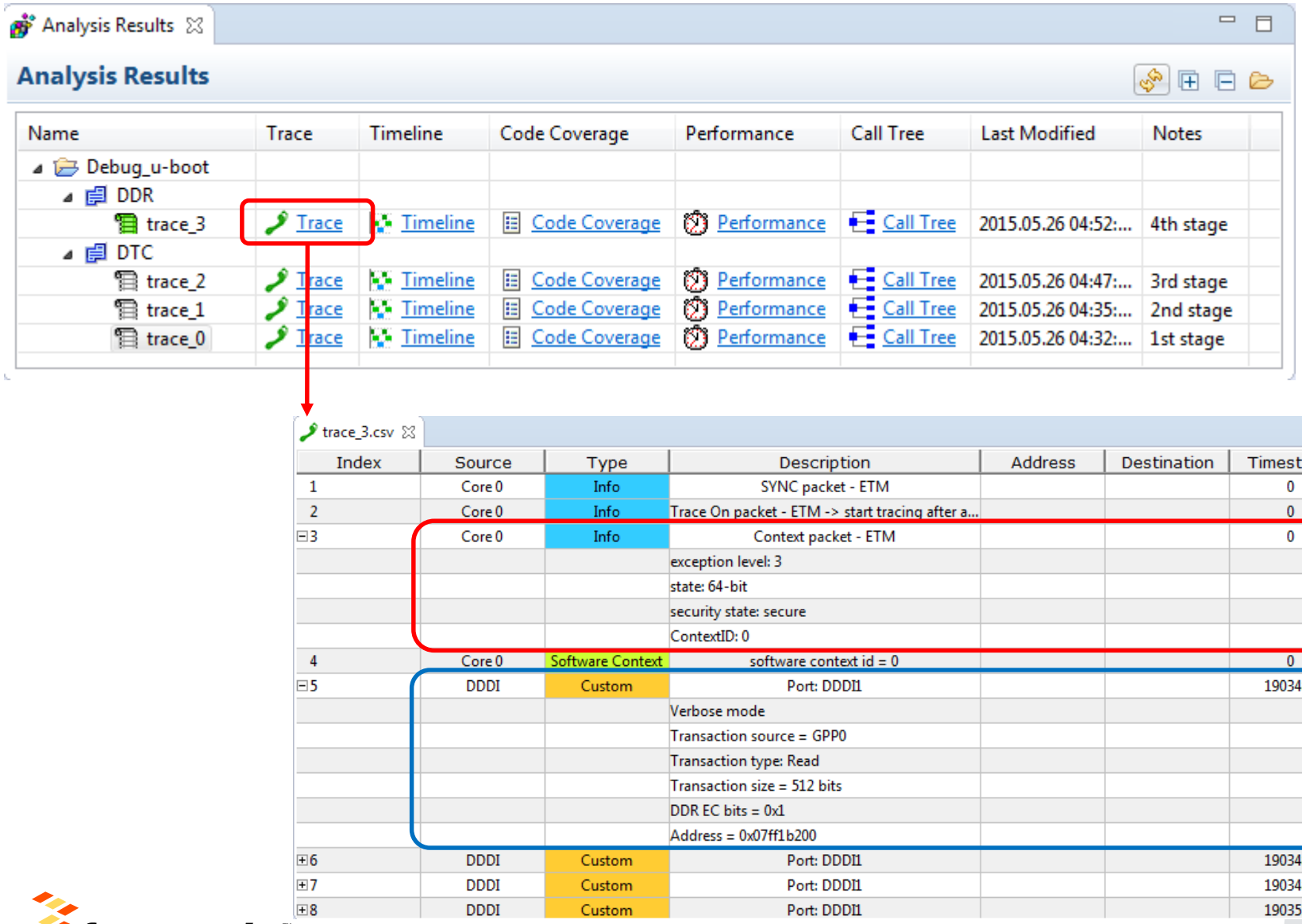


u-boot tracing: 4th stage setup (cont'd)

- Re-apply target settings



u-boot tracing: 4th stage results

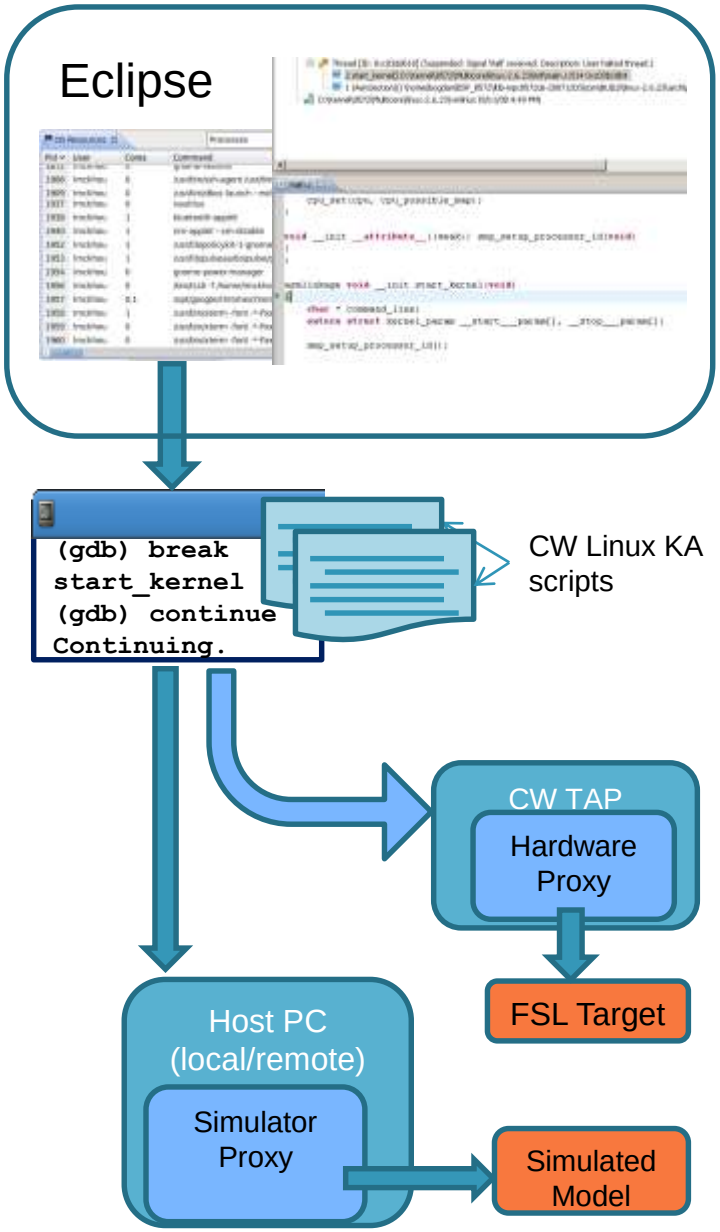


Linux Kernel Debug

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

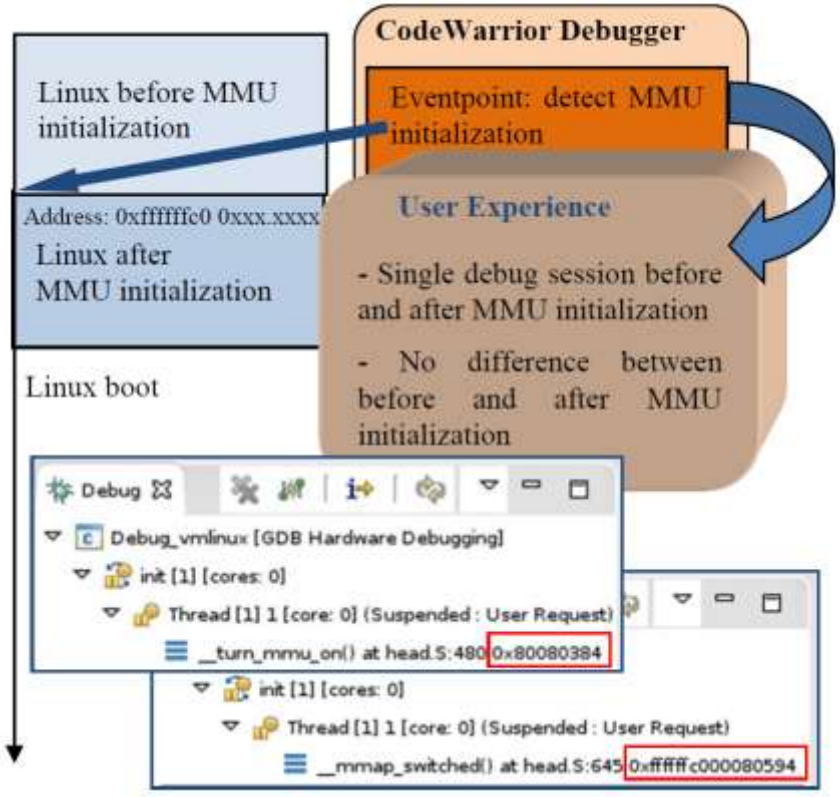


Linux kernel debug – Capabilities

- **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

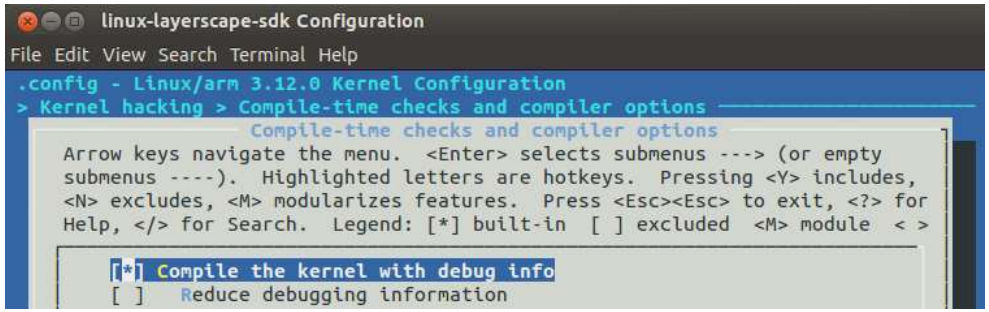
Linux kernel debug – Capabilities

- **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**



  - Build image
    - bitbake fsl-image-kernelitb

Linux kernel debug – Prerequisites –Activity (cont)

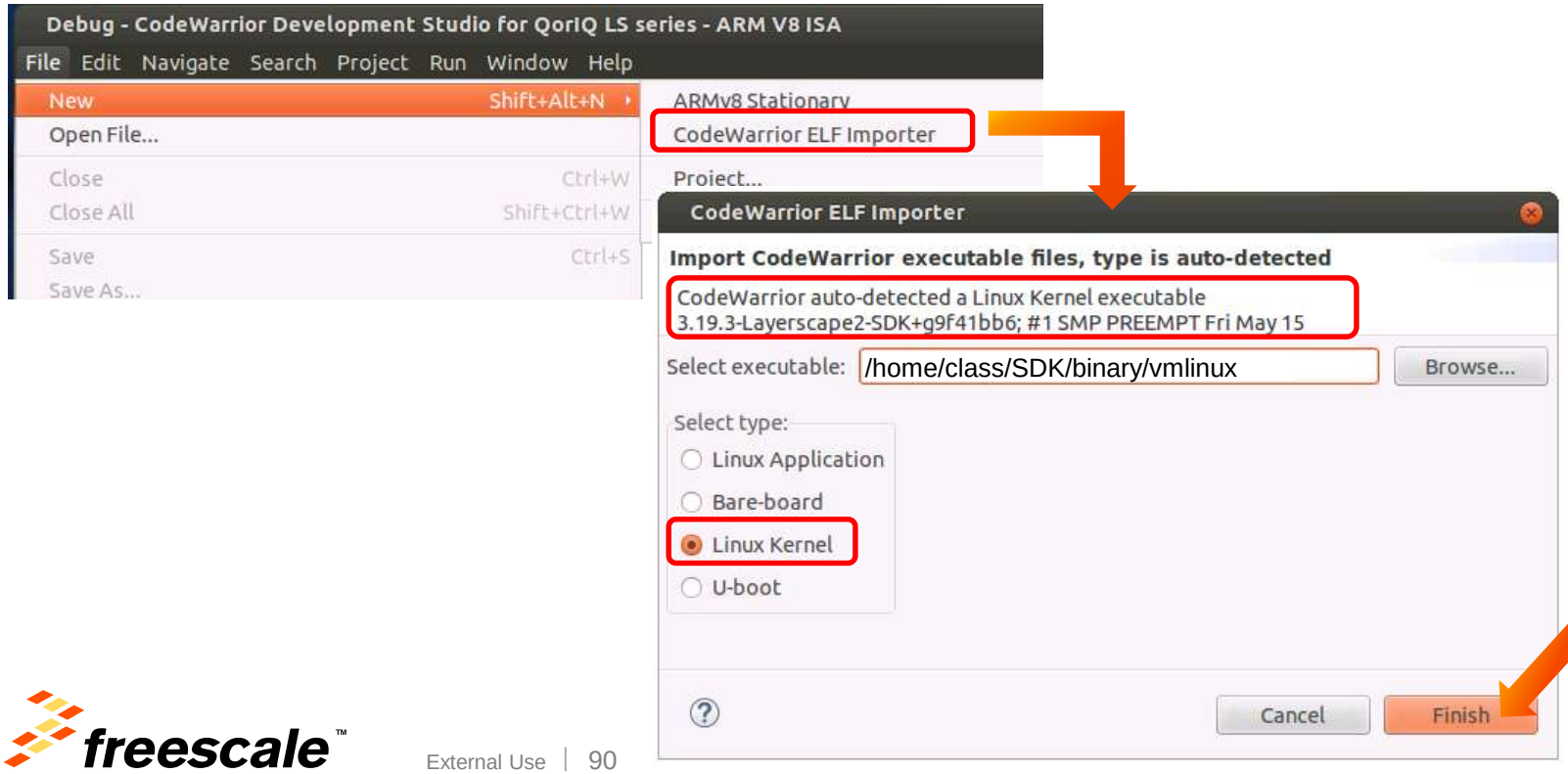
- **LS2085A SDK**
  - Build Linux kernel
    - **bitbake virtual/kernel**
  - Get the images (paths relative to <YoctoInstallationPath>)
    - **u-boot**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/u-boot-ls2085aqds_config.bin
    - **uImage**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/uImage-LS2085ARDB.bin
    - **DTB (device tree)**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/uImage-LS2085A-RDB.dtb
    - **Ramdisk**: build_ls2085ardb_release/tmp/deploy/images/ls2085ardb/fsl-image-core-LS2085ARDB.ext2.gz.u-boot
    - **vmlinux (in sync with uImage):** 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

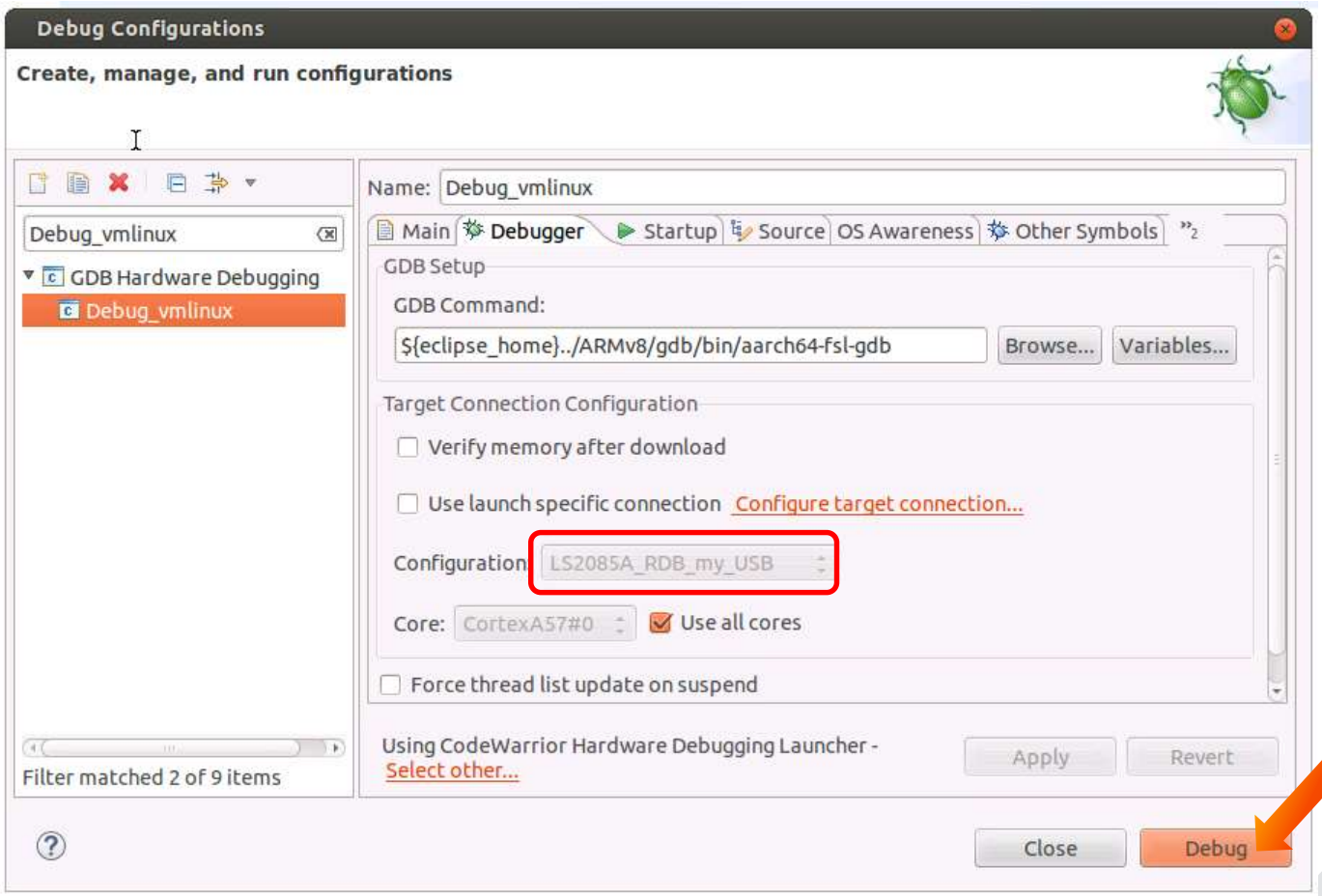
Linux kernel debug – Create project – Activity

- 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



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‘

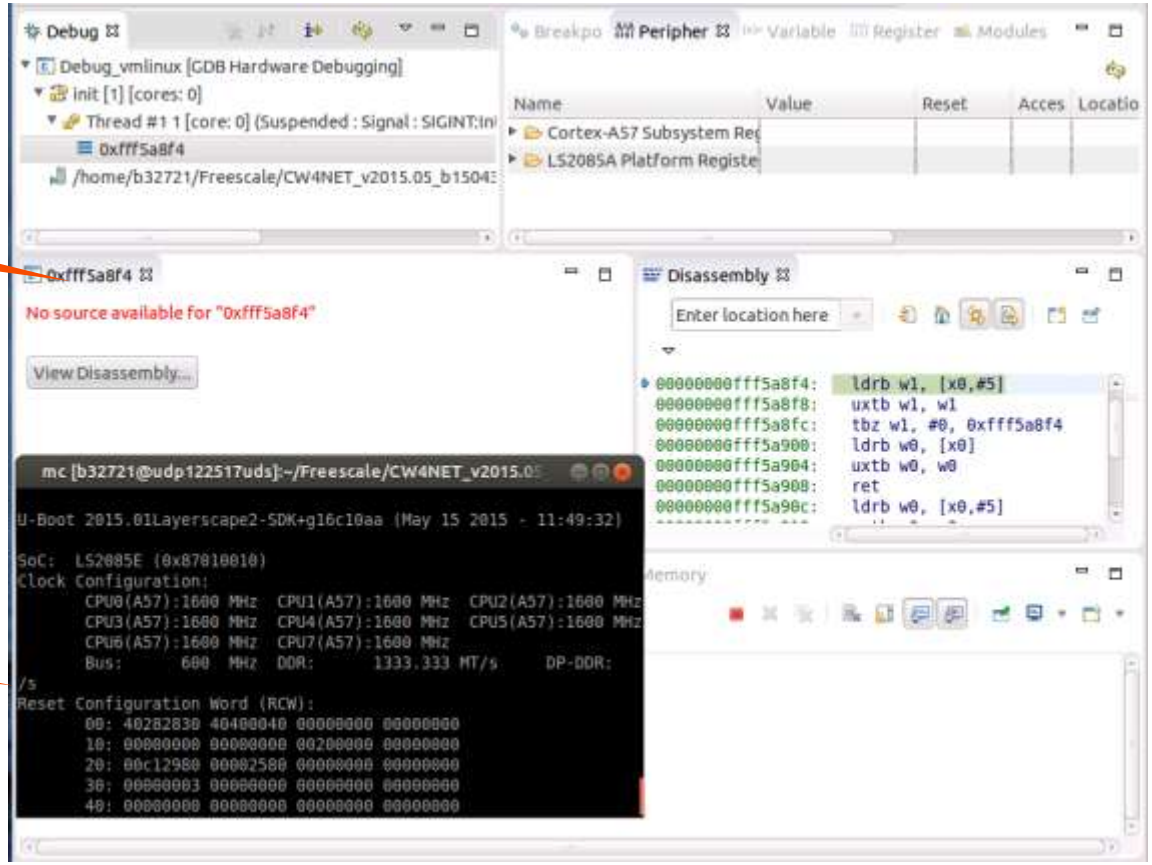
- To **boot Linux:**
  - **boot –** boot Linux from flash
  - **run bootLinux –** boot Linux from TFTP

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**

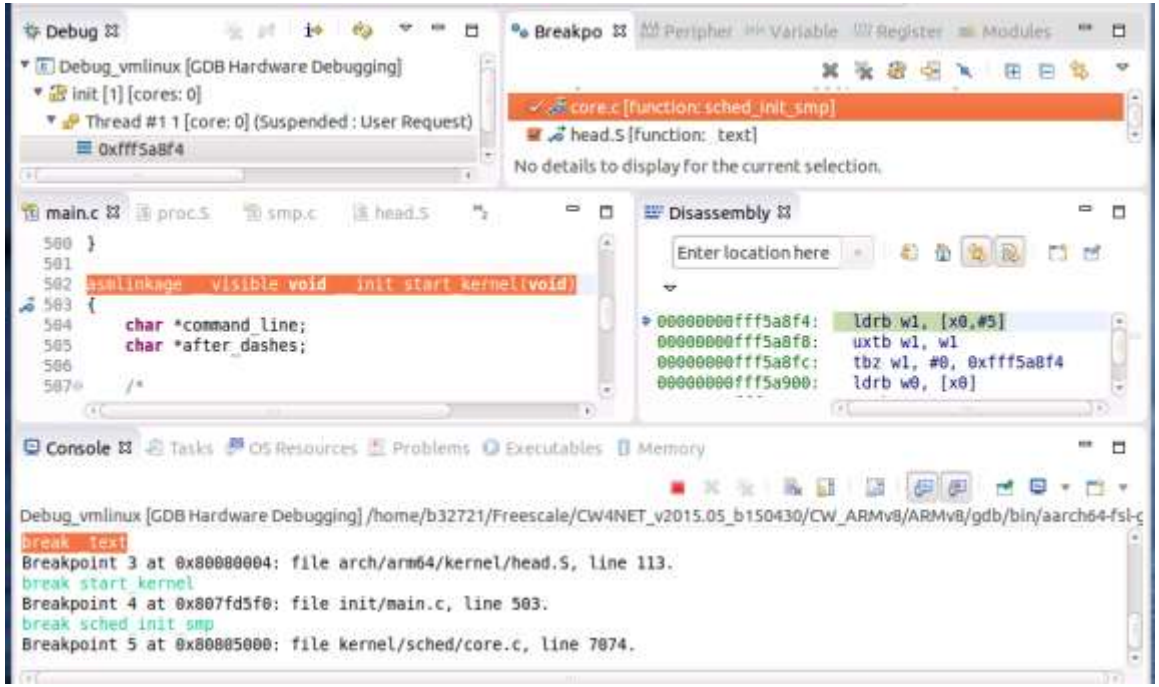
No symbols available as while being in u-boot, the Linux has not been started yet

U-boot console



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)



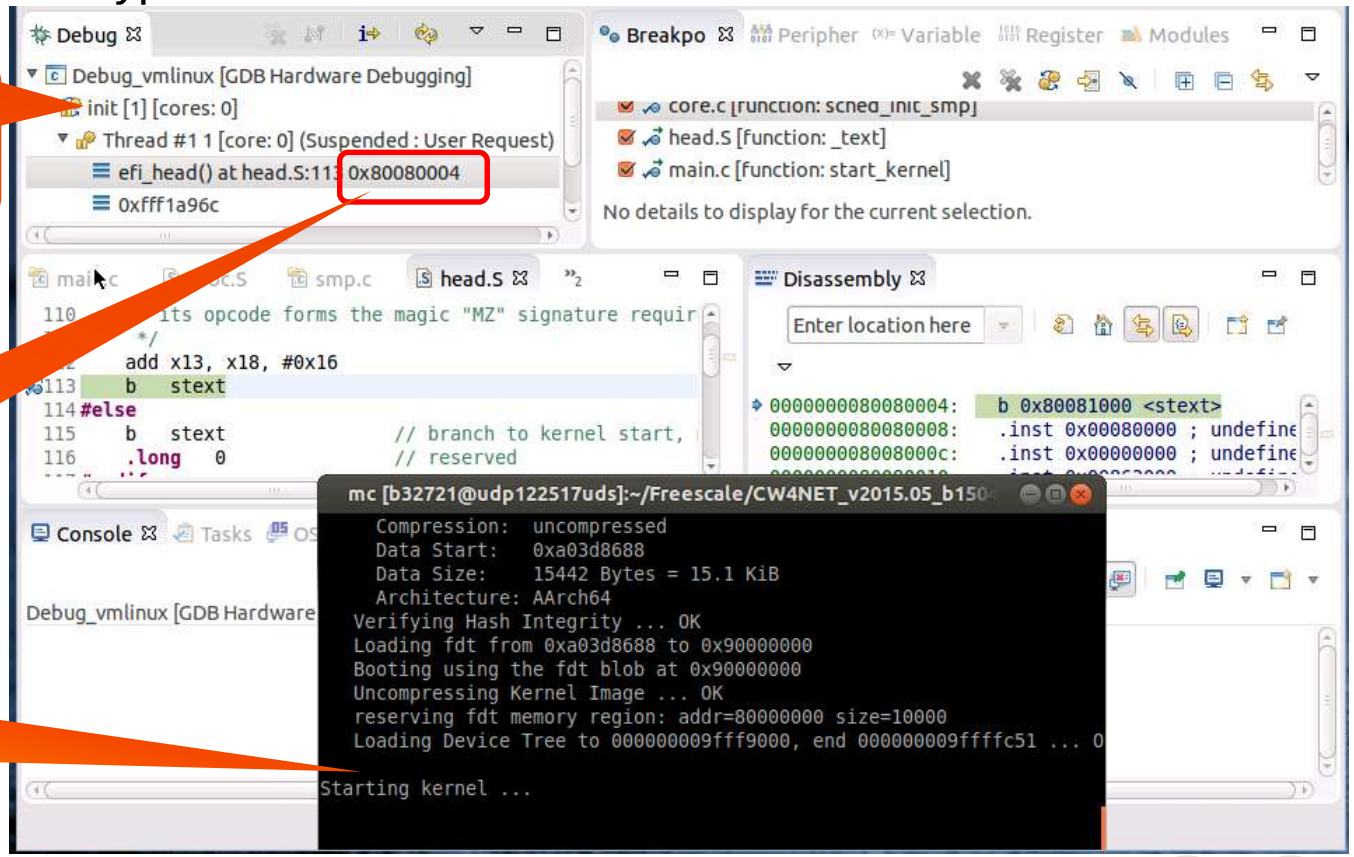
Linux kernel debug – Debug the Linux kernel from the entry point – **Activity (cont)**

- Entry point breakpoints hit
- Note: breakpoint set as SW breakpoint (default) to a memory address before u-boot copy the uImage 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 initalization

Console output stopped at Starting kernel…

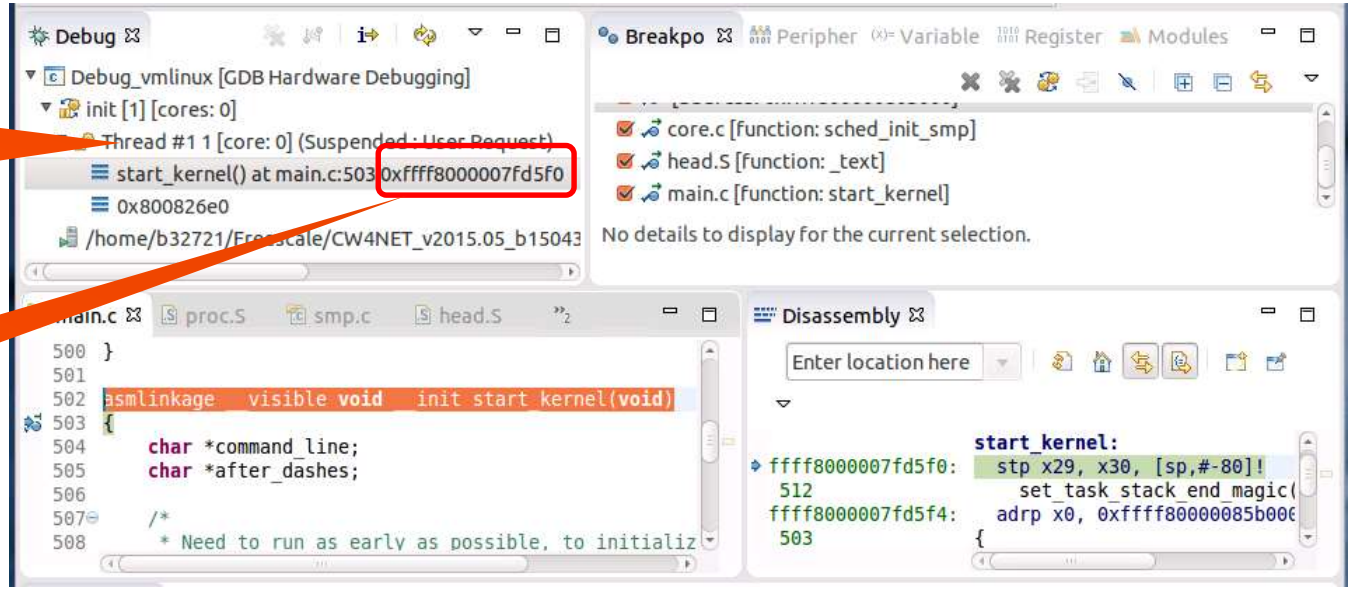


Linux kernel debug – Debug the Linux kernel from the entry point – **Activity (cont)**

- 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

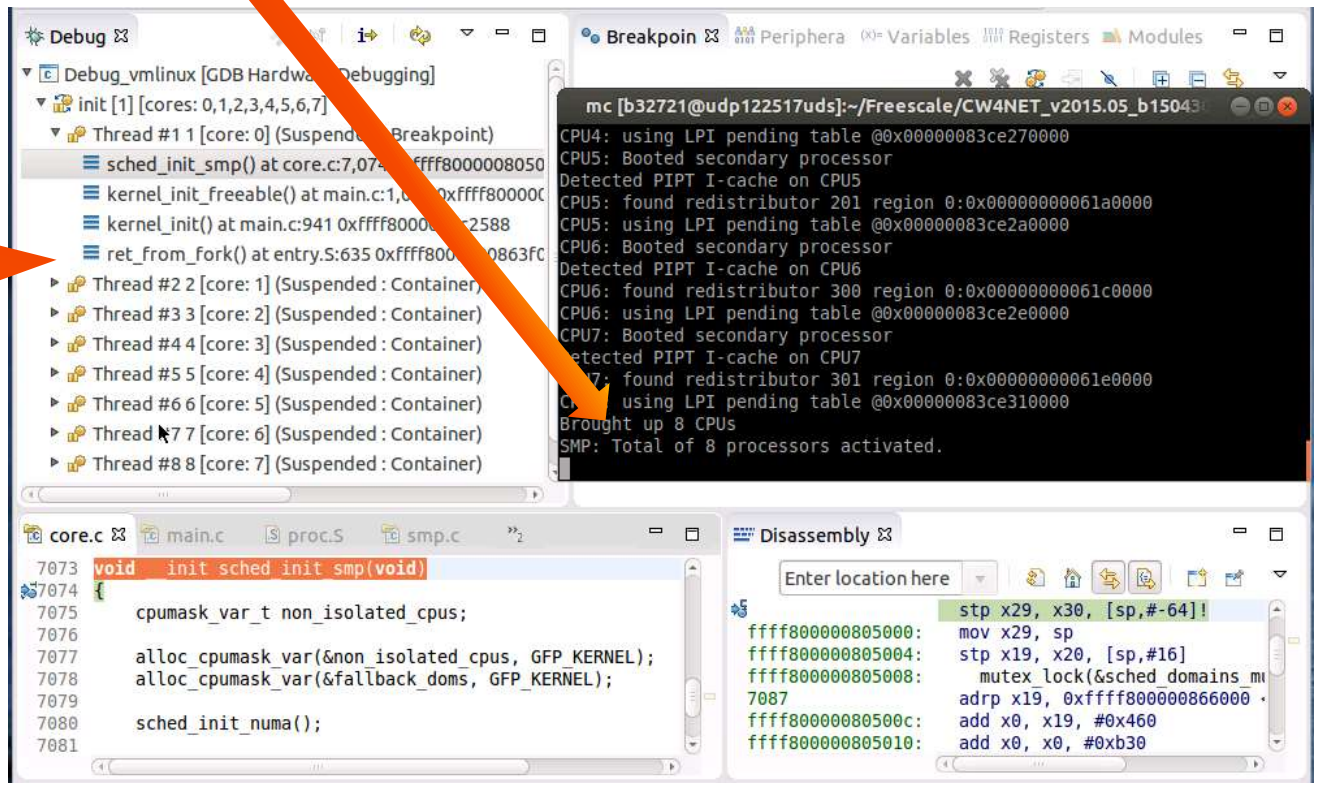
Linux Kernel Awareness automatically apply the symbol relocation corresponding to mapping after MMU initalization



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

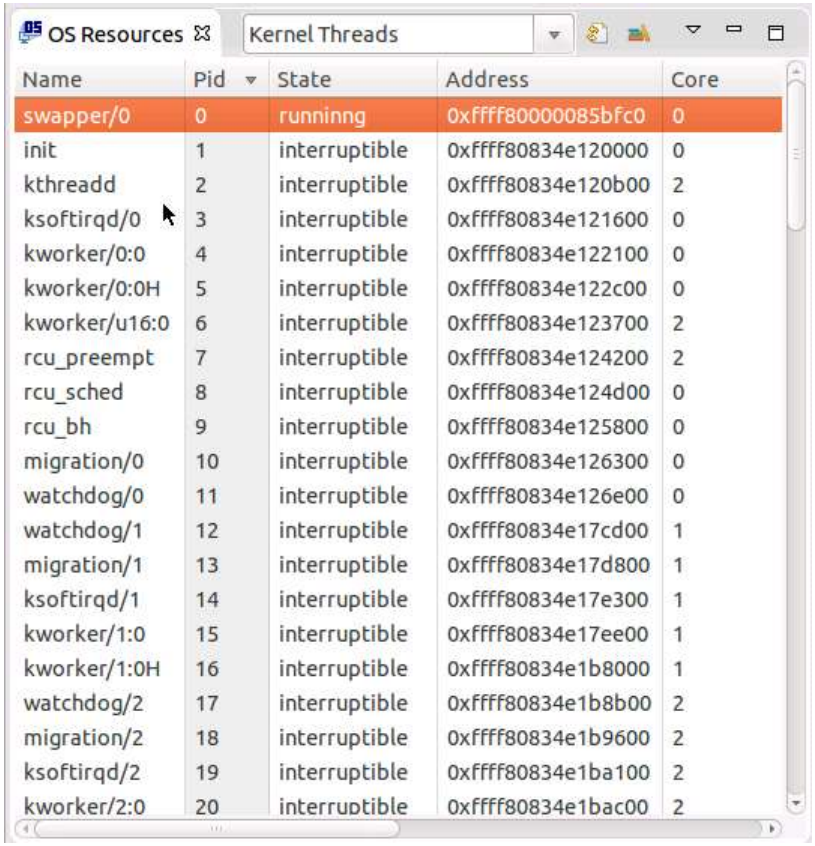
After Linux SMP initialization: 8 cores are shown in the Debug view

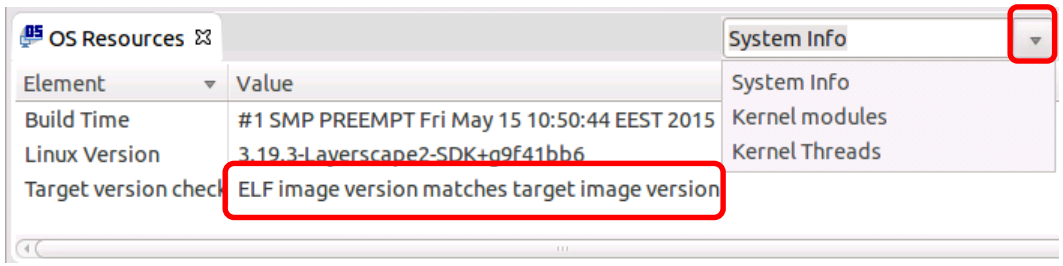


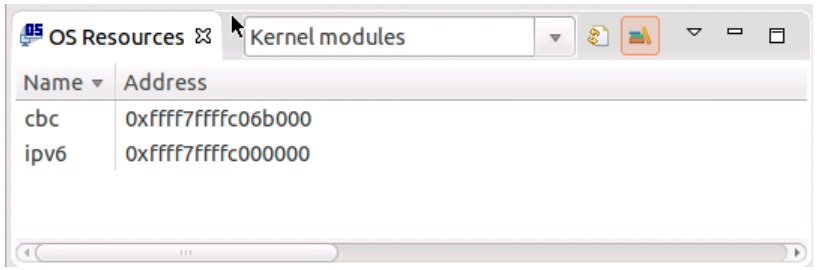
Linux kernel debug – Linux kernel Information– Activity (cont)

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



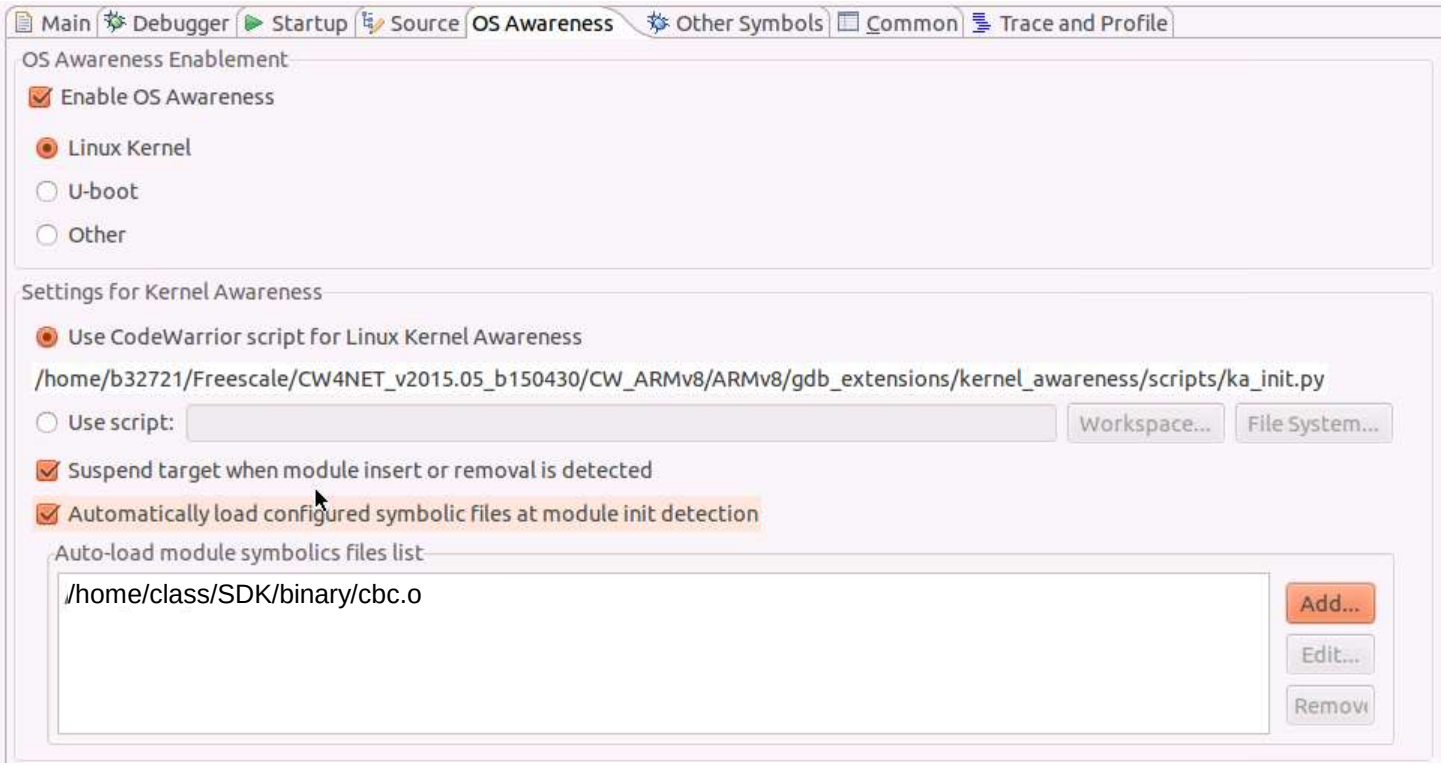




Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

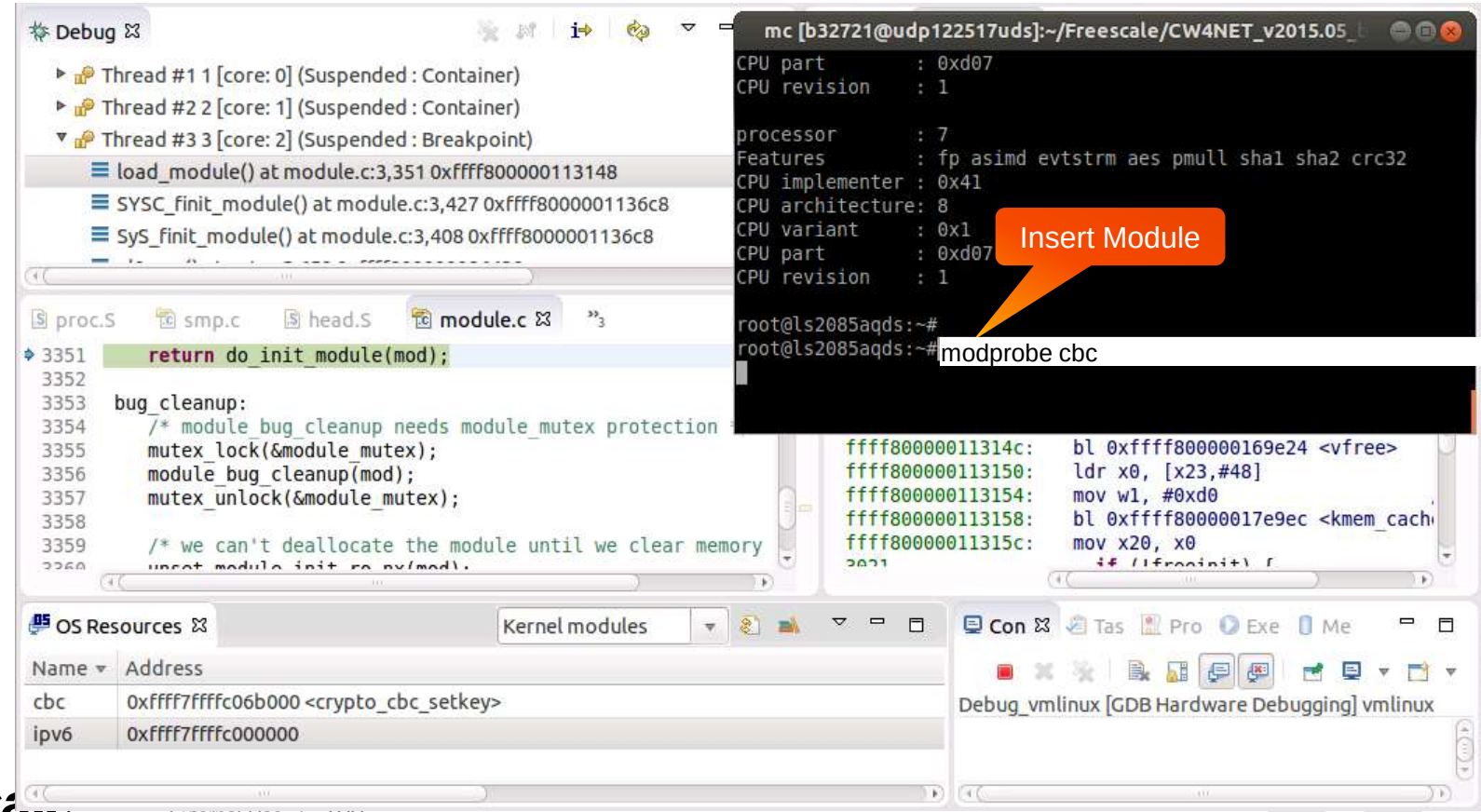
- Configure Kernel module debug parameters: Debug configuration ->OS Awareness tab
  - Check “Suspend target when module insert or removal is detected”
  - Check “Automatically load configured symbolic files at module init detection”.
  - Add the module symbolics files

  Note: you can load the symbolics file also after attaching to linux kernel



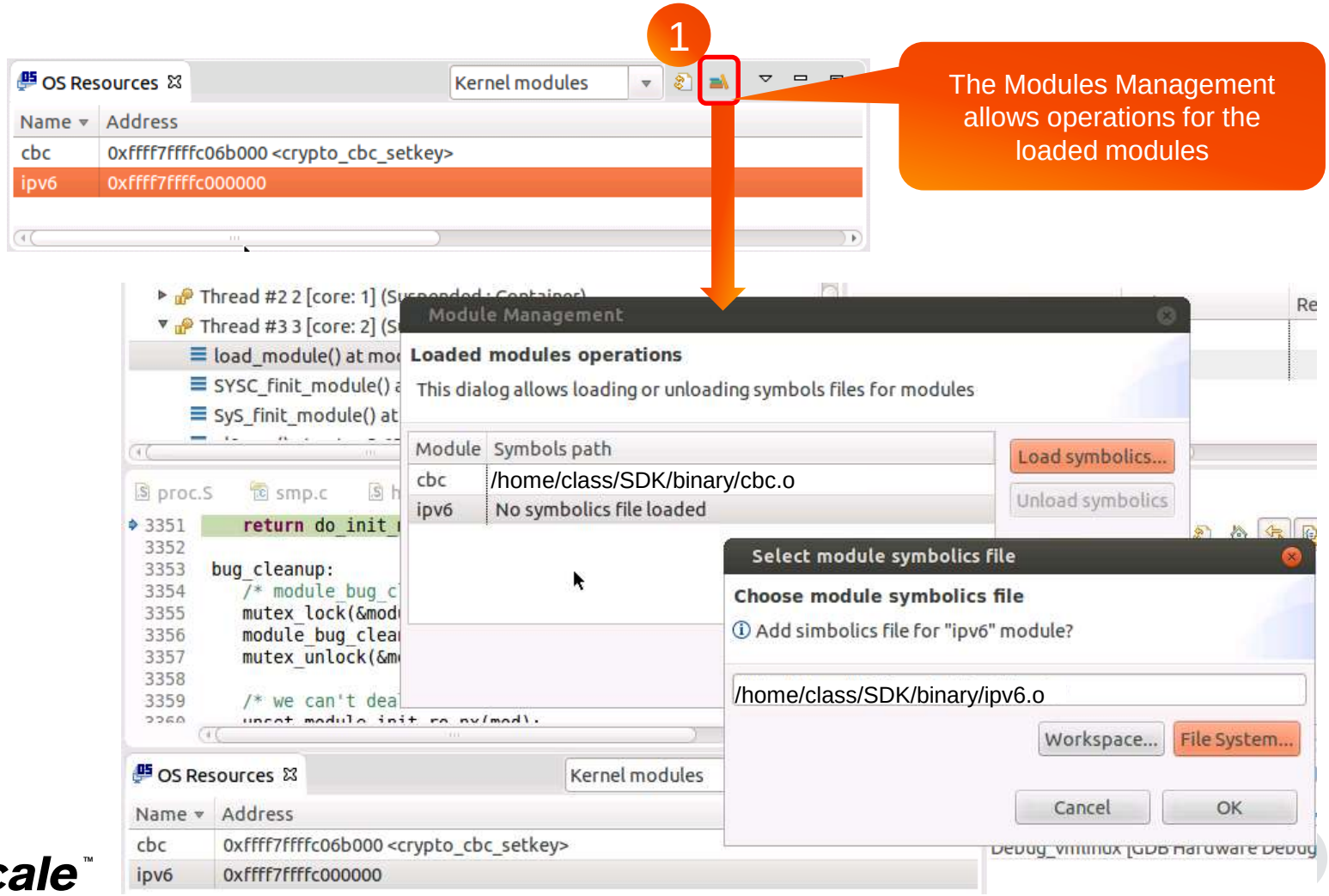
Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

- 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



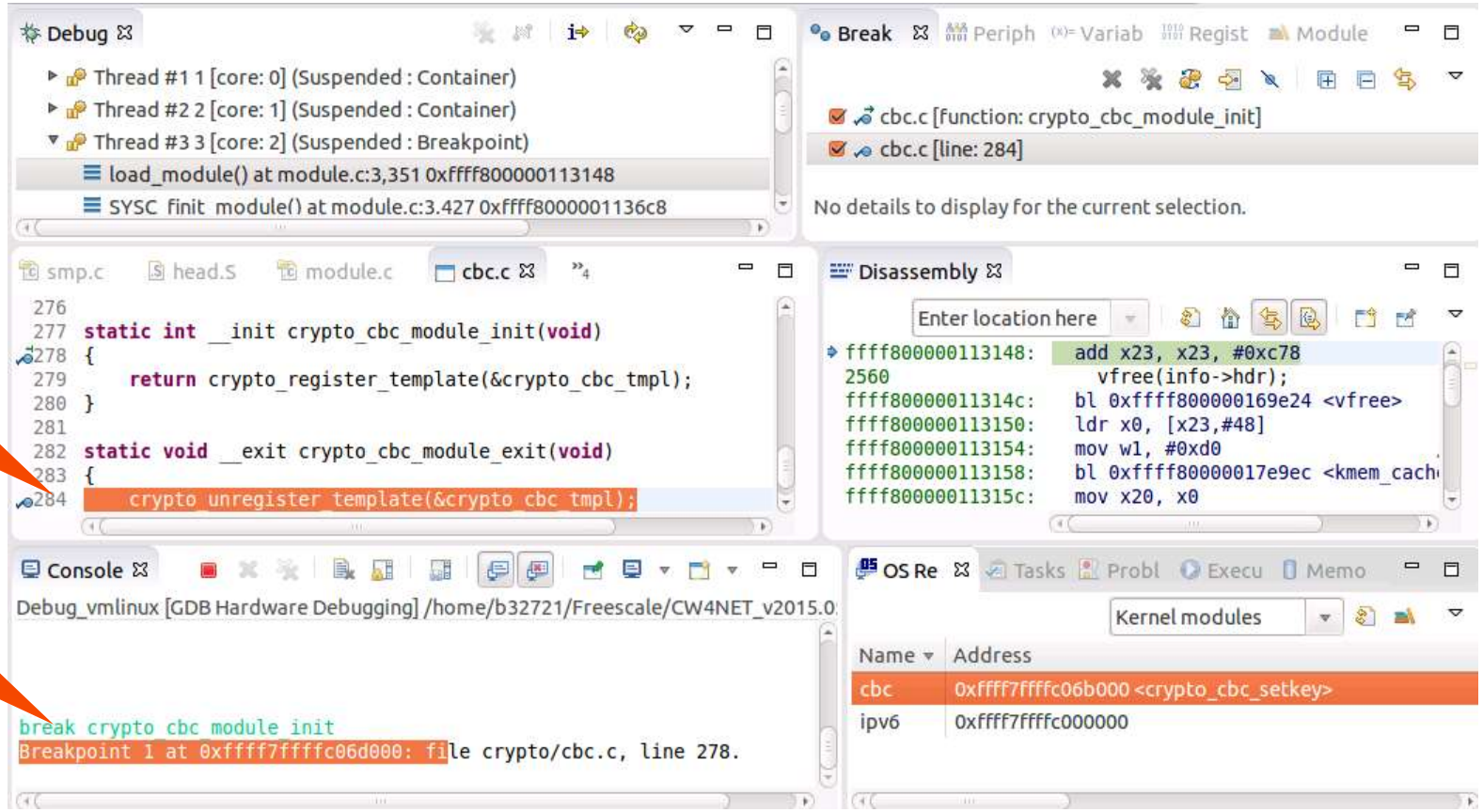
Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

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



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

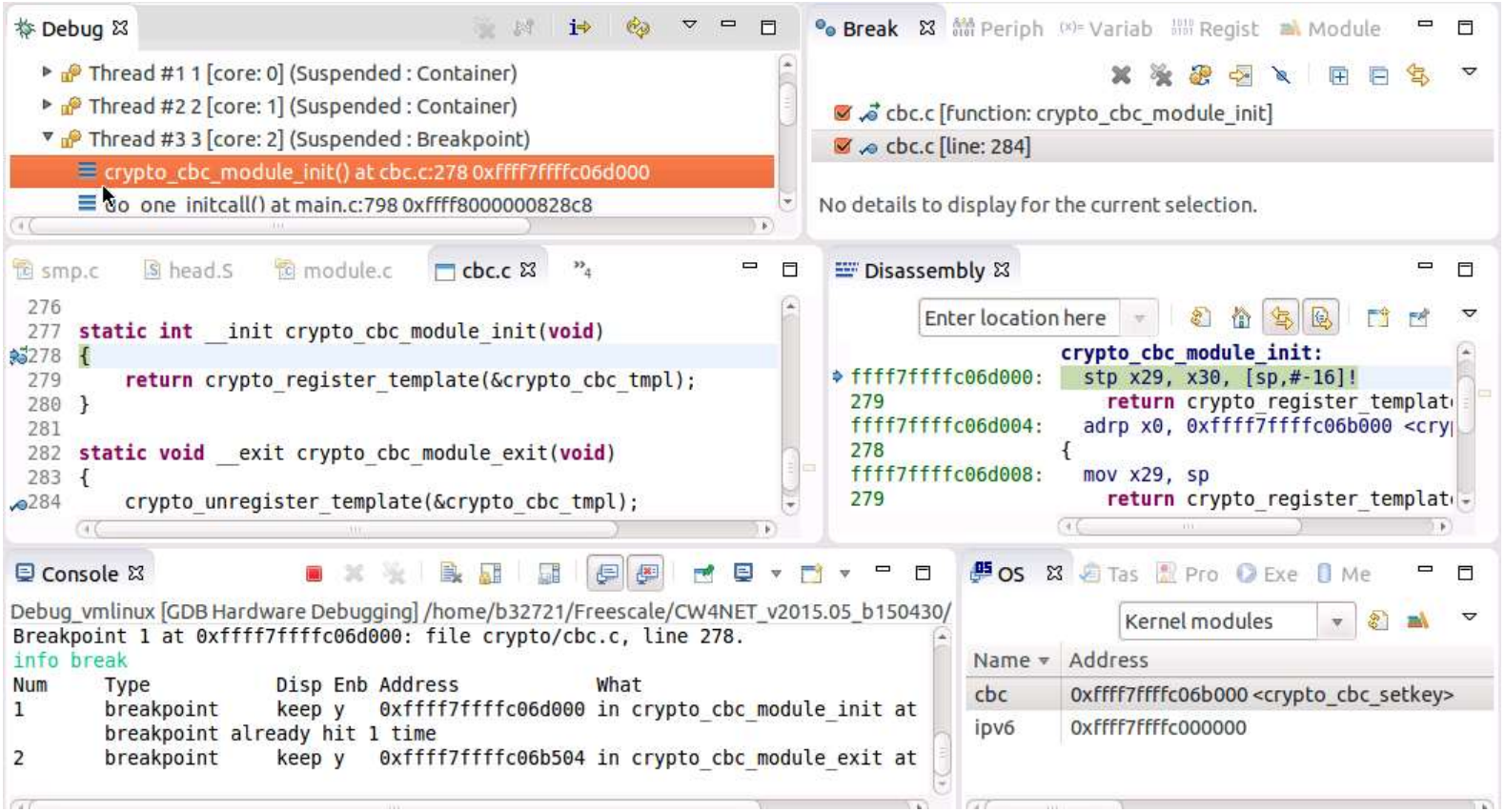


Double-click to set breakpoint

Set breakpoint from console

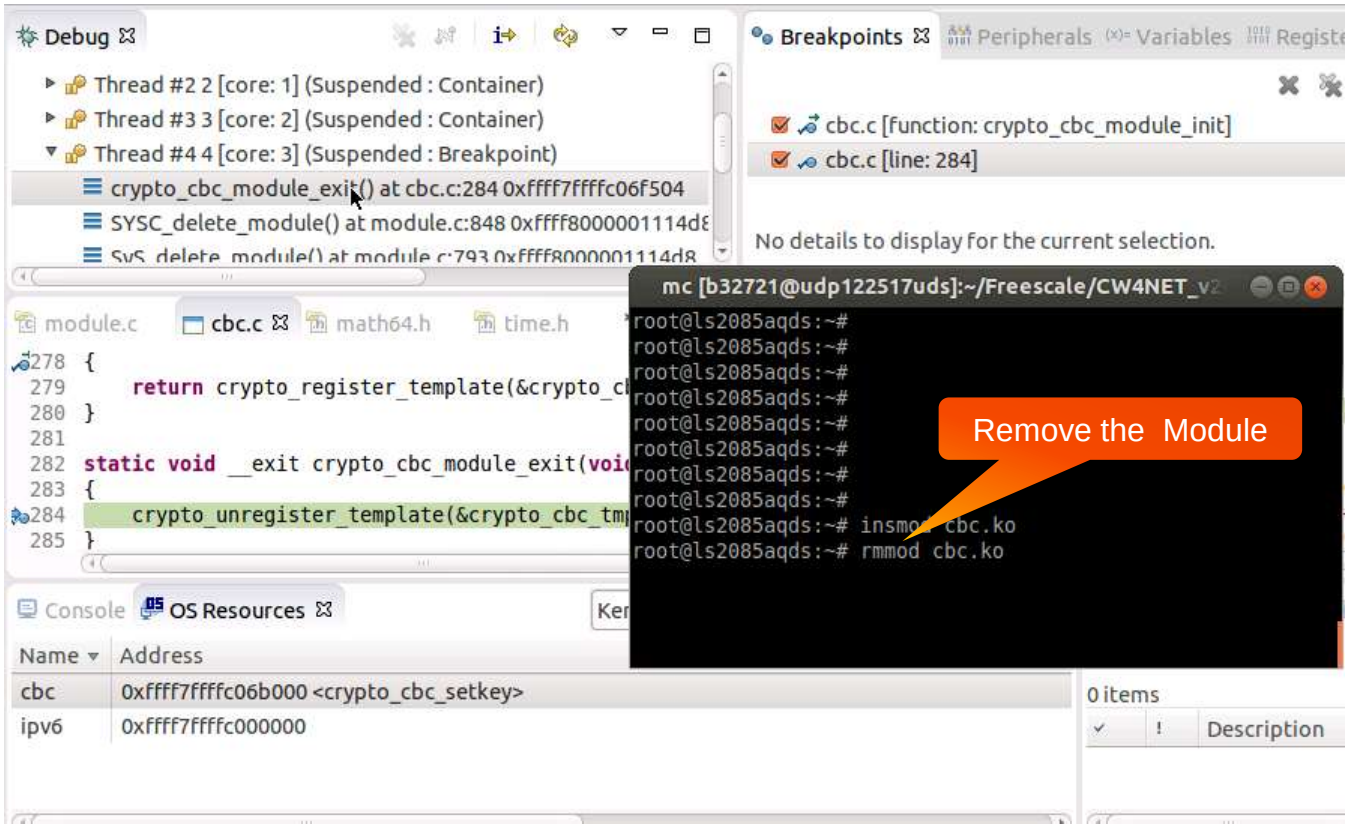
Linux kernel debug – Debugging a Linux kernel module – Activity (cont)

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



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

- Target stops at module exit
- Perform debug as usual



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

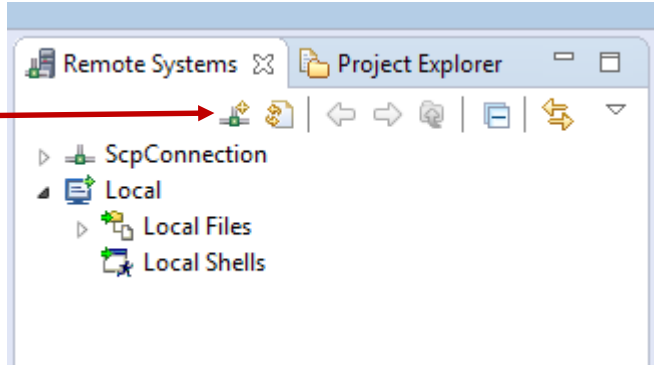
Activity

Trace Compass

Trace Compass – new RSE connection

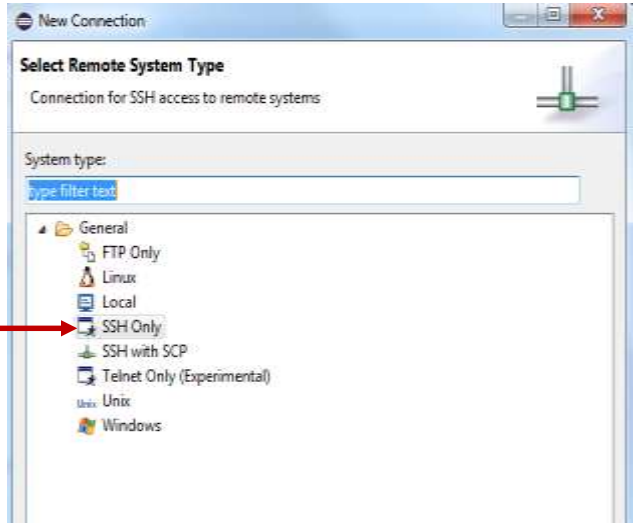
1. Open Remote Systems view (Window->Show View->Other->Remote Systems->Remote Systems)

New RSE connection



2. Create a Linux based RSE connection

SSH Only

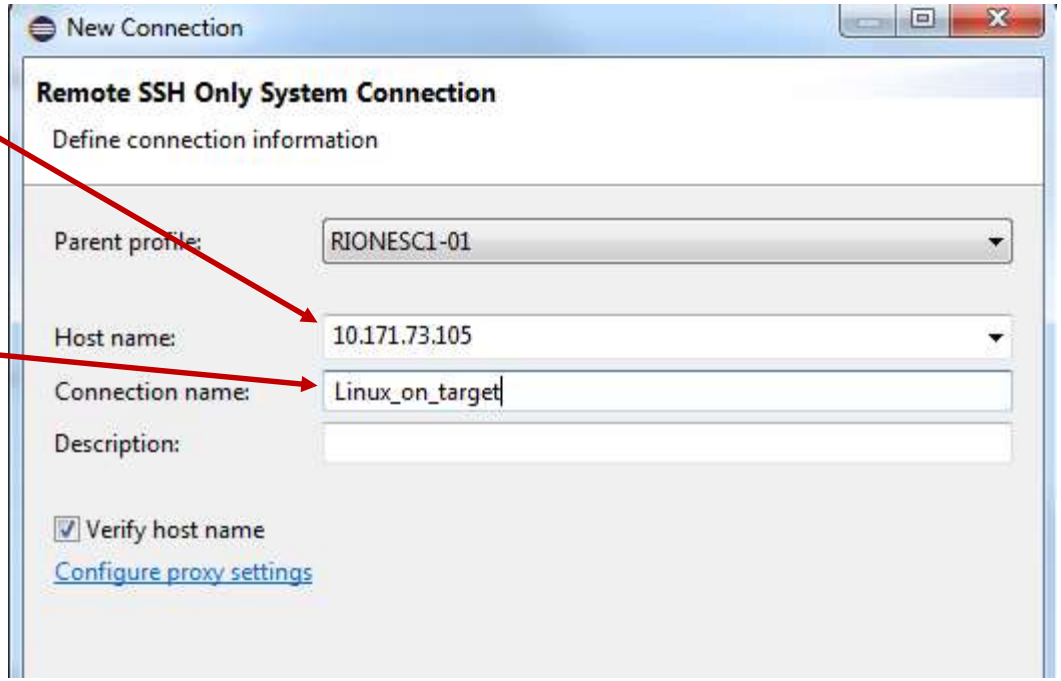


Trace Compass – new RSE connection

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

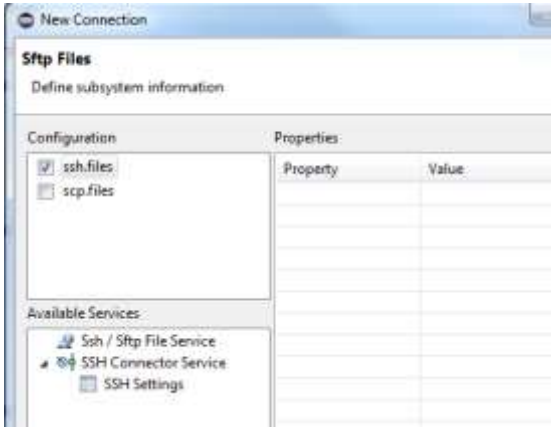
IP of target

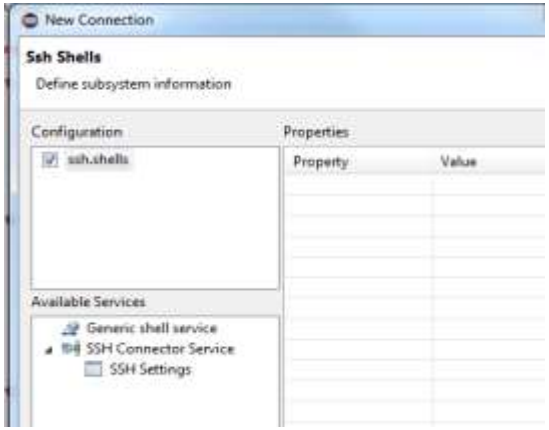
Name the connection

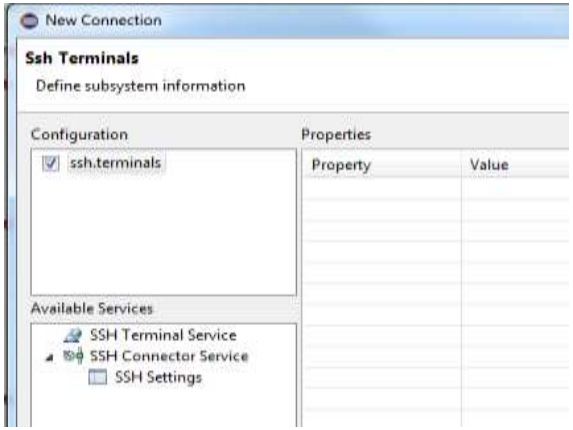


Trace Compass – new RSE connection

4. Continue to follow RSE connection creation wizard

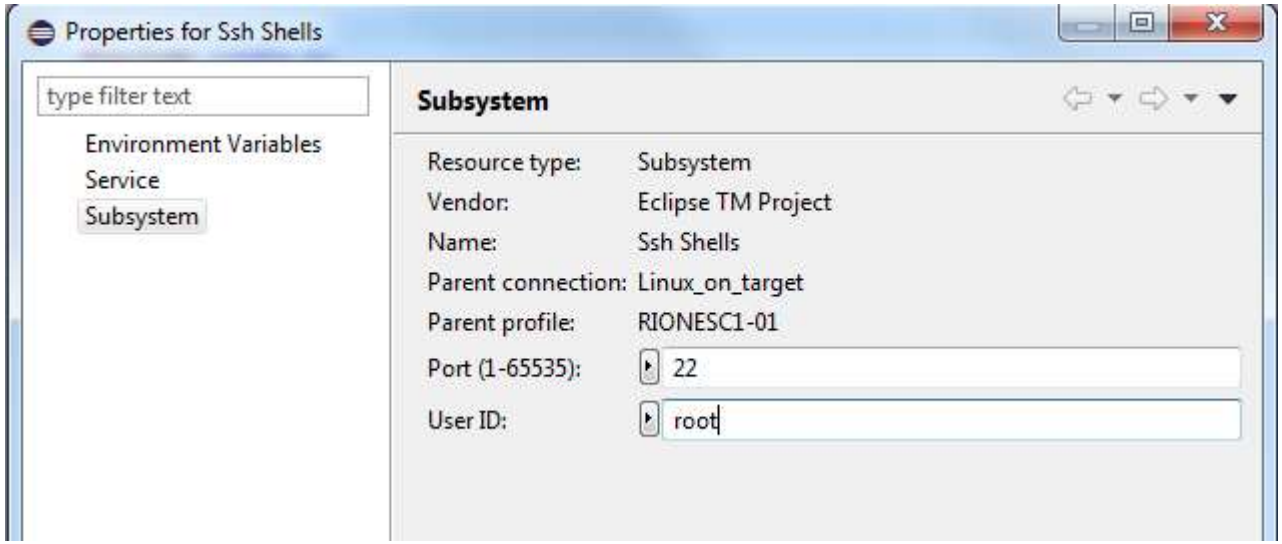






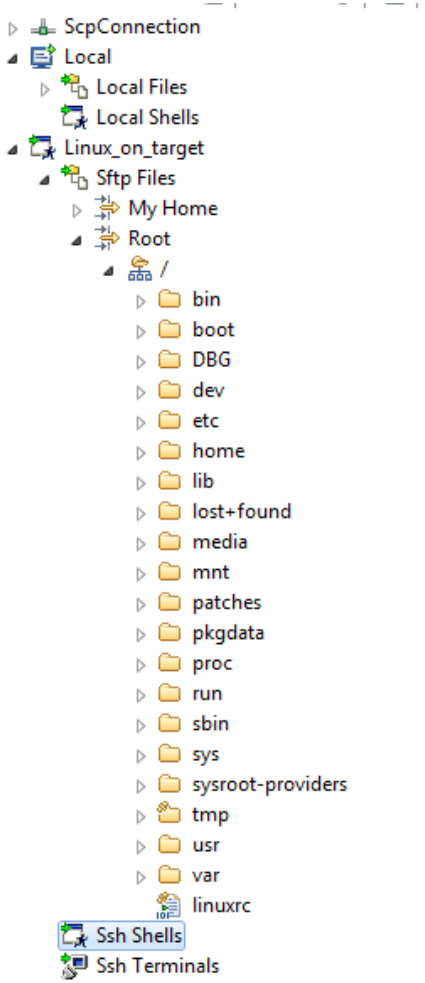
Trace Compass – new RSE connection

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:

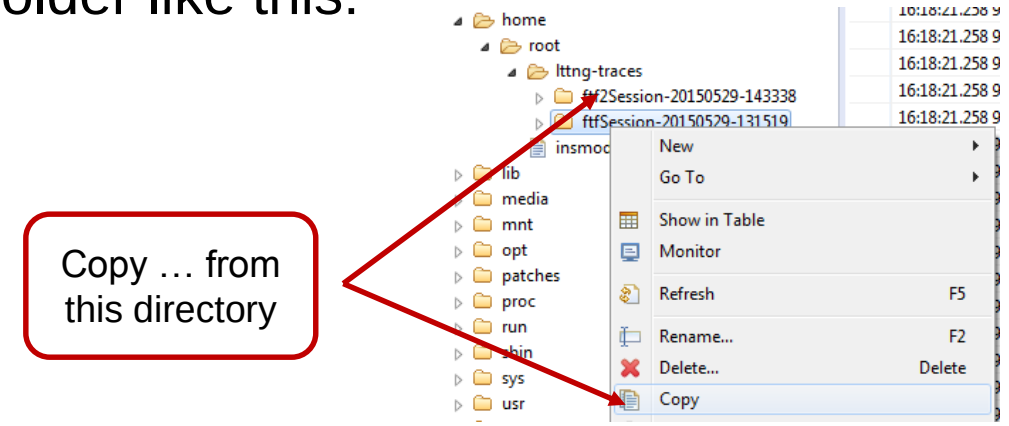


Trace Compass – trace session

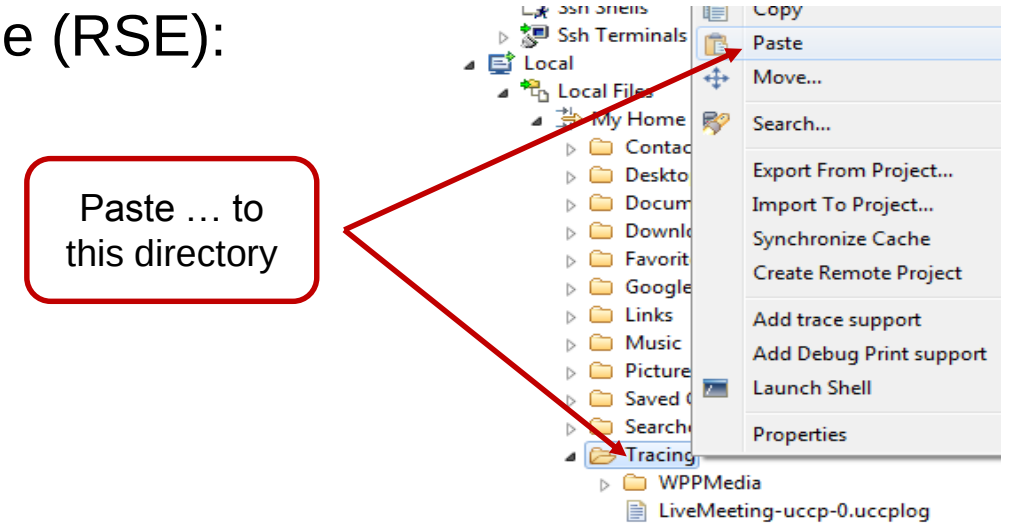
1. Open a Terminal over a RSE connection from CodeWarrior
2. Load LTTng modules:
   modprobe lttng-tracer
3. Check that LTTng modules are loaded:
   lsmod
4. Create a new LTTng session:
   lttng create ftfSession
5. Enable all events for Kernel tracing:
   lttng enable-event --kernel --all
6. Start tracing 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 (*lttng-traces*)
11. Copy the session folder like this:

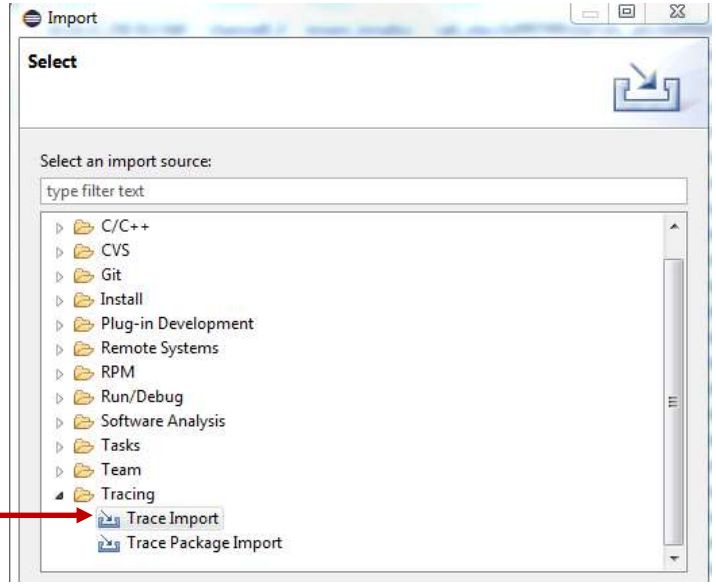


12. Paste in Local node (RSE):



Trace Compass – trace session

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



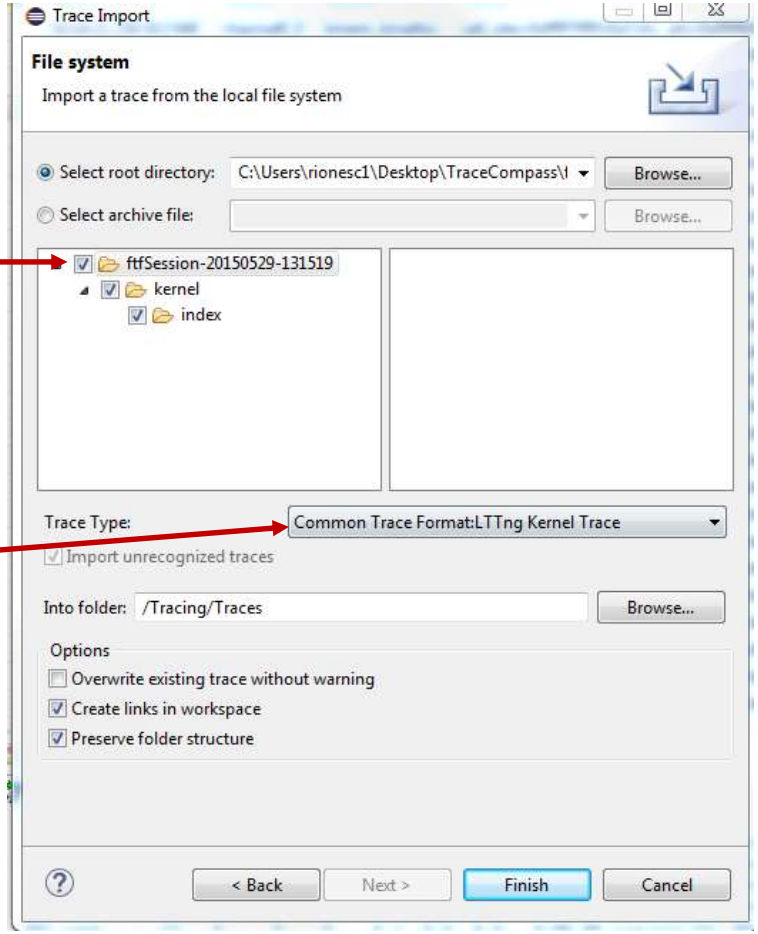
Import trace

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

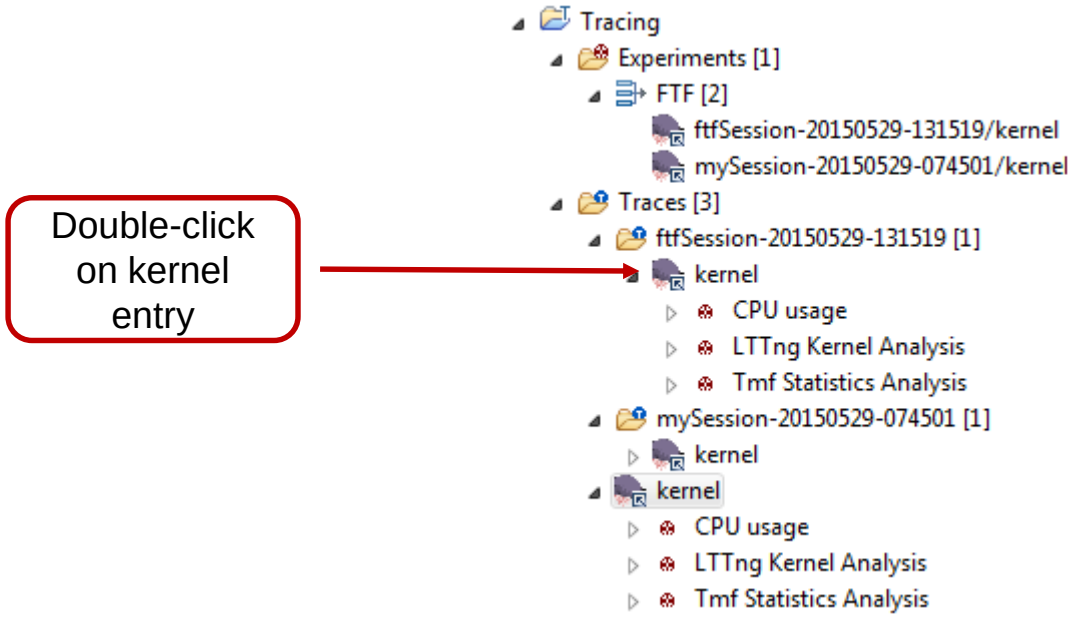
Select Trace Type



Trace Compass – trace session

17. Open *LTTng Kernel* perspective

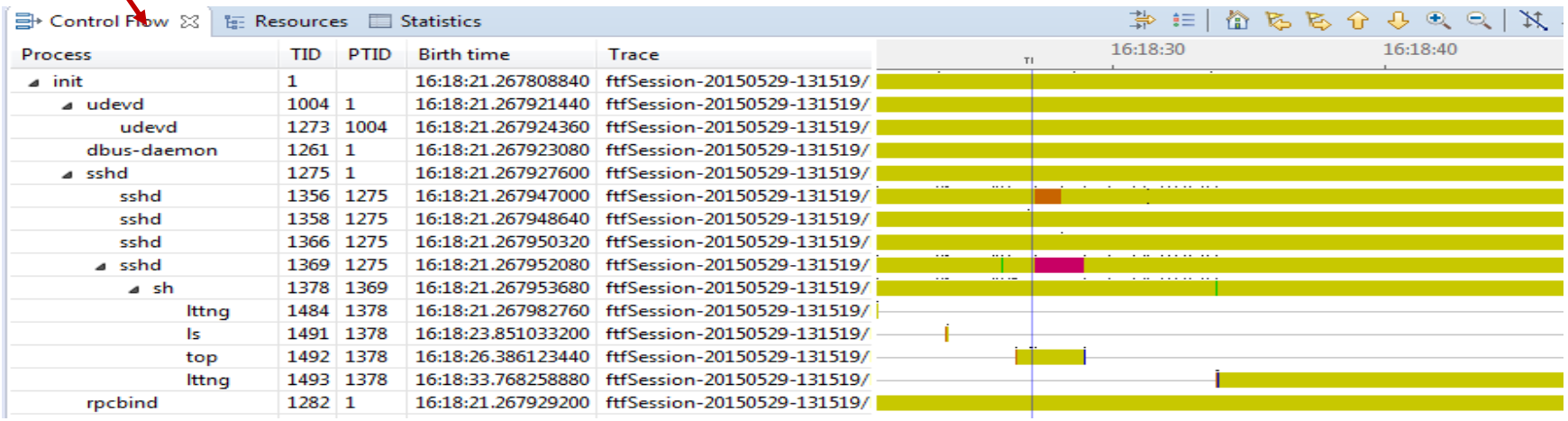
18. Double-click on imported trace session from *Project Explorer* view (kernel entry):



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

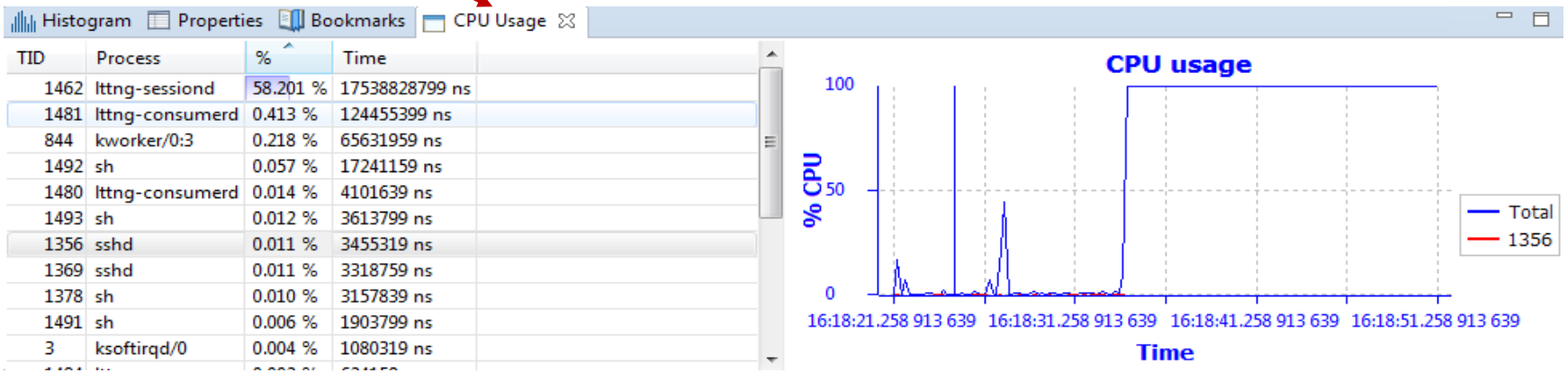
Trace Compass views

Control Flow



Process	TID	PTID	Birth time	Trace
init	1		16:18:21.267808840	ftfSession-20150529-131519/
udevd	1004	1	16:18:21.267921440	ftfSession-20150529-131519/
udevd	1273	1004	16:18:21.267924360	ftfSession-20150529-131519/
dbus-daemon	1261	1	16:18:21.267923080	ftfSession-20150529-131519/
sshd	1275	1	16:18:21.267927600	ftfSession-20150529-131519/
sshd	1356	1275	16:18:21.267947000	ftfSession-20150529-131519/
sshd	1358	1275	16:18:21.267948640	ftfSession-20150529-131519/
sshd	1366	1275	16:18:21.267950320	ftfSession-20150529-131519/
sshd	1369	1275	16:18:21.267952080	ftfSession-20150529-131519/
sh	1378	1369	16:18:21.267953680	ftfSession-20150529-131519/
lttng	1484	1378	16:18:21.267982760	ftfSession-20150529-131519/
ls	1491	1378	16:18:23.851033200	ftfSession-20150529-131519/
top	1492	1378	16:18:26.386123440	ftfSession-20150529-131519/
lttng	1493	1378	16:18:33.768258880	ftfSession-20150529-131519/
rpcbind	1282	1	16:18:21.267929200	ftfSession-20150529-131519/

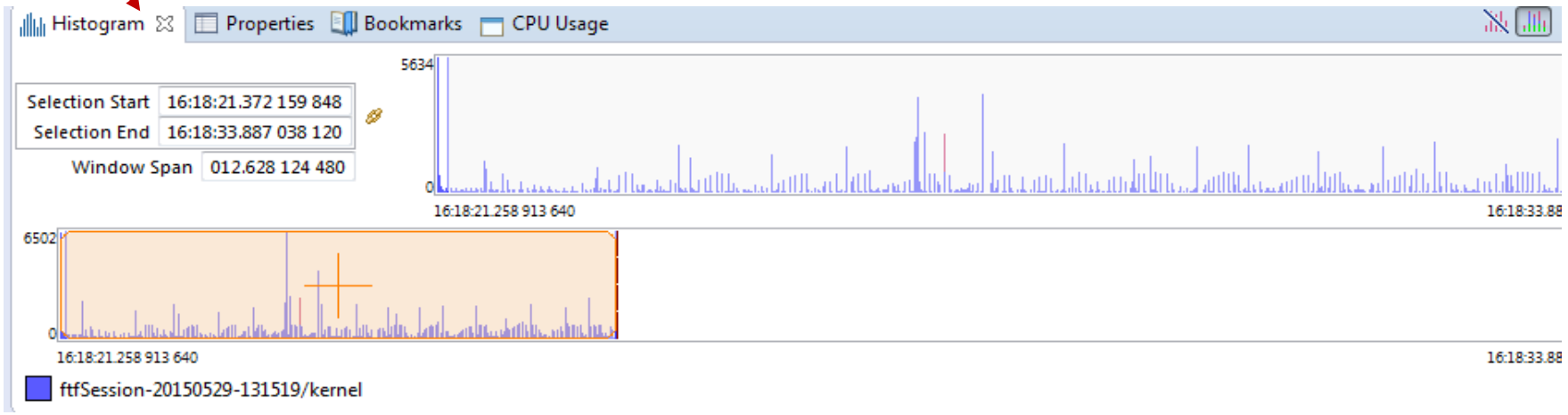
CPU Usage



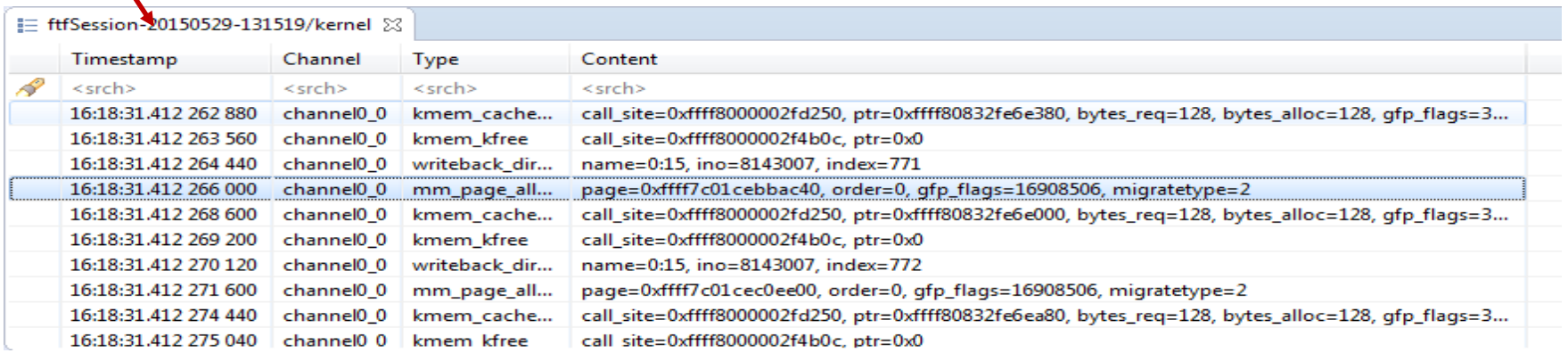
TID	Process	%	Time
1462	lttng-sessiond	58.201 %	17538828799 ns
1481	lttng-consumerd	0.413 %	124455399 ns
844	kworker/0:3	0.218 %	65631959 ns
1492	sh	0.057 %	17241159 ns
1480	lttng-consumerd	0.014 %	4101639 ns
1493	sh	0.012 %	3613799 ns
1356	sshd	0.011 %	3455319 ns
1369	sshd	0.011 %	3318759 ns
1378	sh	0.010 %	3157839 ns
1491	sh	0.006 %	1903799 ns
3	ksoftirqd/0	0.004 %	1080319 ns

Trace Compass views

Histogram

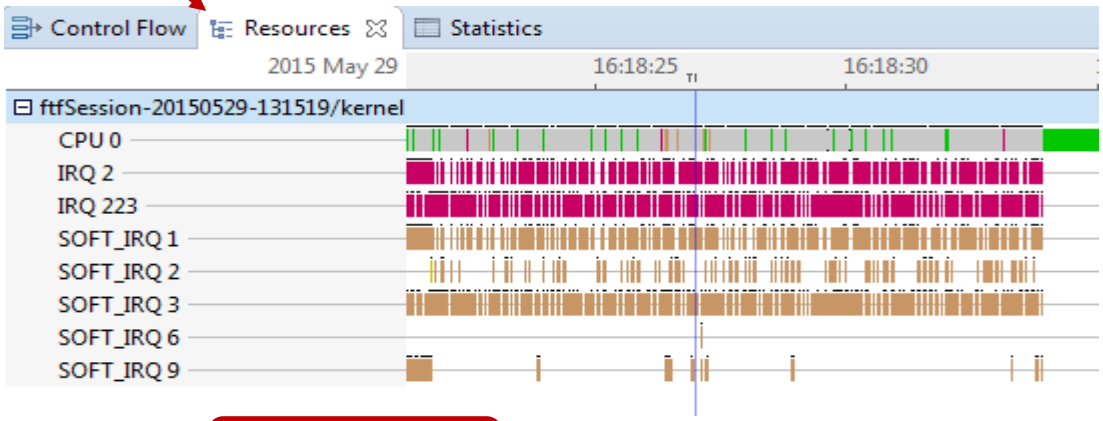


Events

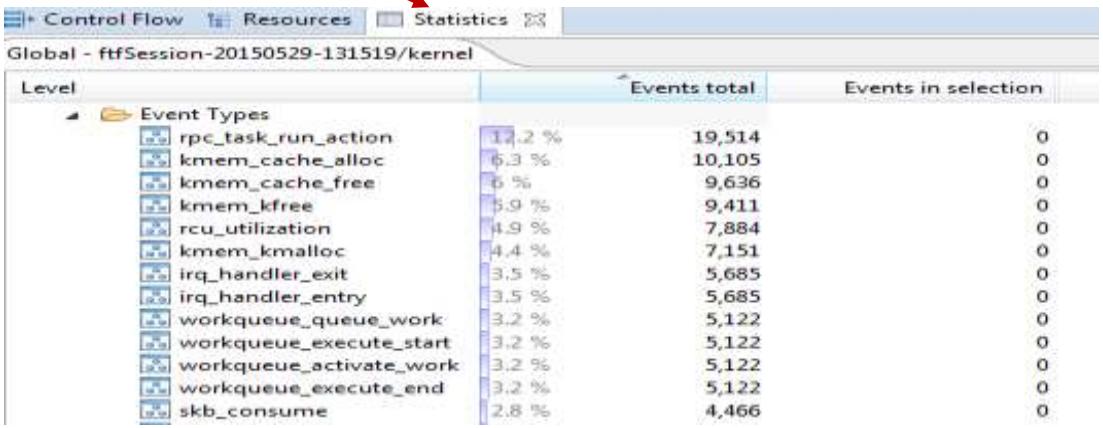


Trace Compass views

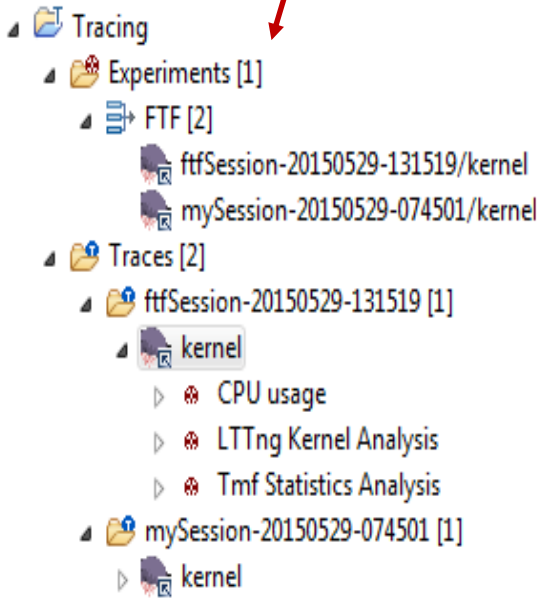
Resources



Statistics

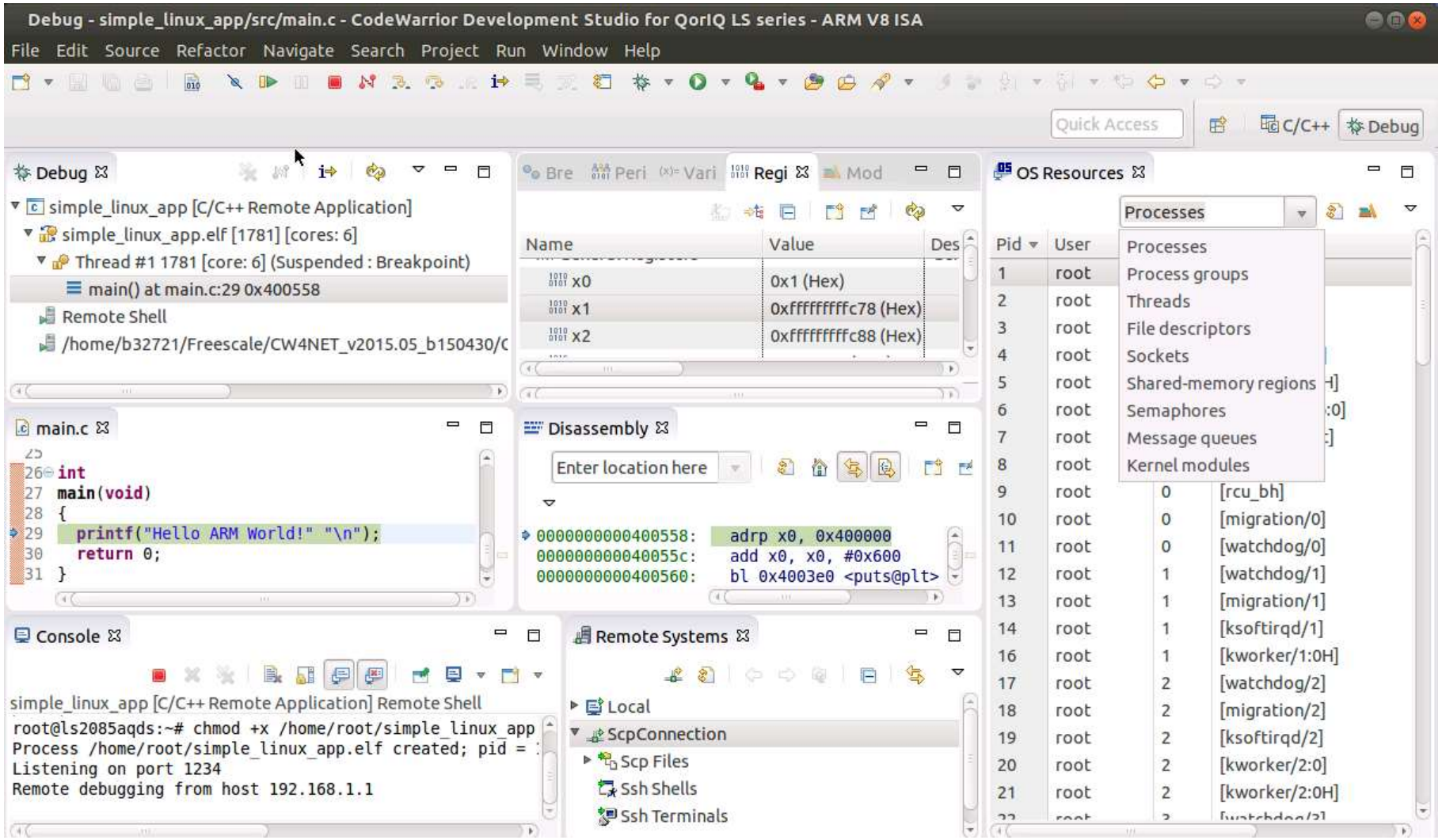


Project Explorer – Trace Compass session



Linux Application Debug

CodeWarrior– Debugging ARM Target



Two ways to run GDB

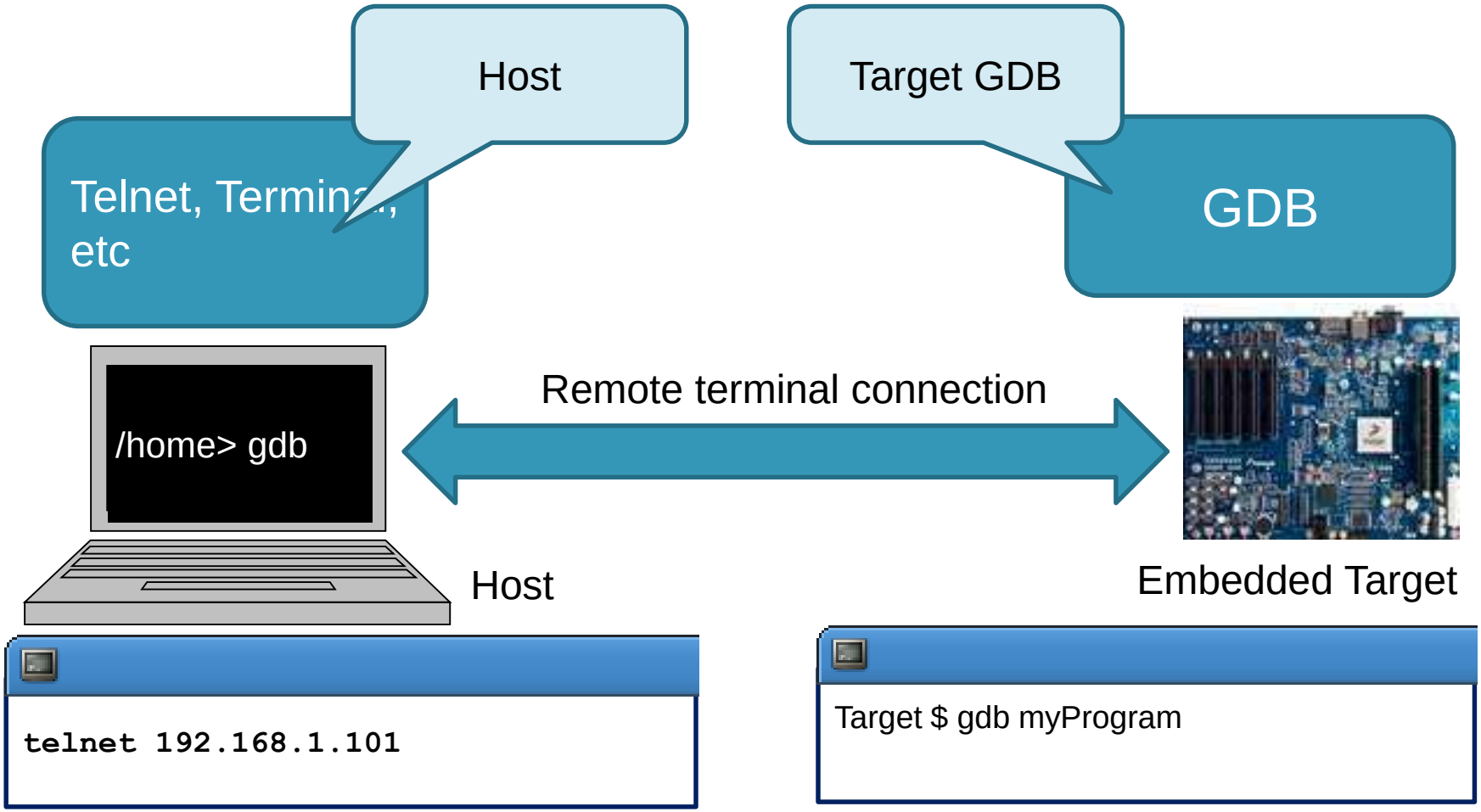
Target (self-hosted)

- 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

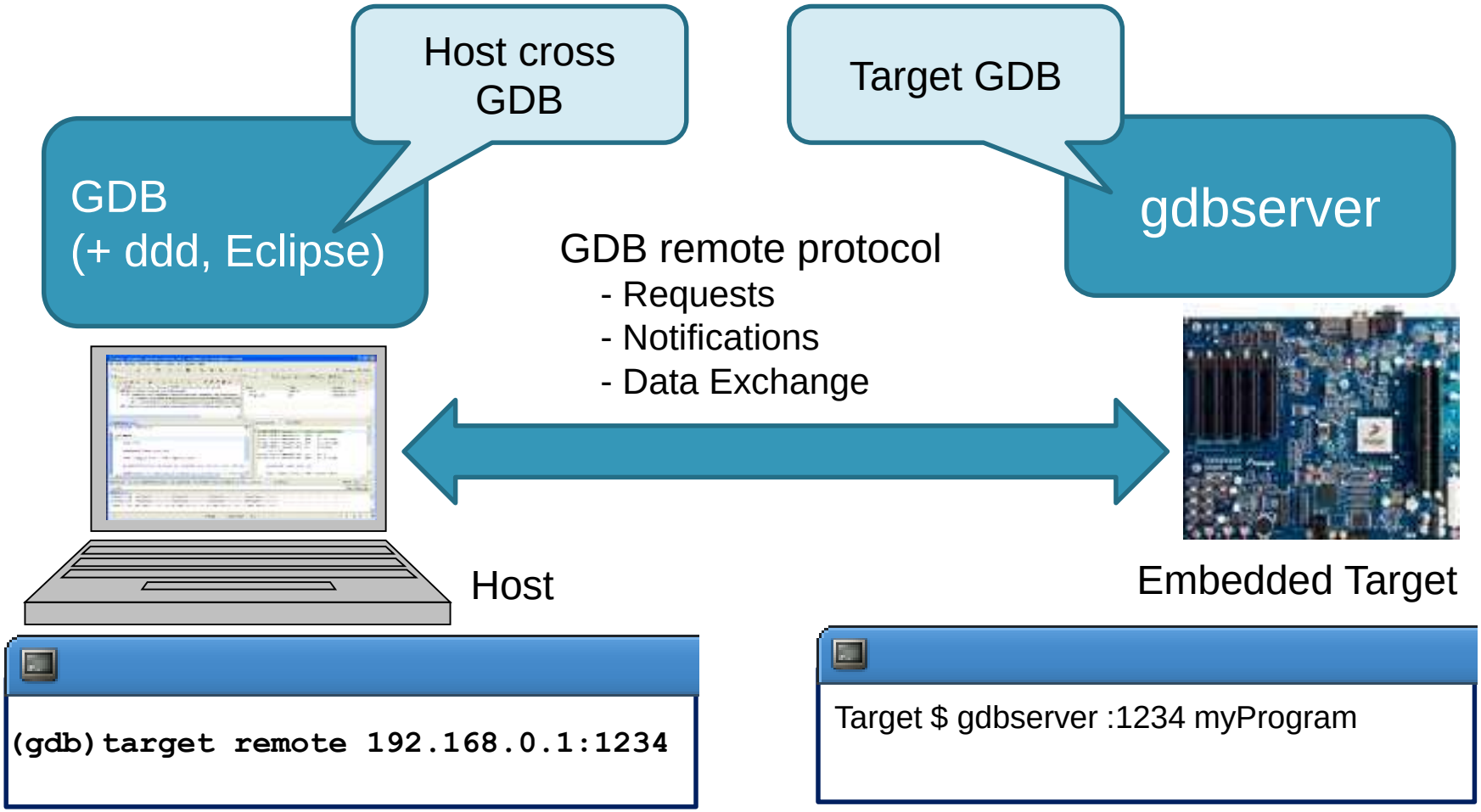
Native (Host)

- 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



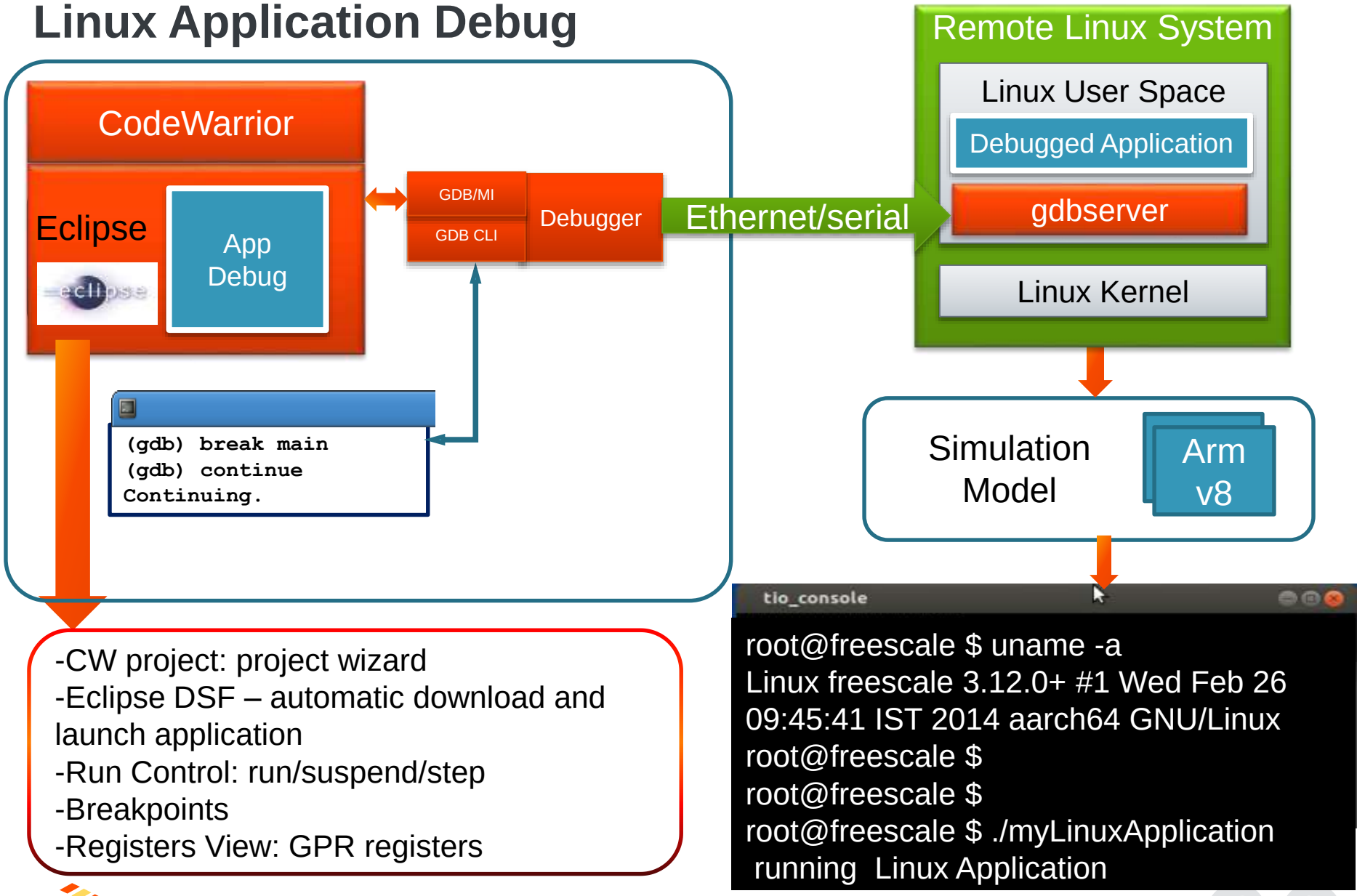
Linux application debug – Capabilities

- **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



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

Standalone (CLI) Linux Application debug

- Compiling the application

```
$ aarch64-linux-gnu-gc-linux-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
```

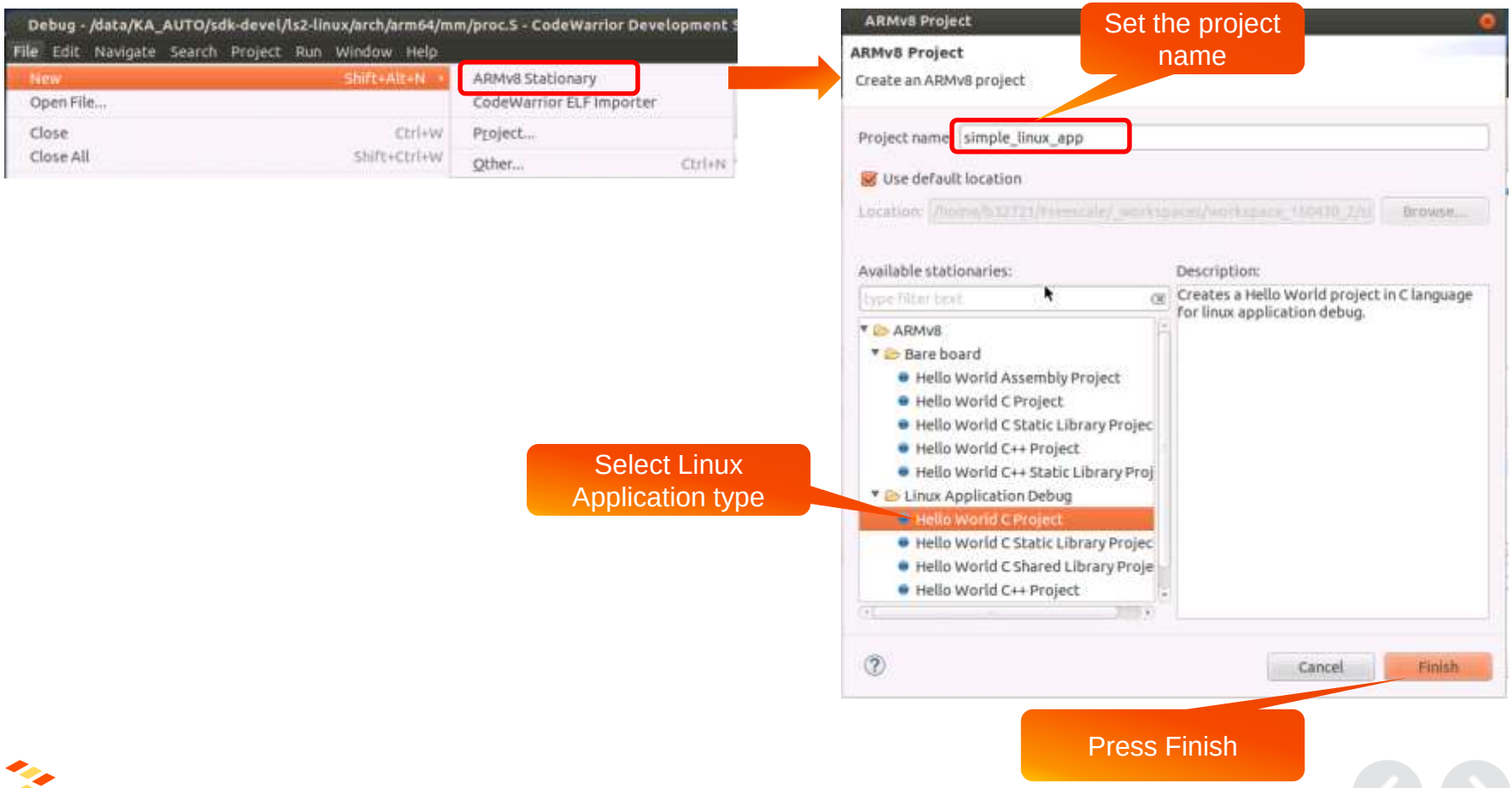
- GDB commands
  - run/Ctrl-C(stop), next, step, nexti, stepi, breakpoint
  - backtrace, frame, print, x, dump
  - set, call, jump, return

Activity

Linux Application Debug – Simple Example

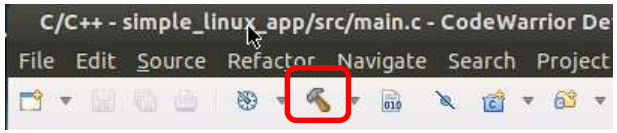
Linux application debug – Debugging a simple Linux Application debug project – **Activity**

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

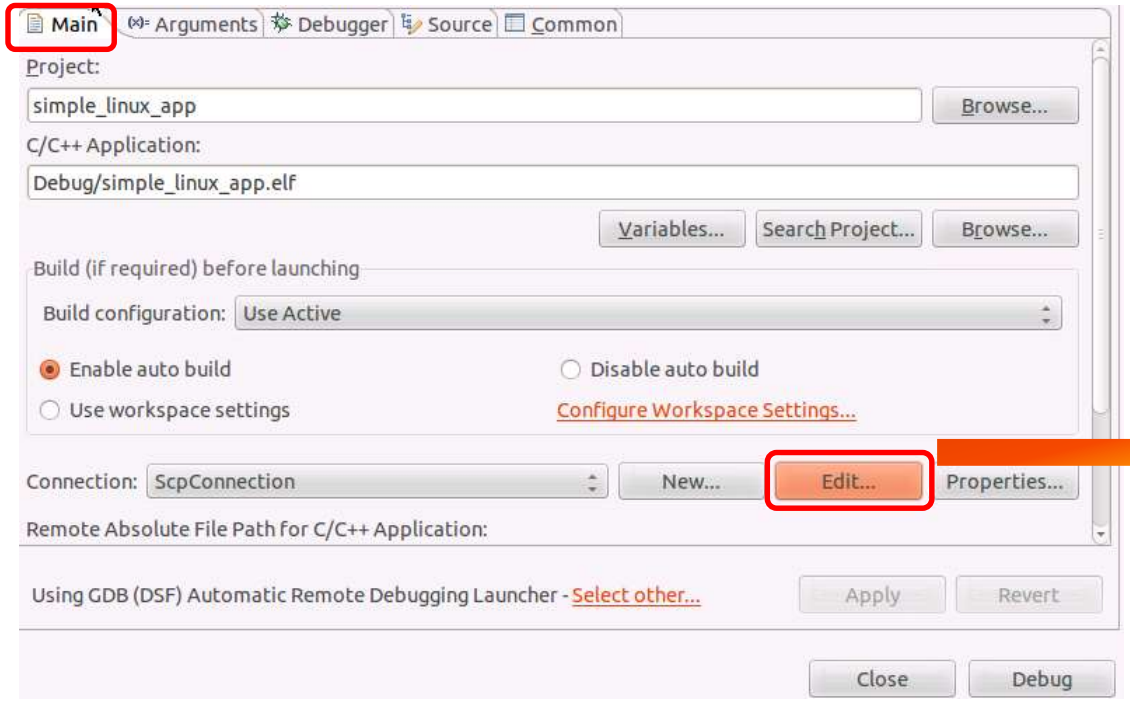


Linux application debug – Debugging a simple Linux Application debug project – **Activity**

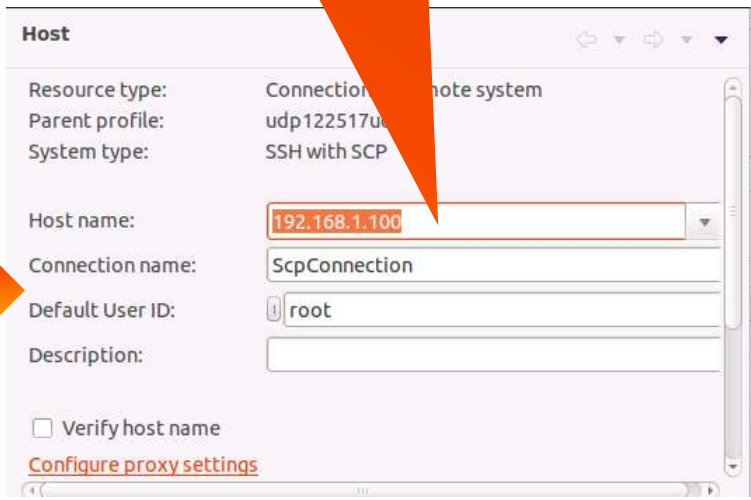
- Build Project



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



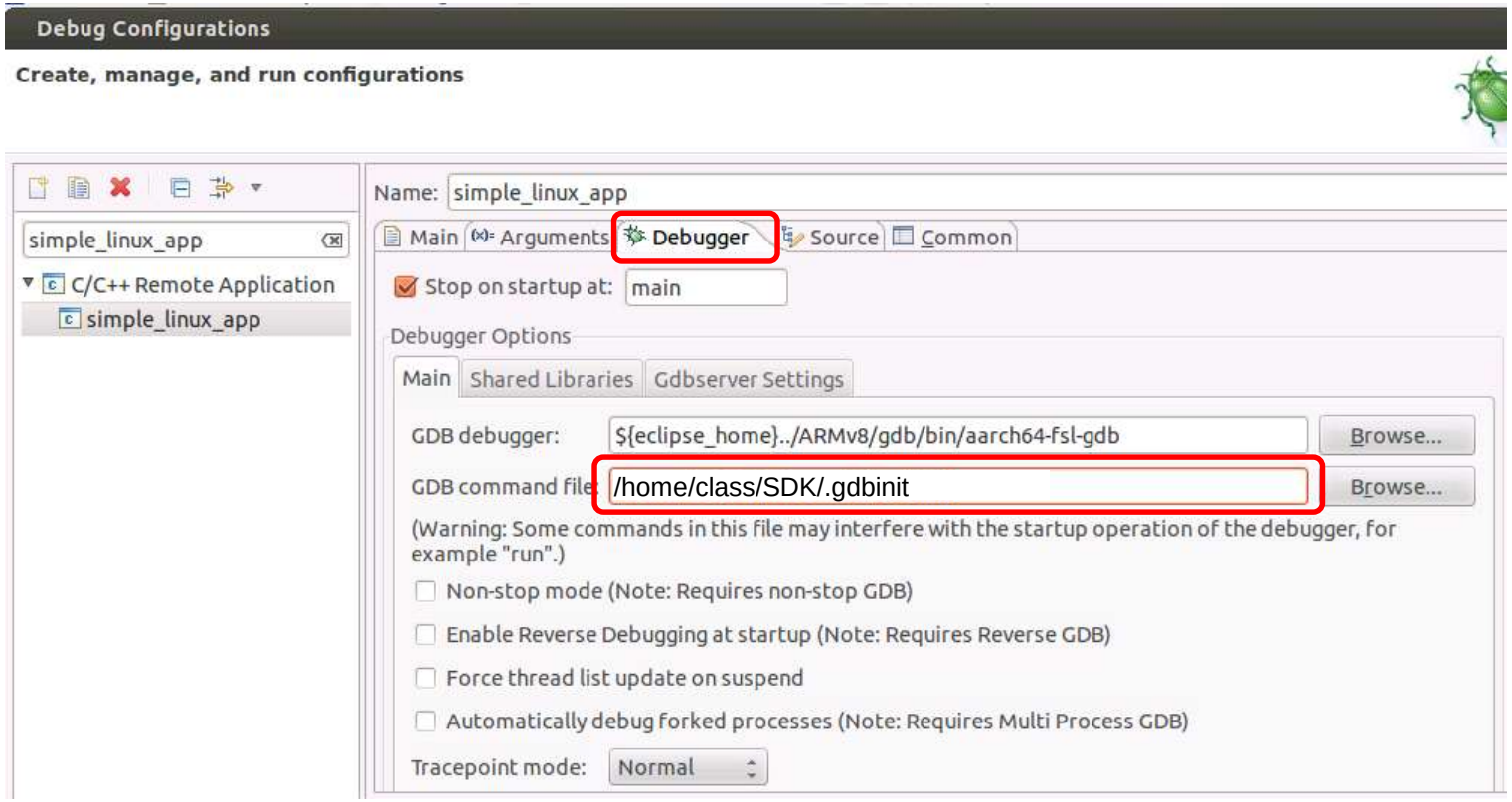
Set the IP address of the remote Linux target



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

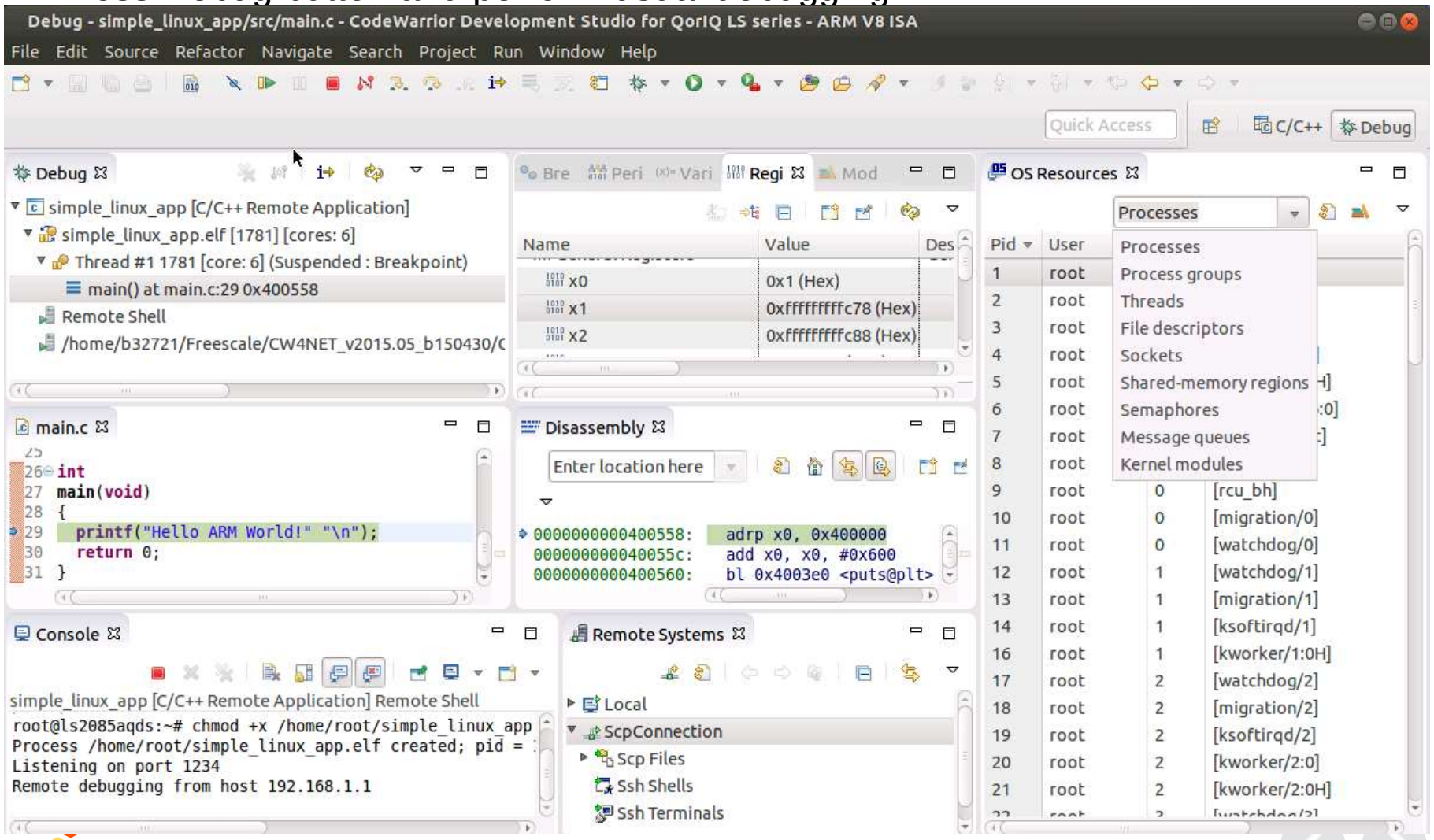
- Set the gdb initialization file where the sysroot is set. .gdbinit file contains:

set sysroot <YoctoInstallationPath>/fsl-qoriq-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



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

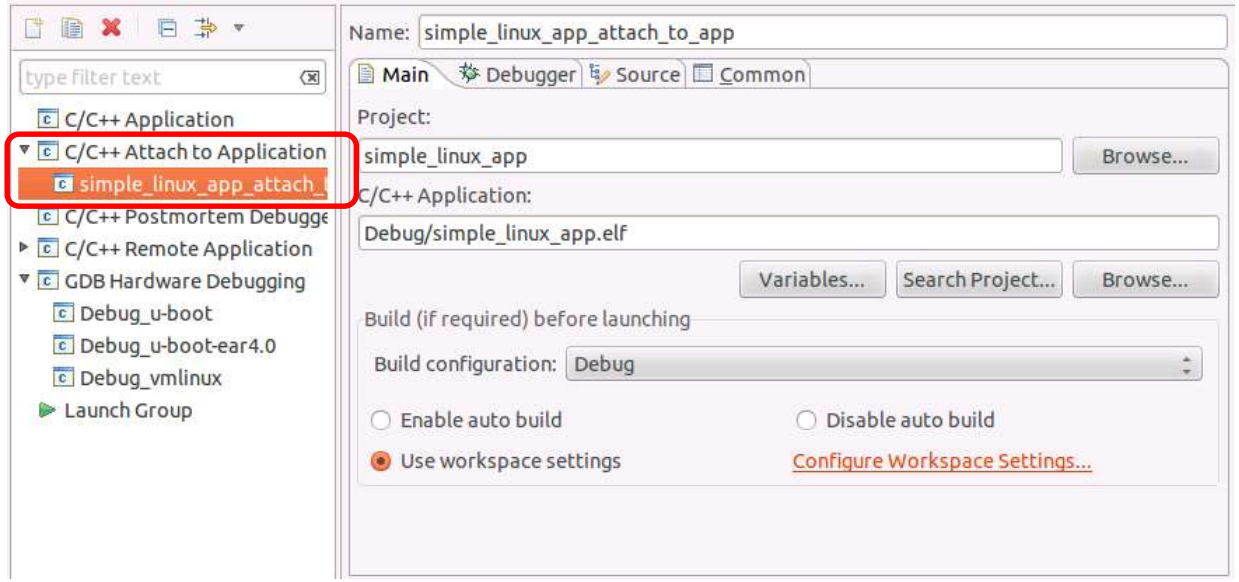
Linux application debug – Attach to a running Linux Application example – Activity (prerequisites)

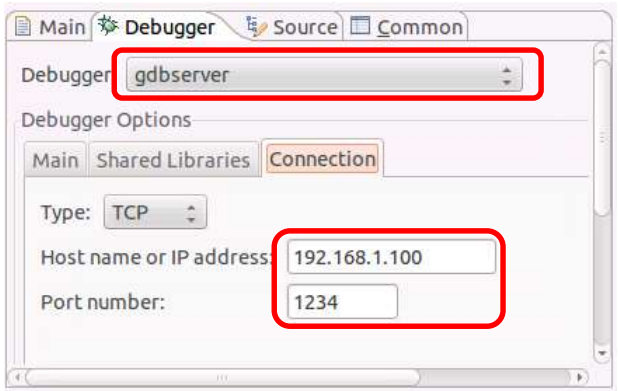
- The application **simple_linux_app** should be copied on target (using a CodeWarrior download session or manually using **scp**)
- Assume a **ssh/telnet console** is active on the target board
- Run the application on target
  - root@ls2085ardb:~# ./simple_linux_app.elf
- This will make sure that the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

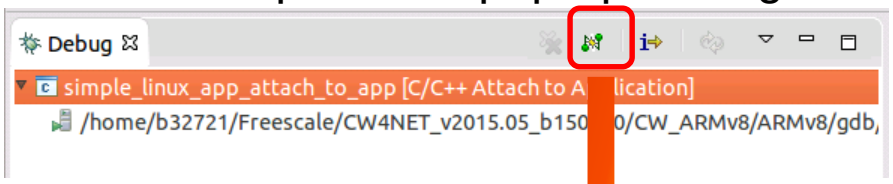
- Open **Debug Configurations: Run → Debug Configurations**
- For C/C++ Attach to Application: create a new launch
- The Main tab will automatically be completed
- In the Debugger tab:
  - Main sub-tab: add gdbinit file
  - Connection sub-tab: set the target parameters: IP address and port
- Press Debug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



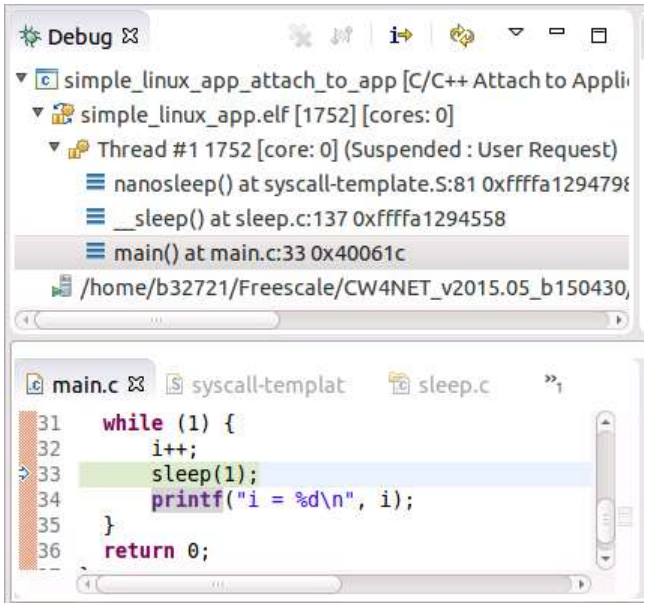


Linux application debug – Attach to a running Linux Application example – Activity

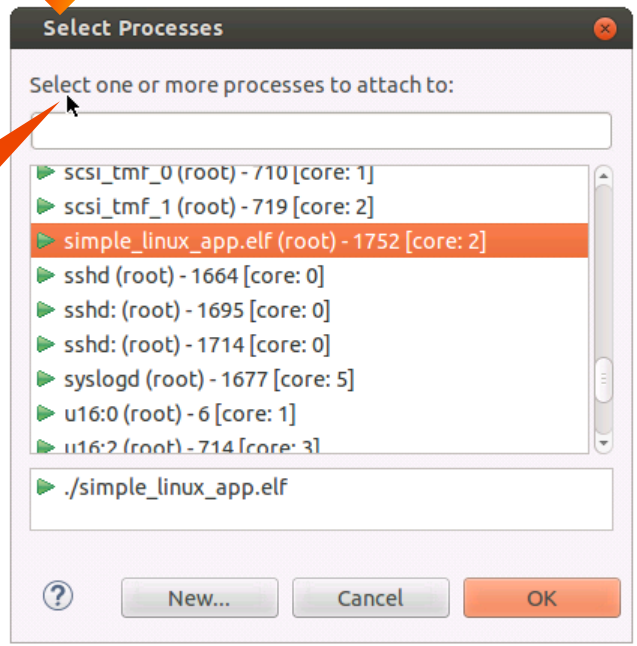
- Hit “Connect to a process” icon to open the pop-up dialog for selecting the application



- 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

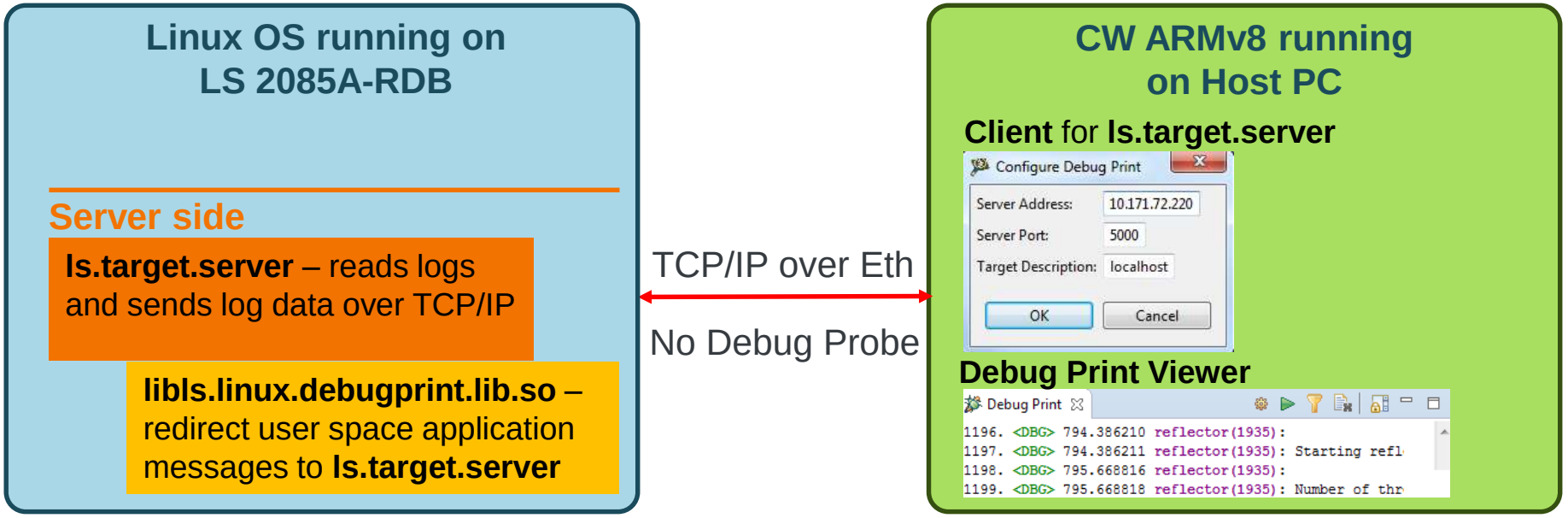
Demonstrate the CodeWarrior Debug Print capabilities

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

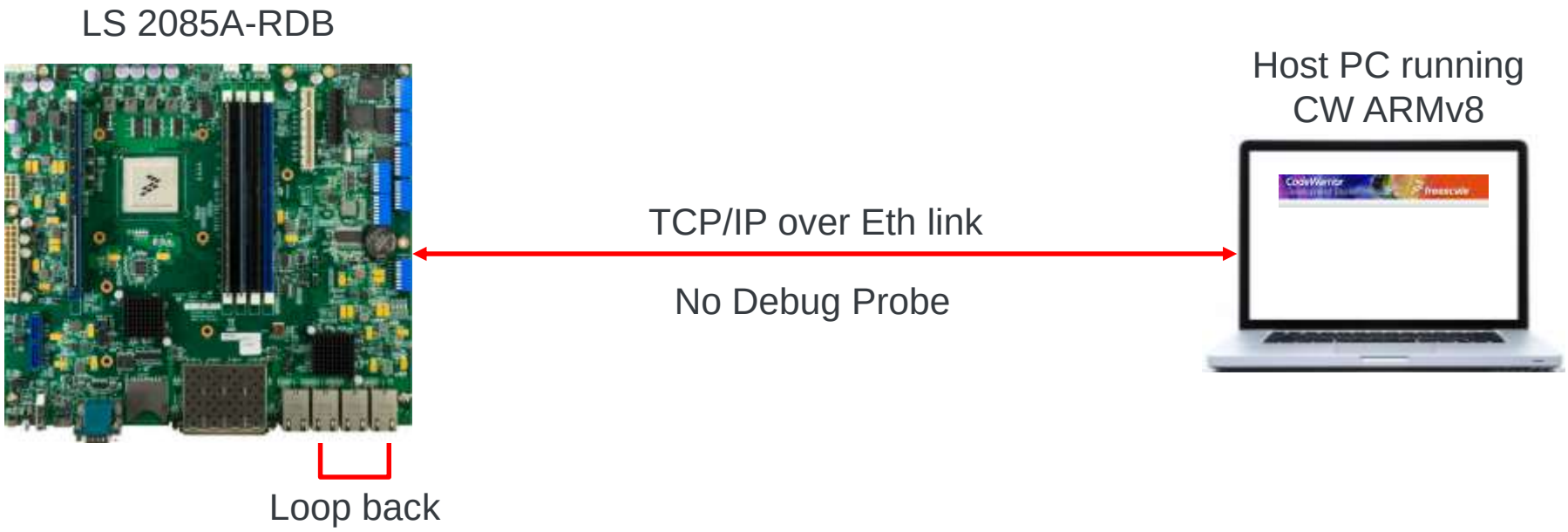
Debug Print – Fundamentals

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



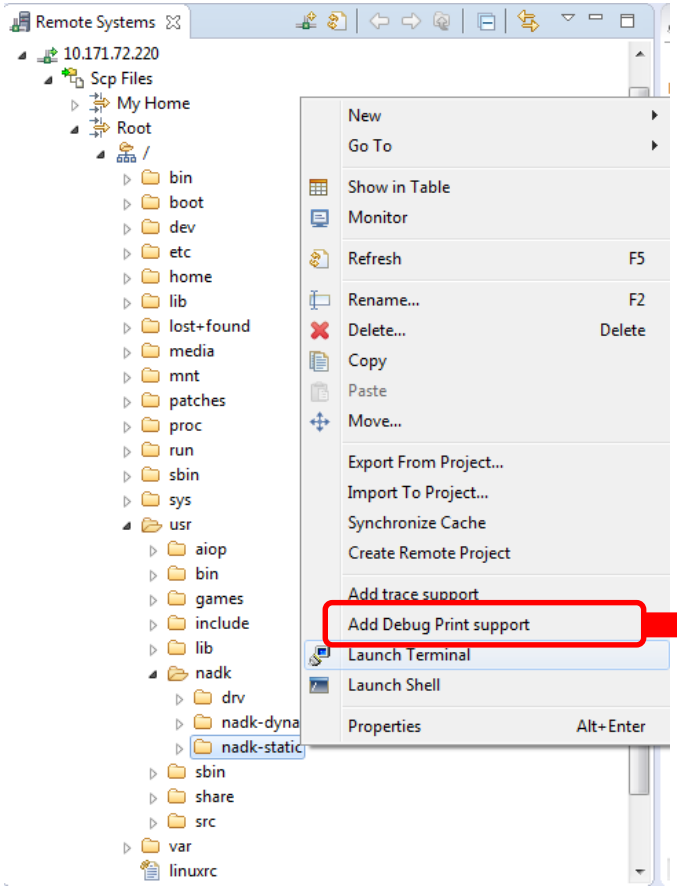
NADK reflector – Hardware setup



- 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]`

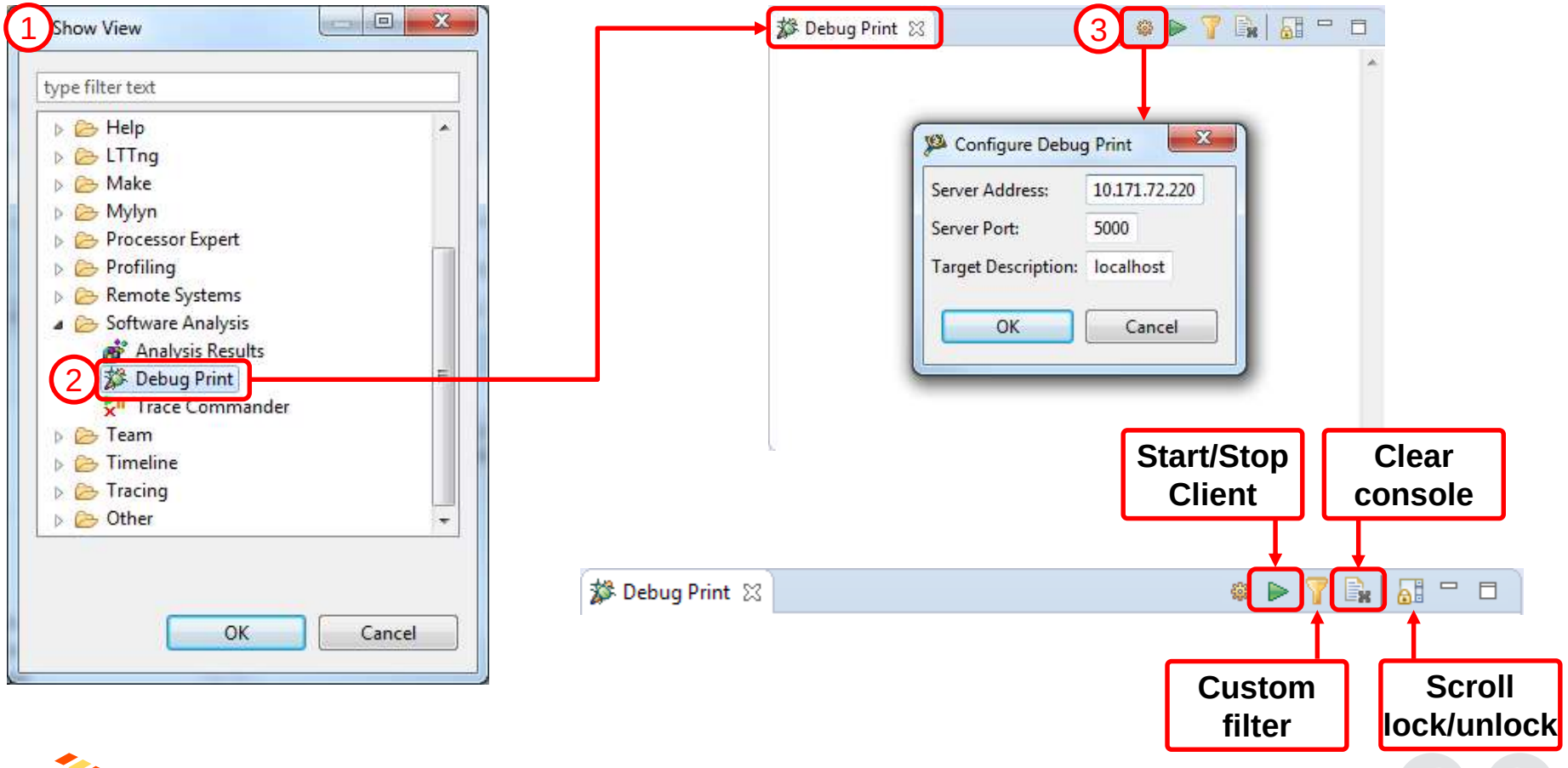
```
PORT   : default 5000
-k     : it does not clear the kernel buffer, but uses an
         internal server logic for determining which are the
         newer messages
```

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

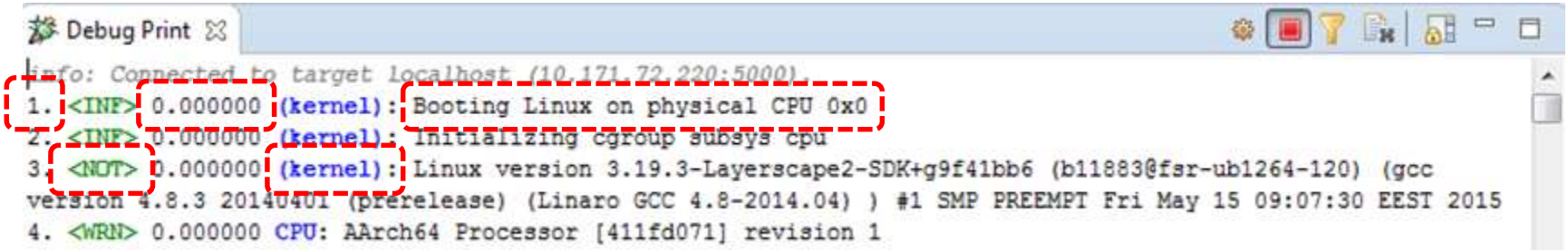
```
root@ls2085ardb:/usr/nadk/nadk-static#./linux.armv8.debugprint/bin/ls.target.server
Using port 5000
Initializing
```

Client Side: setup

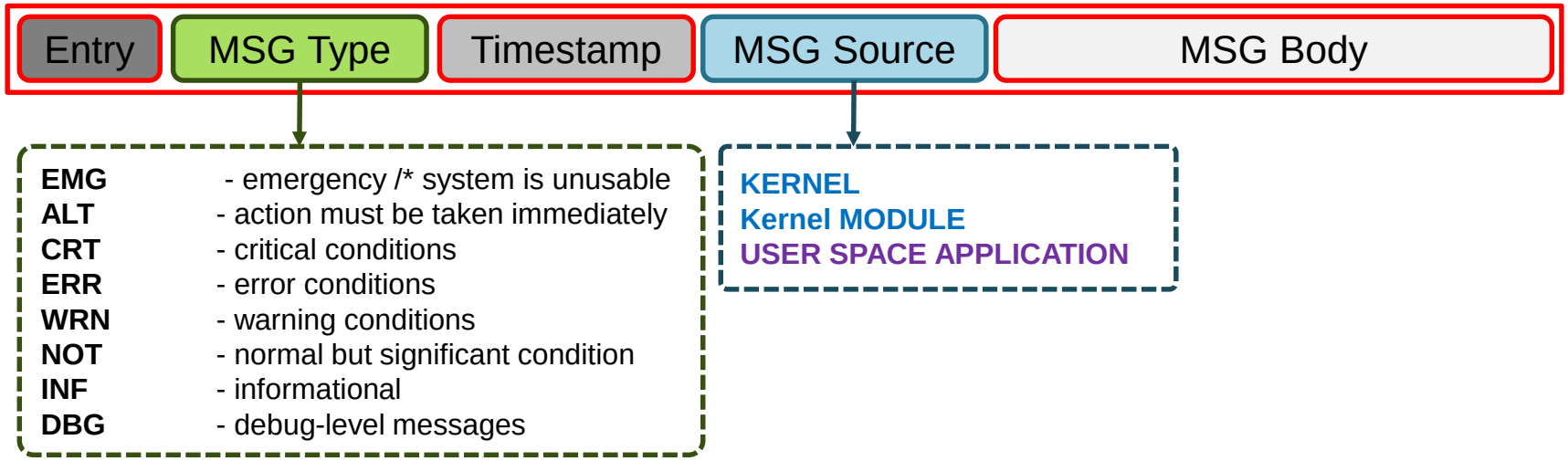
Open the `Debug Print Viewer` and connect the Client with the Server using TCP/IP and Port



Client Side: Debug Print messages format



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
```

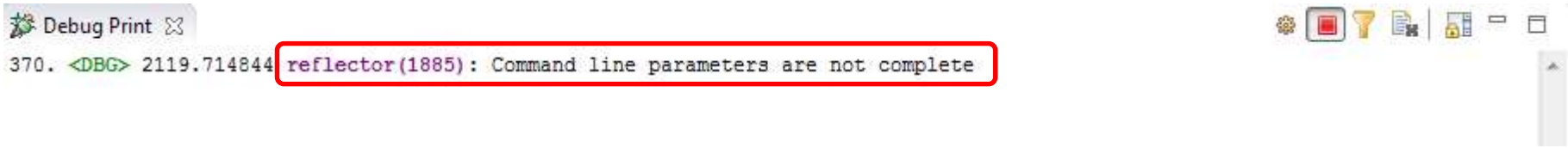
Step 2: Debug Print client will catch all logs during setup:

Debug Print

```
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:



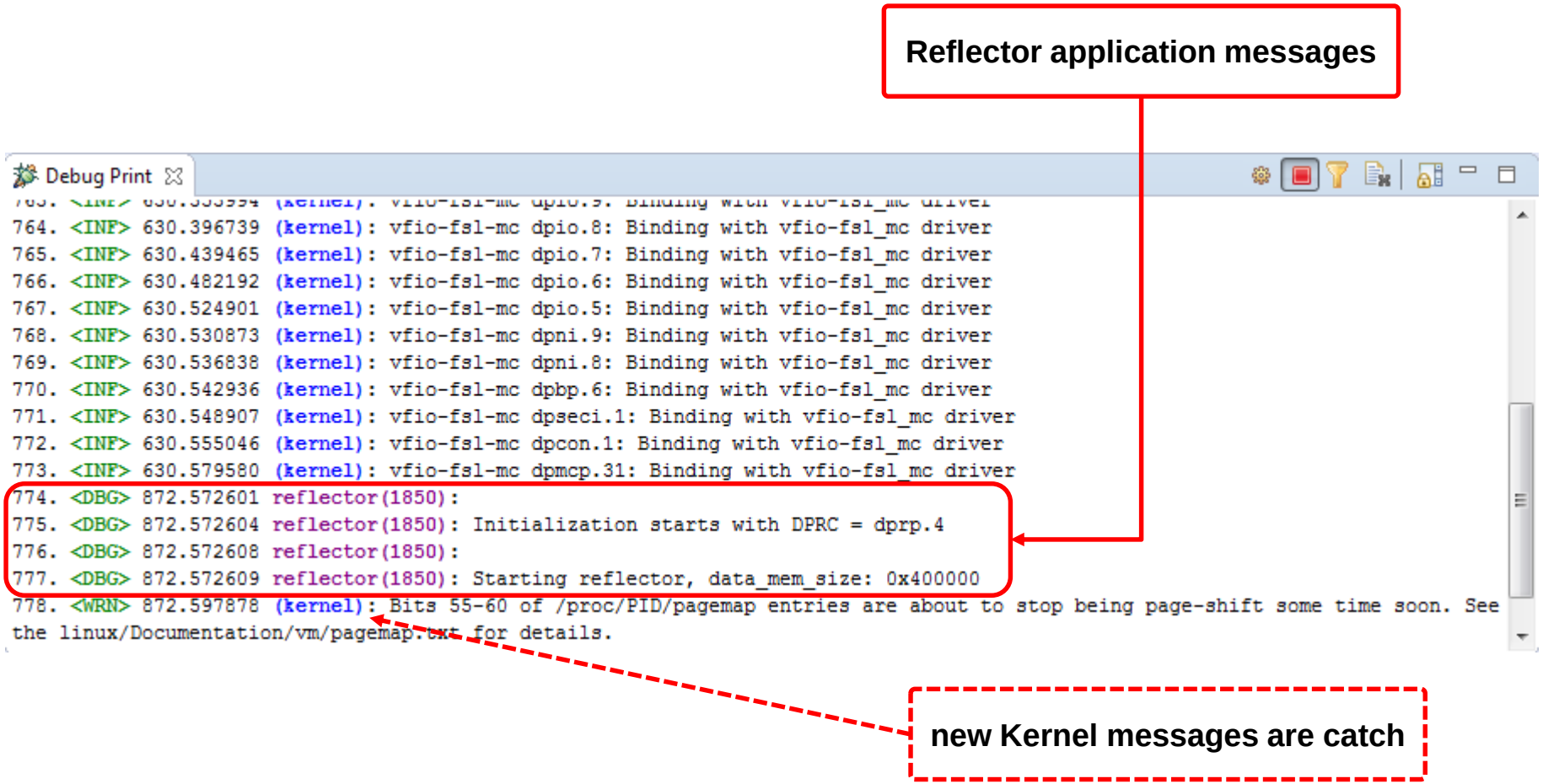
STEP3: redirect nadk-reflector standard output to Server by loading the appropriate library and start the application:

```
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



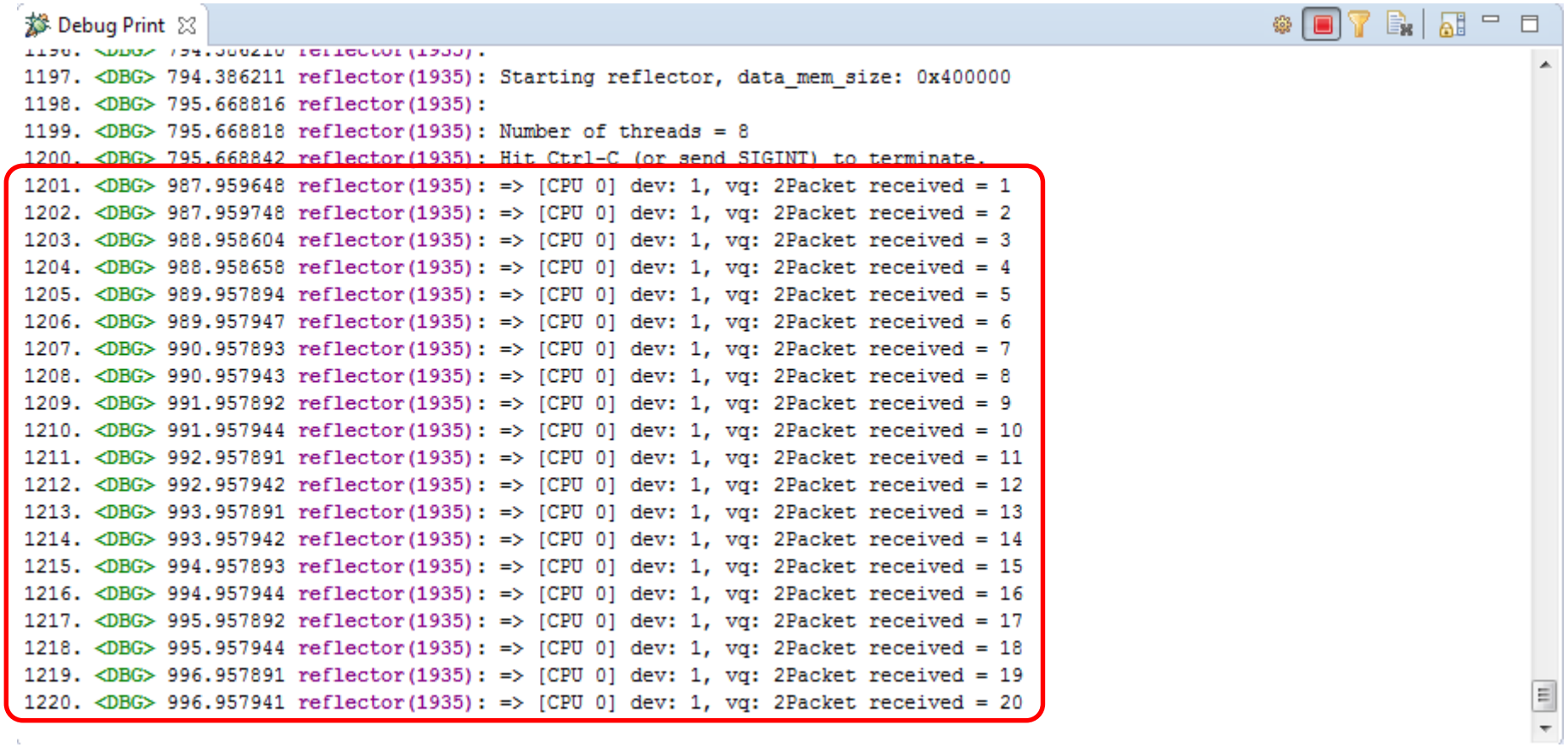
NADK reflector: Debug Print results (cont'd)

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
```

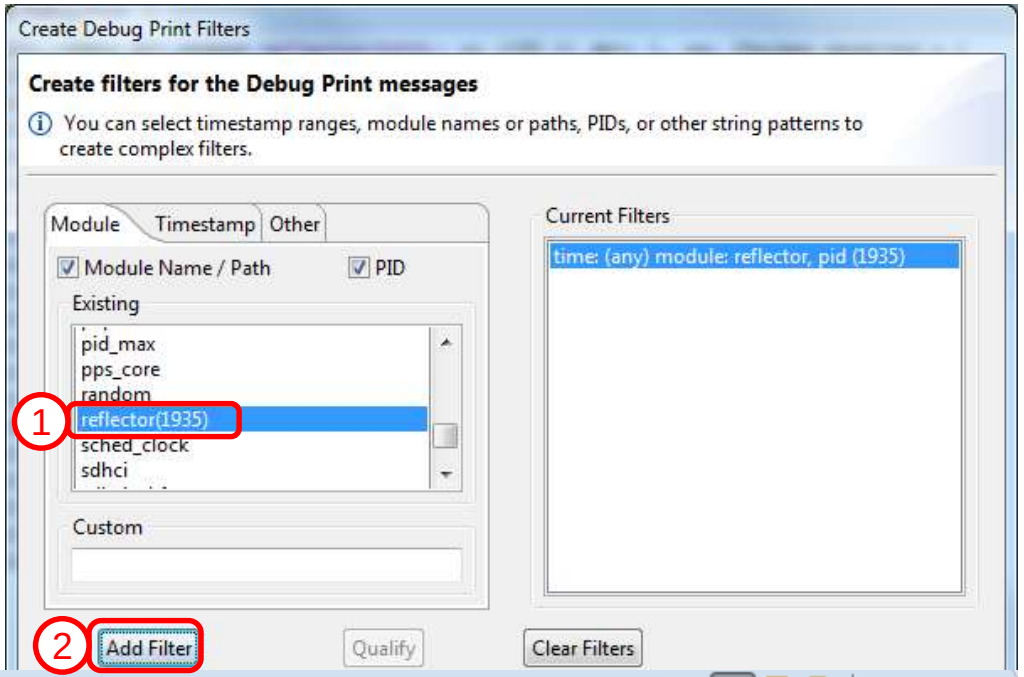
NADK reflector: Debug Print results (cont'd)

Debug Print Client console should catch all nadk-reflector messages



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

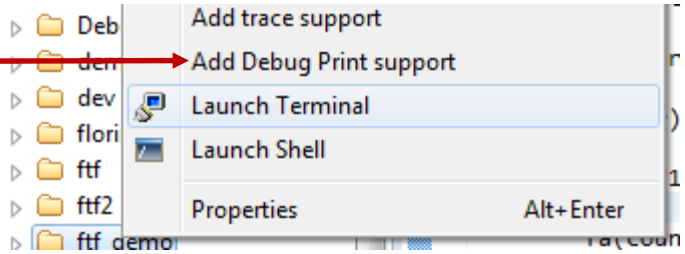
Activity

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):

Add Debug Print support



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

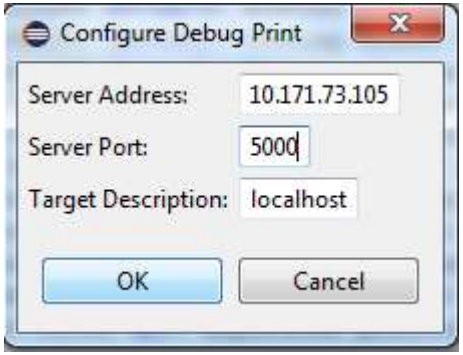
```
root@ls2085aqds:~/dbgPrint# ls
linux.armv8.debugprint
root@ls2085aqds:~/dbgPrint#
```

Linux Debug Print

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

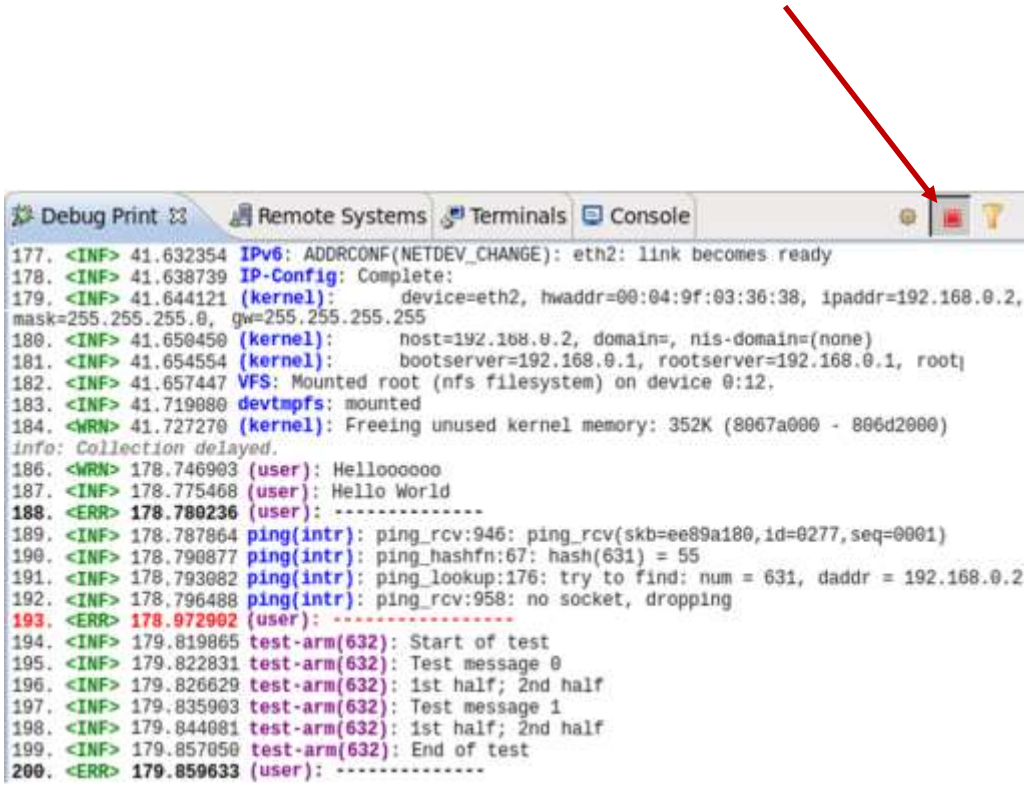
```
root@ls2085aqds:~/dbgPrint# ./linux.armv8.debugprint/bin/ls.target.server
Using port 5000
Initializing
```

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

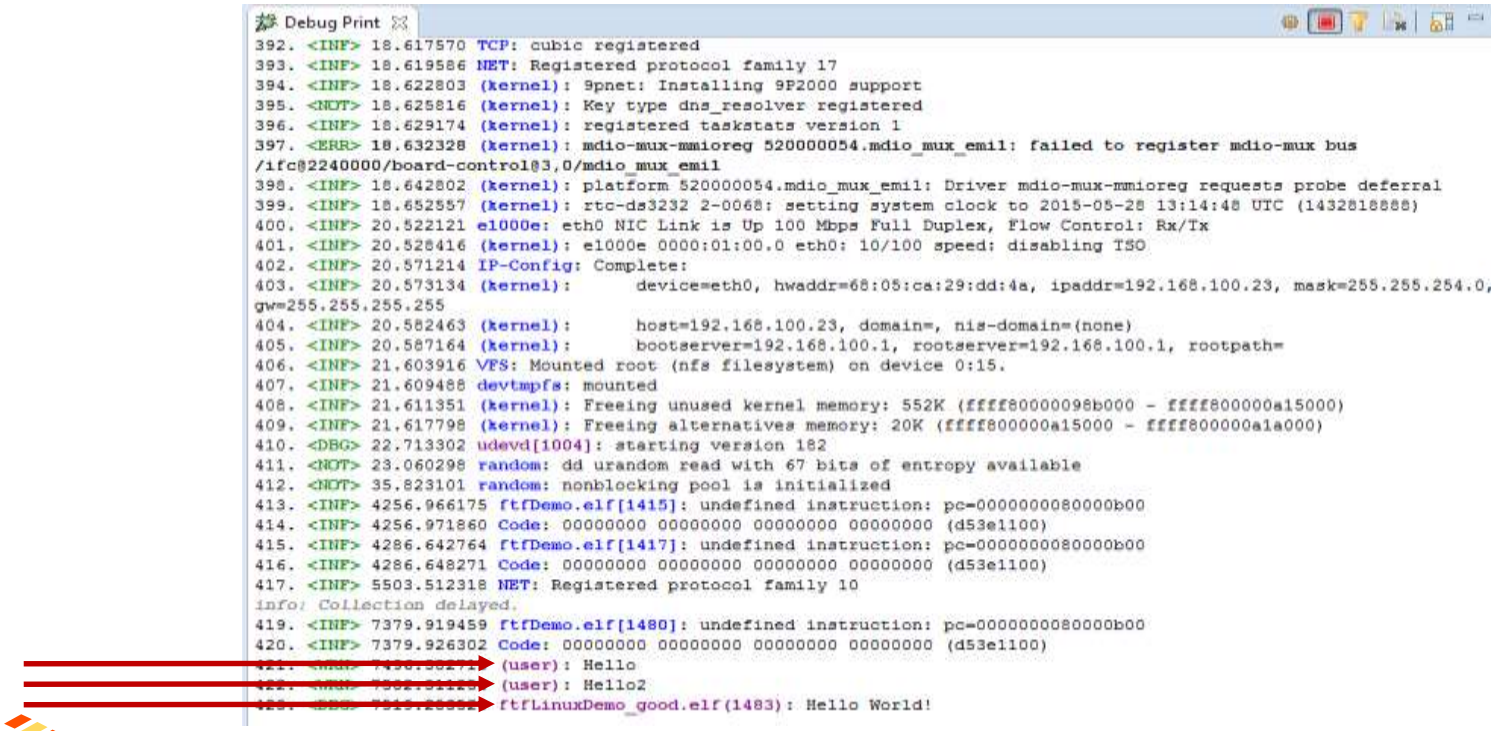
7. Start Debug Print message collection from viewer:



Linux Debug Print

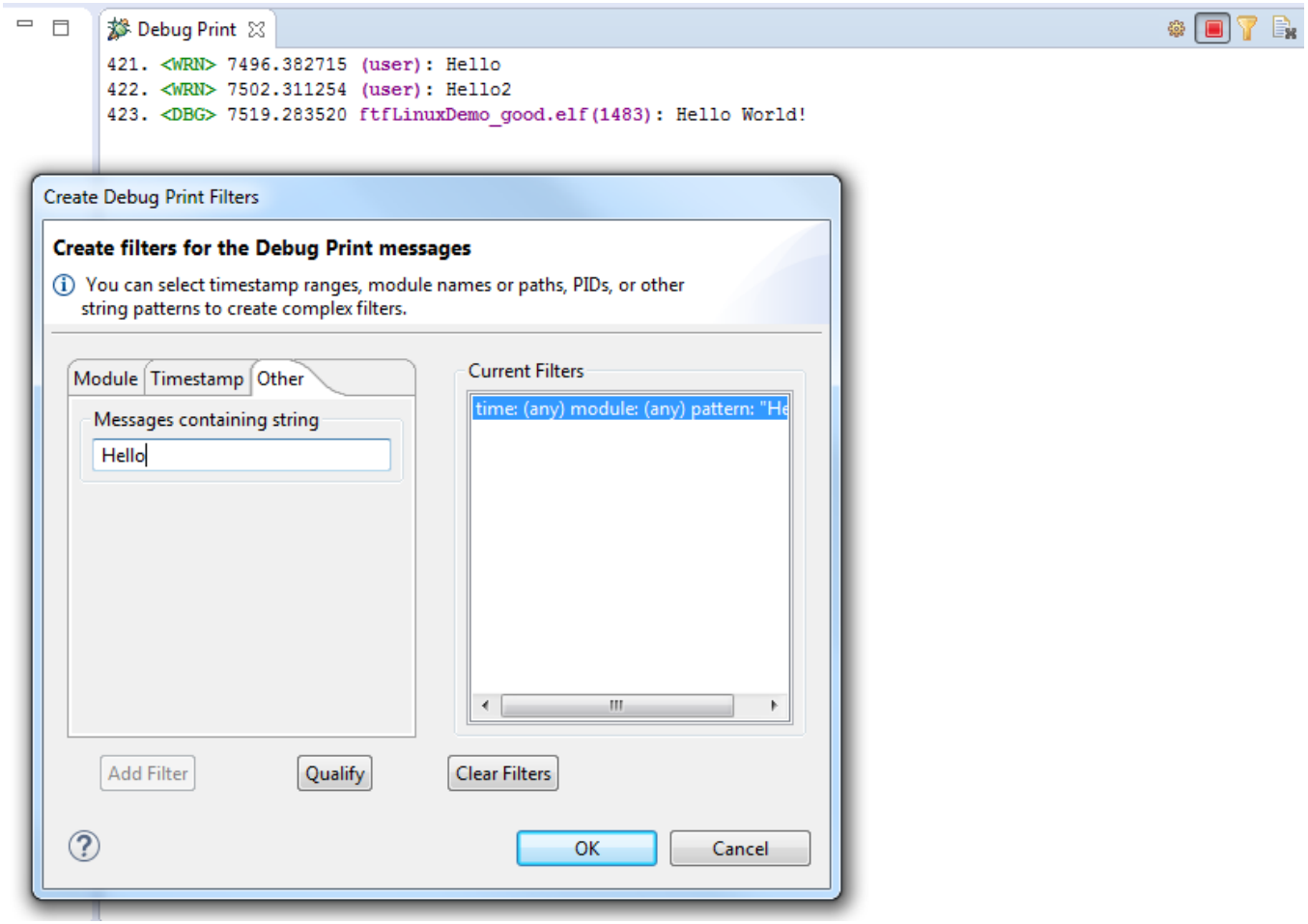
8. From a separate Terminal launch some user applications that contains some *printf()* calls or *echo* some messages:

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



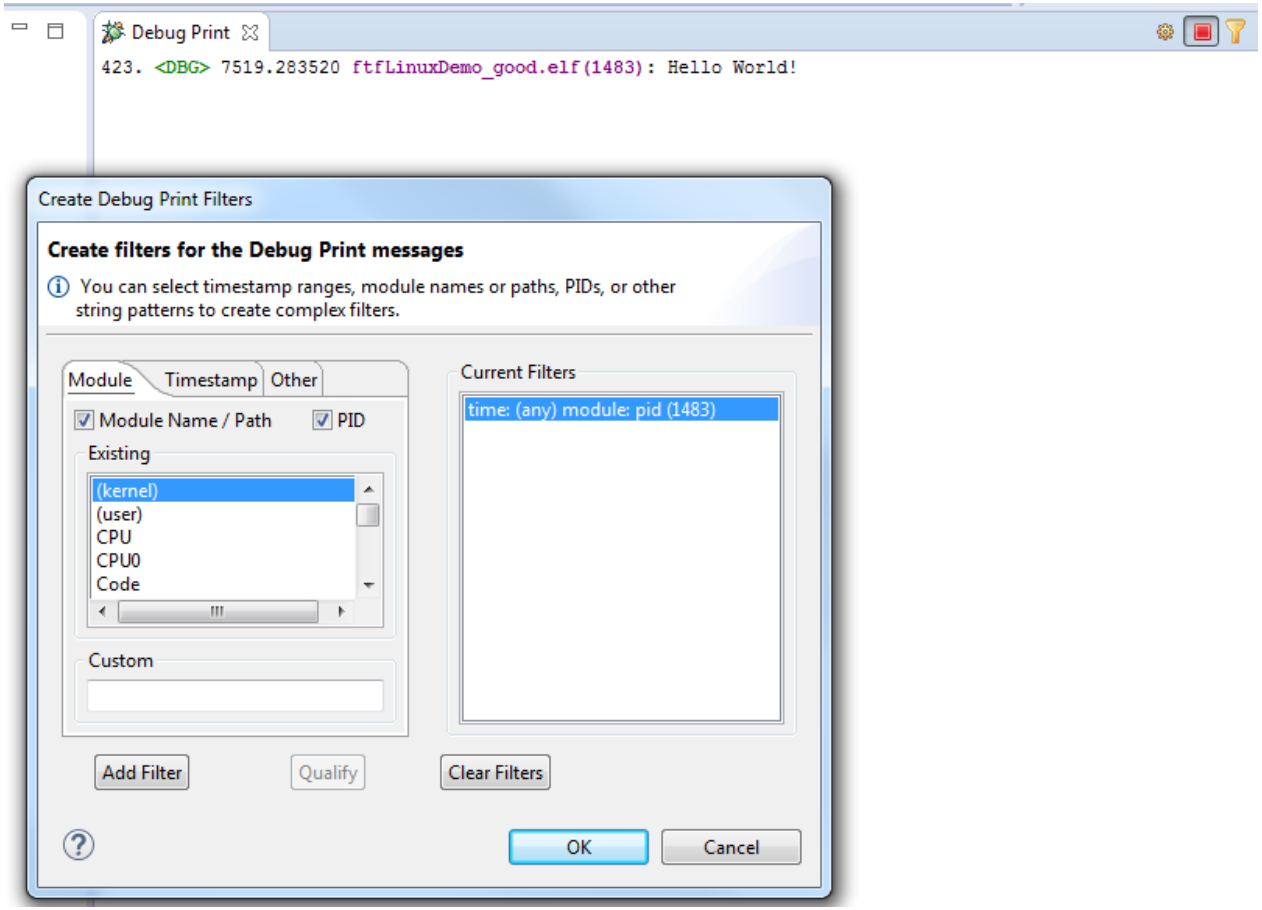
Linux Debug Print

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

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

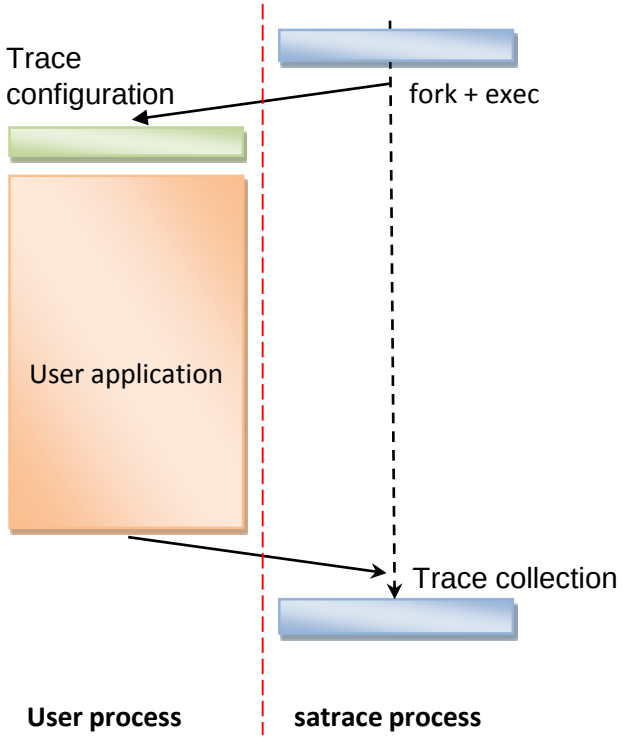
Ethernet cable + `linux.armv8.satrace`

Hardware Probe using JTAG
(E.g. CodeWarrior USB T

Linux probeless trace – Command Line API

```
This application starts and collects trace on target.

Usage: ./linux.armv8.satrace/bin/ls.linux.satrace [Options] app [app_args]

User space options:
  -A [ --archive-file ] arg (=[app_name].cwzsa)       Archive path
  -b [ --backtrace ]                                  Shows backtrace on
                                                      SEGFAULT.
        [app_name] - Name of the traced application.

Kernel space options:
  -K [ --kernel ] path          Archive path.
  -i [ --kernel-image ] path    vmlinux image compiled with debugging symbols.
  -m [ --module-name ] mod_name Name of the traced module.

System trace options:
  -S [ --system ]               Starts the system trace

General options:
  -v [ --verbose ]      Verbose mode
  -V [ --version ]      Product version
  -h [ --help ]         Displays this help message

Note:
        Do not mix kernel and user space options, otherwise all user space options will be ignored.
        The kernel space trace will be collected after catching the SIGINT signal (CTRL+C).

 Examples :
        ./linux.armv8.satrace/bin/ls.linux.satrace -A archive.cwzsa ./my_app
        ./linux.armv8.satrace/bin/ls.linux.satrace ./my_app my_arg1 my_arg2
        ./linux.armv8.satrace/bin/ls.linux.satrace -K kernelTest
        ./linux.armv8.satrace/bin/ls.linux.satrace -K kernelTest -i ~/vmlinux
```

Linux user space application trace *(command line)*



Activity

Linux application trace

Linux probeless trace *(CodeWarrior)*

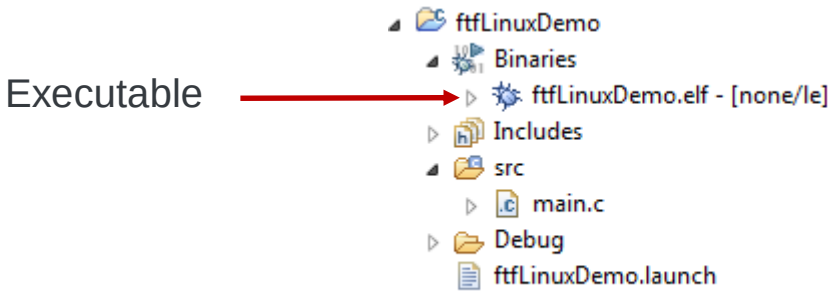
1. Create and build a Linux C/C++ application
2. You may use a code like this:
3. Build the project
4. Create or use an existing RSE connection
5. Verify that you can see the target file system

main.c

```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;
}
```

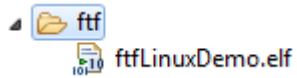
Linux probeless trace *(CodeWarrior)*

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



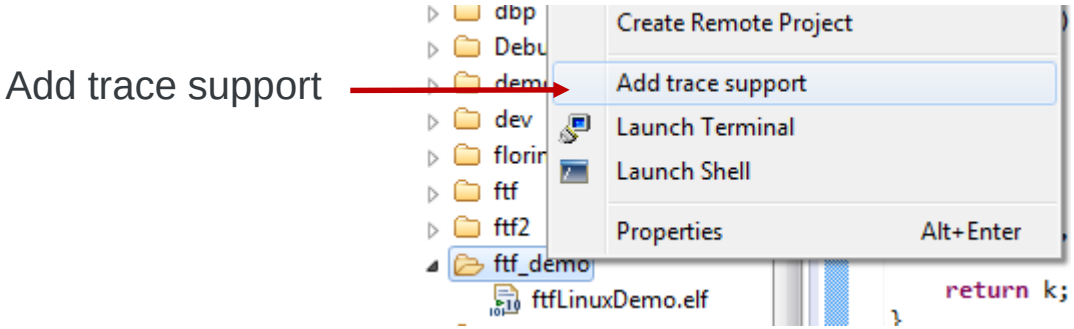
7. Create a demo folder into target file system

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

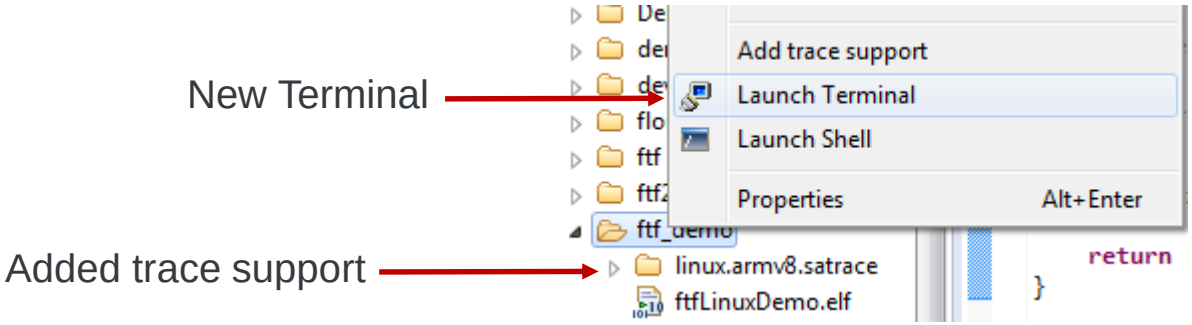


Linux probeless trace *(CodeWarrior)*

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

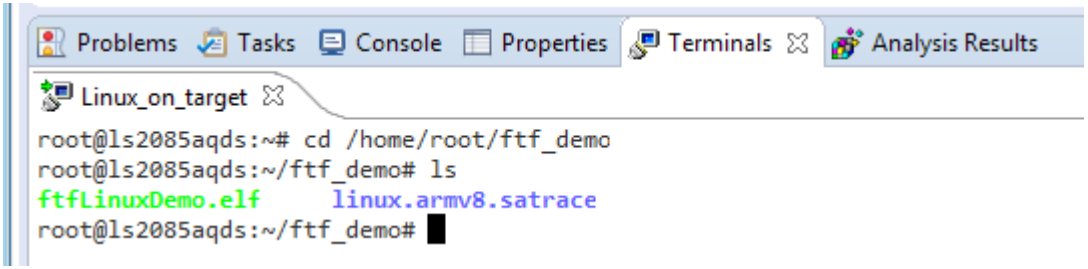


10. Launch a Terminal



Linux probeless trace *(CodeWarrior)*

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



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)*

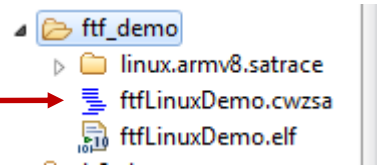
13. The results will be:

```
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#
```

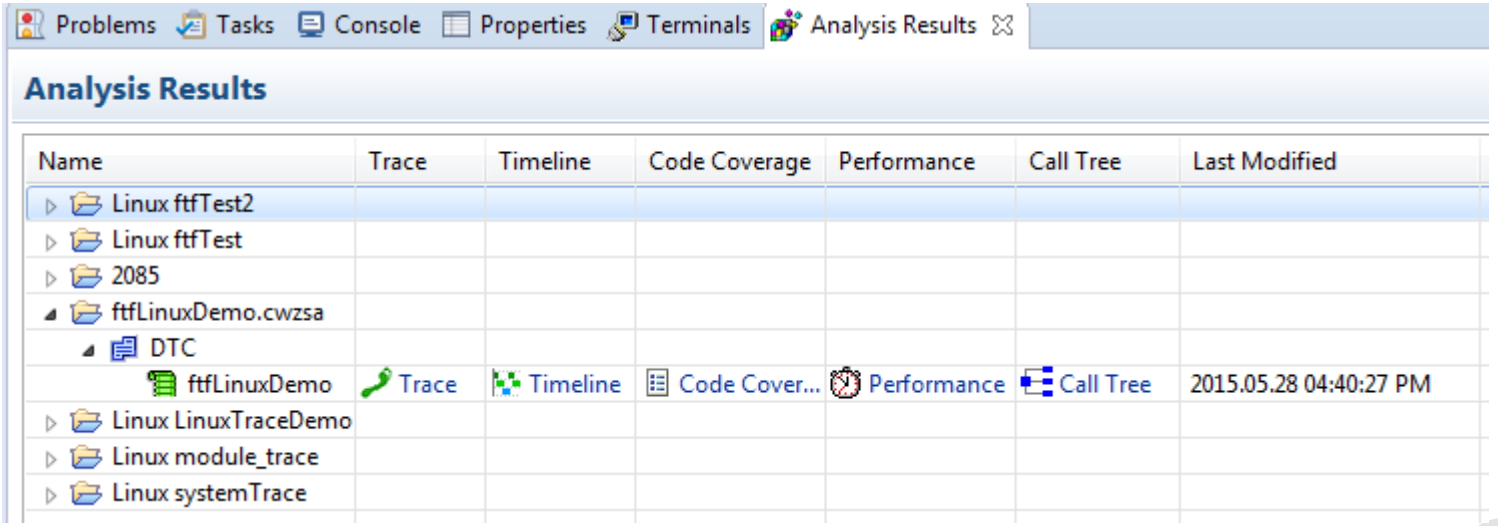
Linux probeless trace *(CodeWarrior)*

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

Trace results file



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



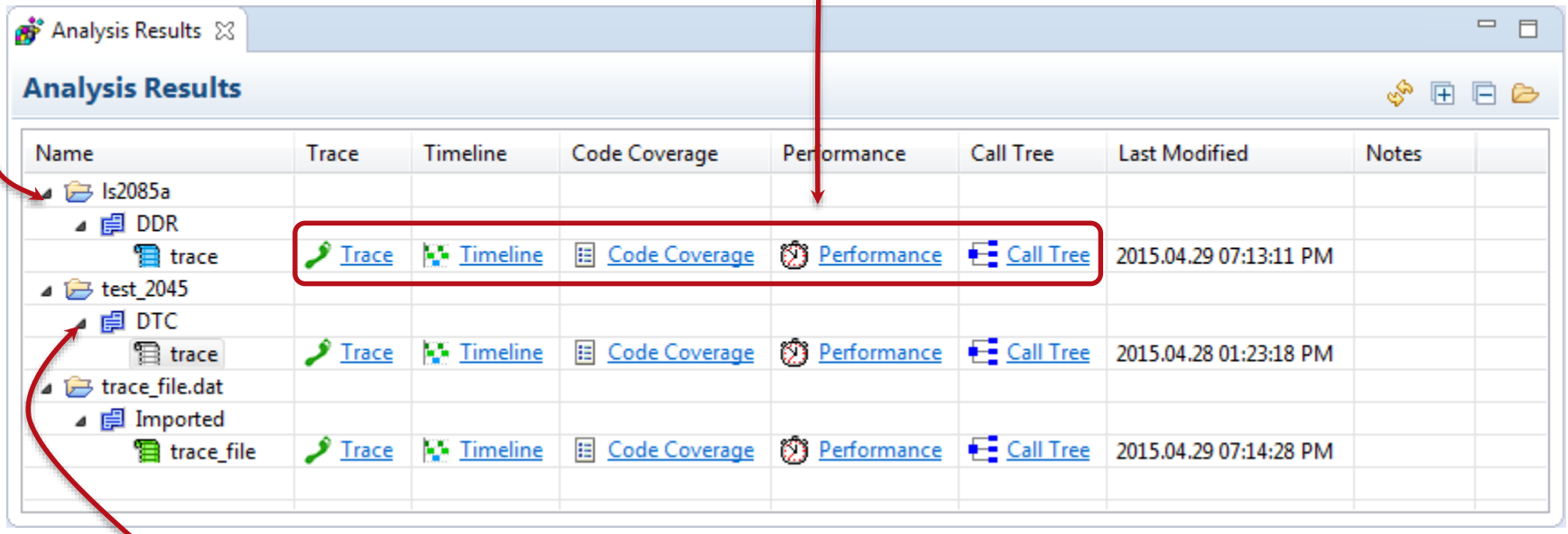
Linux trace – view results

Analysis Results content

Links

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

Platform configuration
used to collect trace

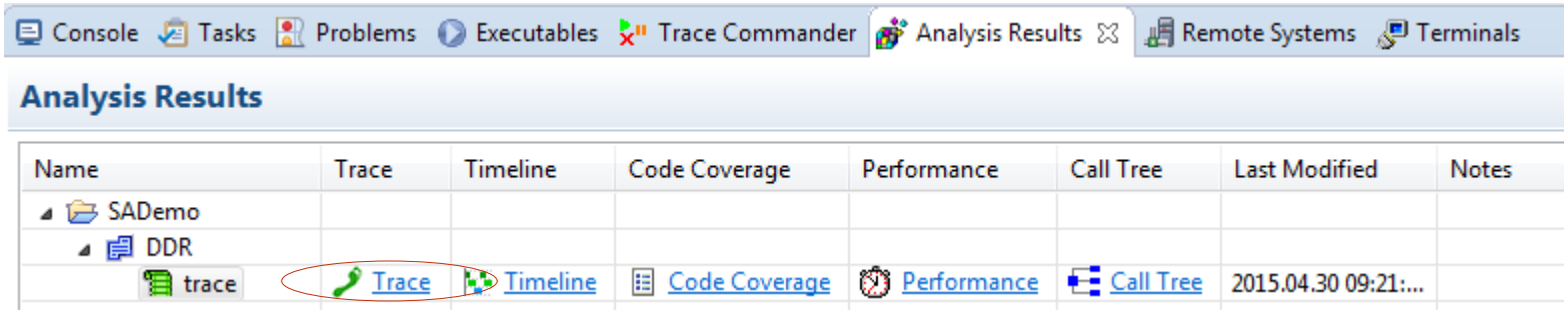


Data source from where the trace was collected:

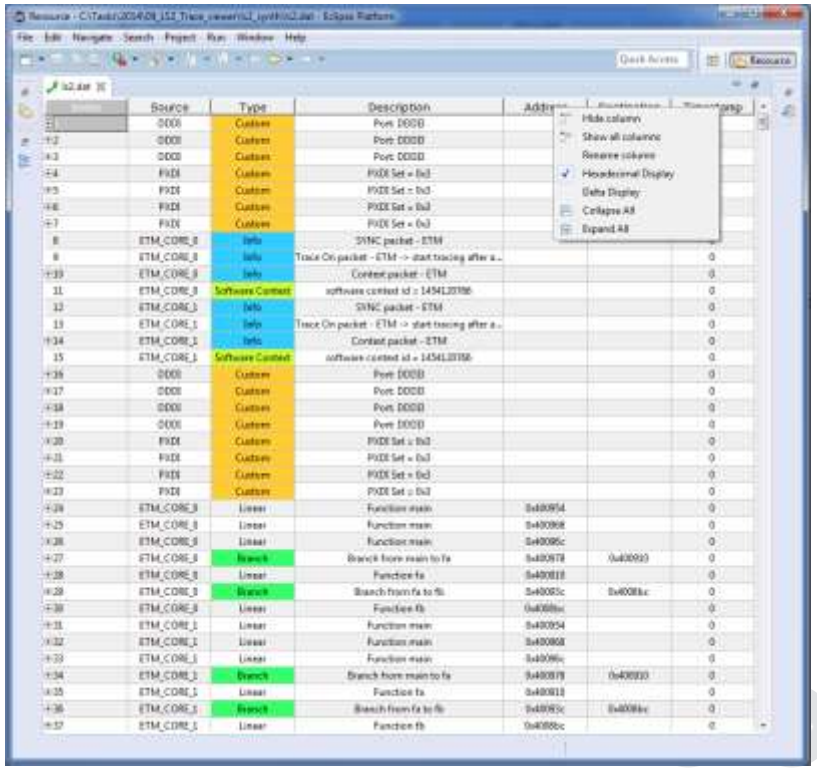
- ➢ DTC
- ➢ DDR
- ➢ or Imported trace

Trace Viewer

1. From results folder:



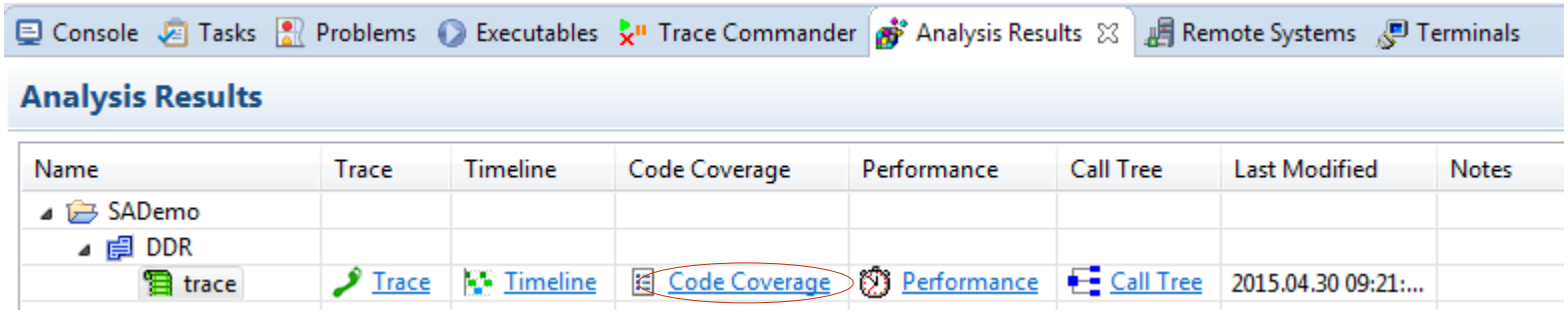
2. Open Trace Viewer:



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

Code coverage

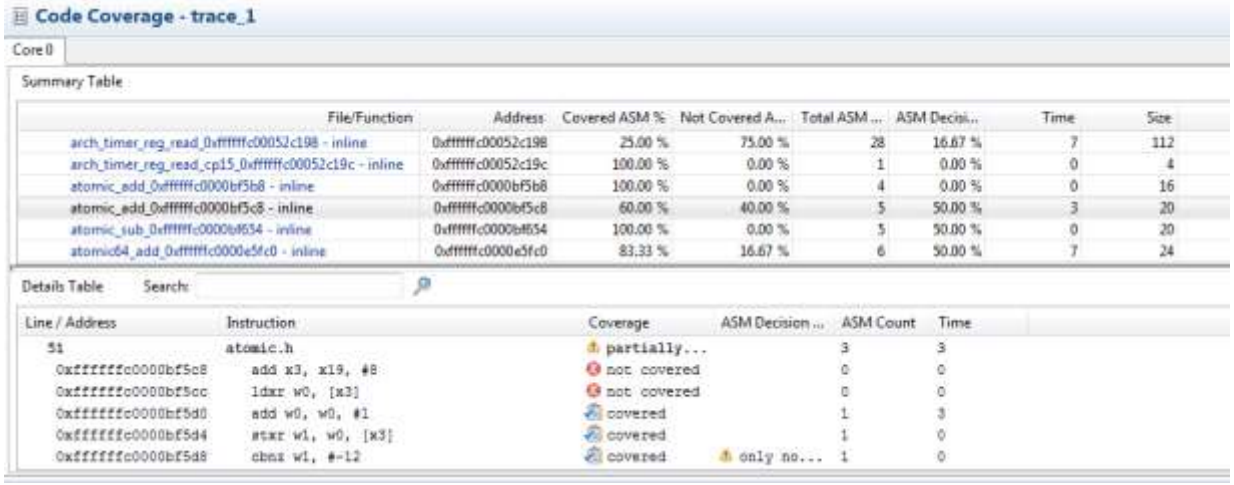
1. From results folder:



2. Open code coverage viewer:

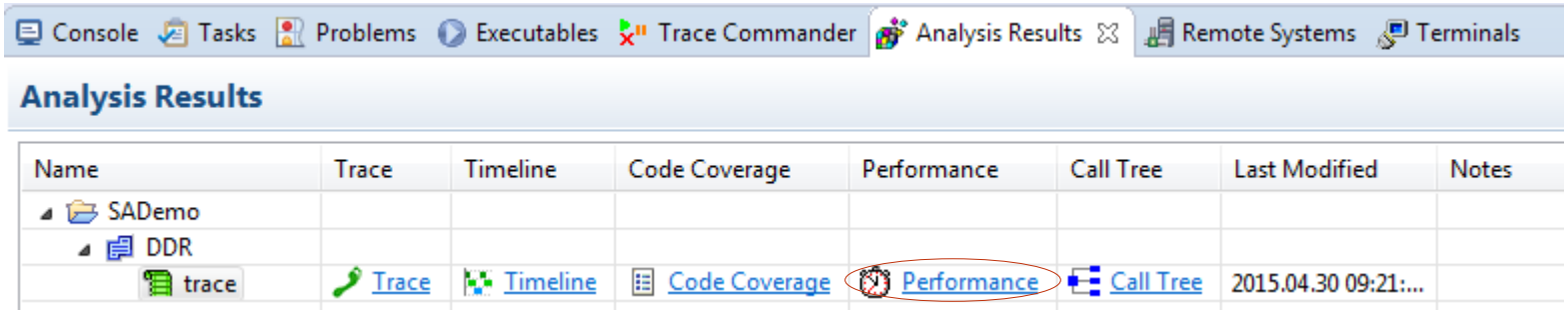
Split pane with 2 types of info:

- ✓ Summary table displaying statistics for each function
- ✓ Details table displaying line-by-line coverage of selected function



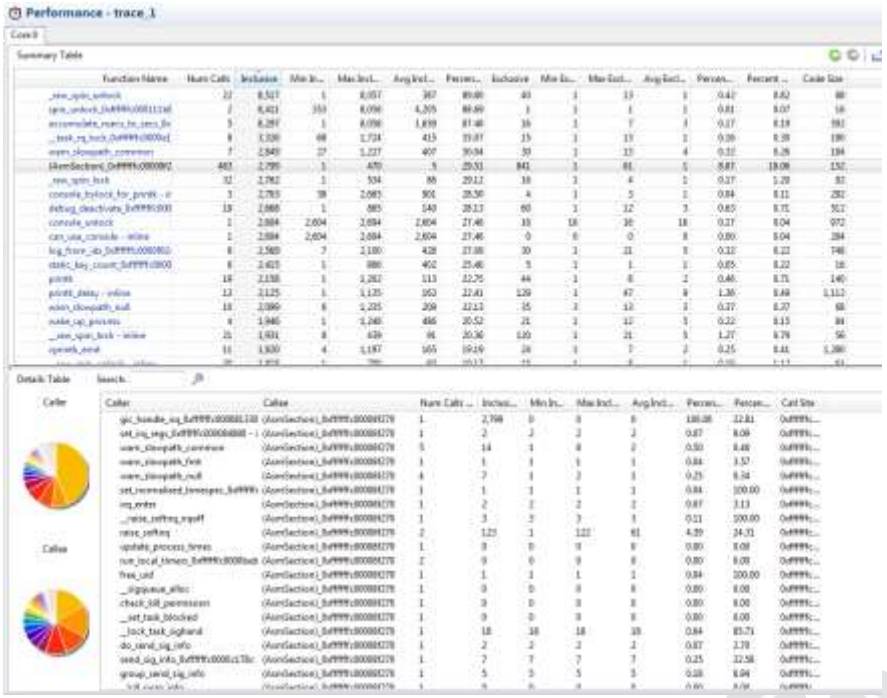
Performance profiler

1. From results folder:



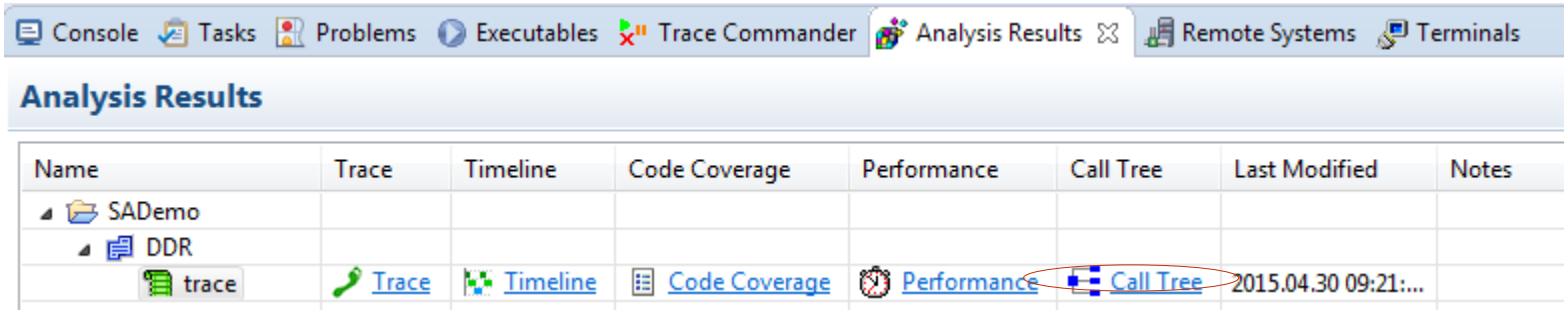
2. Open performance viewer:

- ✓ 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

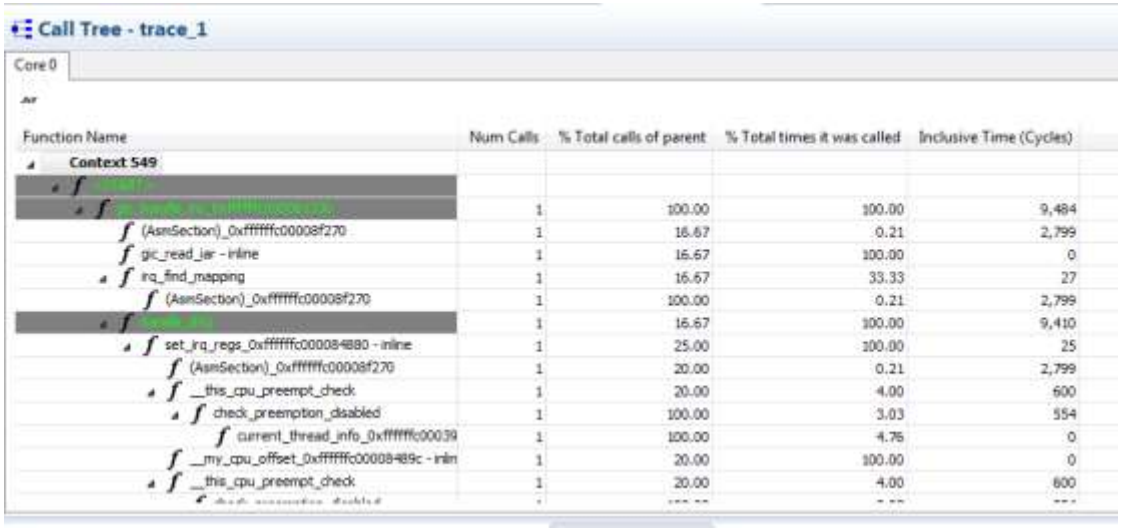


Call tree profiler

1. From results folder:



2. Open call tree viewer:



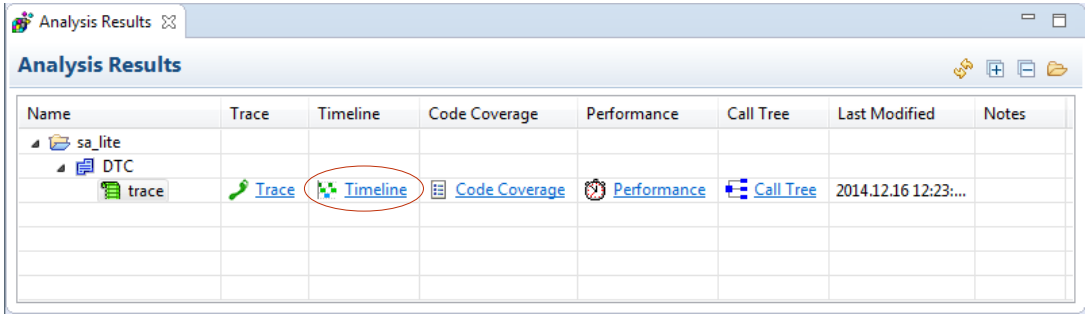
Shows call tree of executed functions. Two highlighted paths:

✓ Green color shows critical path

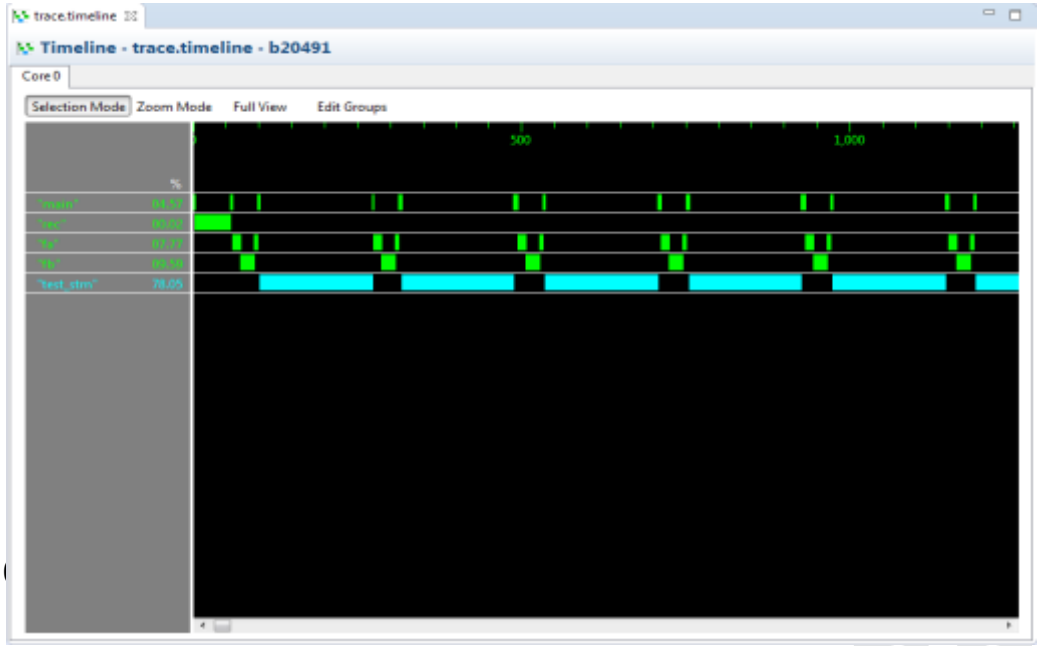
✓ Grey background shows max stack path

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 an

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

http://www.freescale.com/codewarrior

Q&A

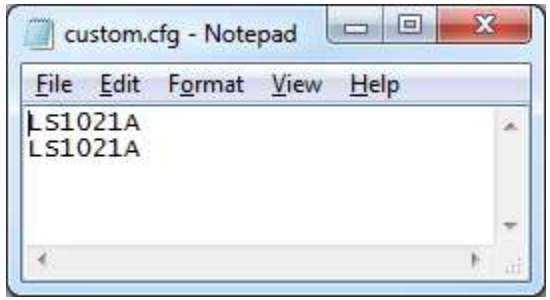
www.Freescale.com

BACKUP

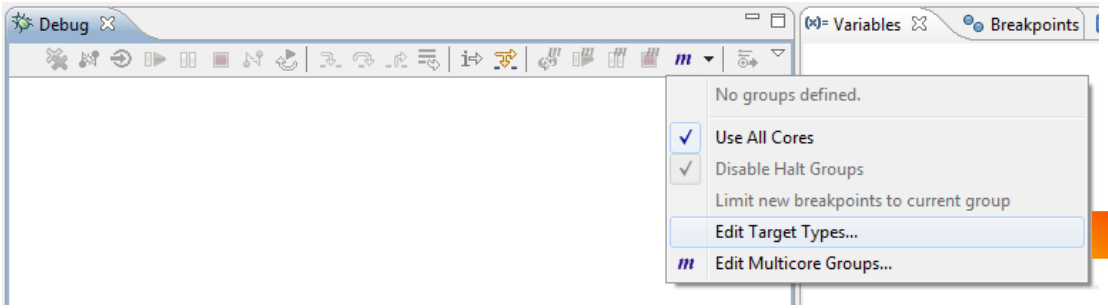
Custom Board Setup

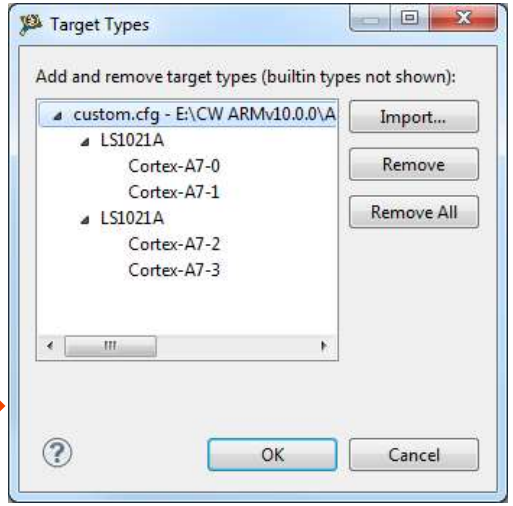
Stock Freescale Boards vs. Customer Designs

- 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:



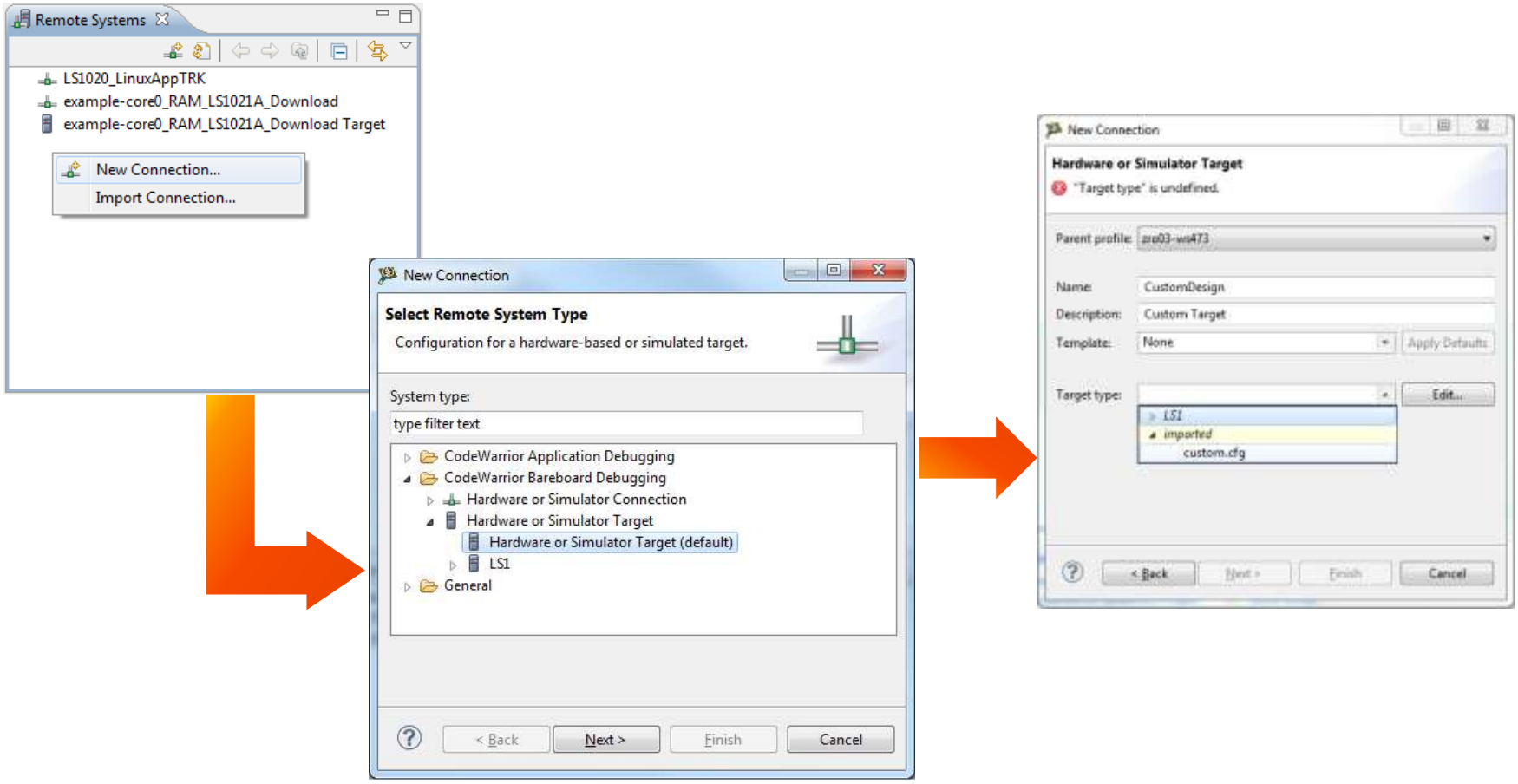
- Import it.





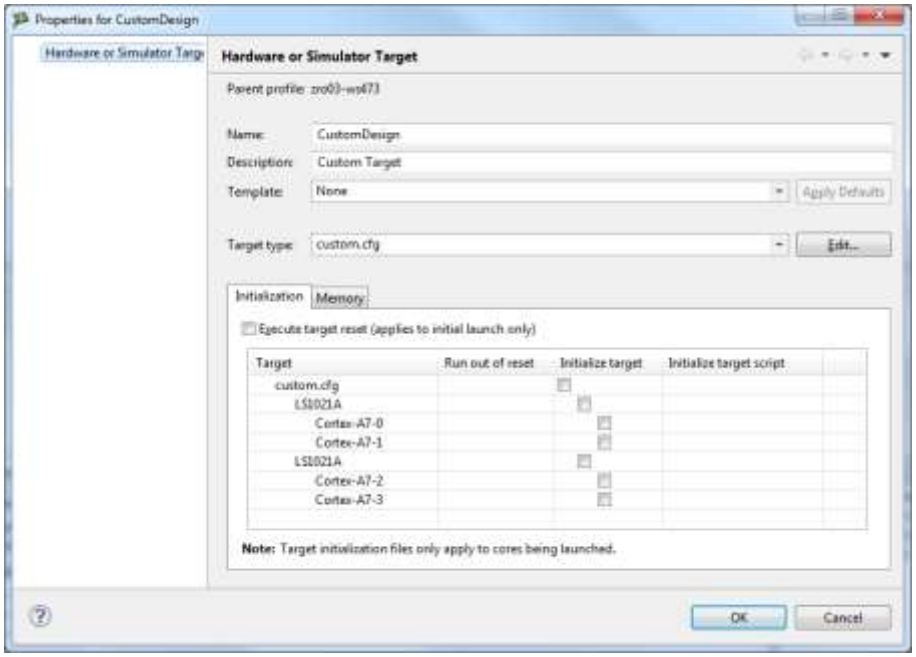
Creating a Custom Target System

- Create a new target system



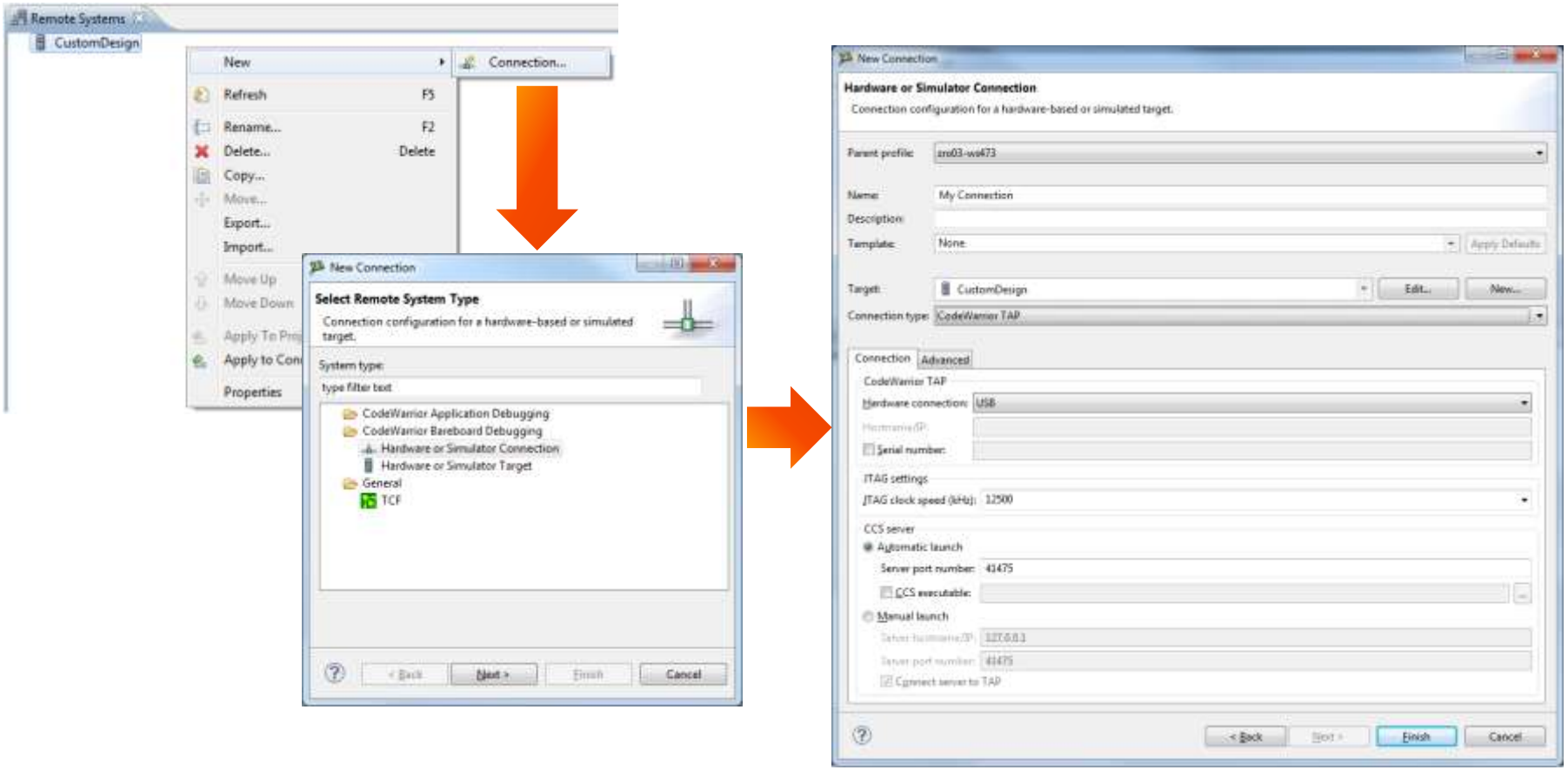
Configure the Newly Created Target

- 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